MMEditing

MMEditing Authors

Feb 21, 2023

COMMUNITY

Languages: English |

MMEditing is an open-source toolbox for image and video processing, editing and synthesis.

MMEditing supports various foundamental generative models, including:

- Unconditional Generative Adversarial Networks (GANs)
- Conditional Generative Adversarial Networks (GANs)
- Internal Learning
- Diffusion Models
- And many other generative models are coming soon!

MMEditing supports various applications, including:

- Image super-resolution
- Video super-resolution
- Video frame interpolation
- Image inpainting
- Image matting
- Image-to-image translation
- And many other applications are coming soon!

MMEditing is based on PyTorch and is a part of the OpenMMLab project. Codes are available on GitHub.

CHAPTER

ONE

DOCUMENTATION

1.1 Contributing guidance

Welcome to the MMEditing community, we are committed to building a toolbox for cutting-edge image, video and 3D content generation, editing and processing techniques.

This section introduces following contents:

- *Pull Request Workflow*
  - 1. fork and clone
  - 2. configure pre-commit
  - 3. create a development branch
  - 4. commit the code and pass the unit test
  - 5. push the code to remote
  - 6. create a pull request
  - 7. resolve conflicts
- *Guidance*
  - unit test
  - document rendering
- *Code Style*
  - Python
  - C++ and CUDA
  - *PR Specs*

All kinds of contributions are welcomed, including but not limited to

Fix bug

You can directly post a Pull Request to fix typo in code or documents

The steps to fix the bug of code implementation are as follows.

1. If the modification involve significant changes, you should create an issue first and describe the error information and how to trigger the bug. Other developers will discuss with you and propose an proper solution.
2. Posting a pull request after fixing the bug and adding corresponding unit test.

New Feature or Enhancement

1. If the modification involve significant changes, you should create an issue to discuss with our developers to propose an proper design.
2. Post a Pull Request after implementing the new feature or enhancement and add corresponding unit test.

Document

You can directly post a pull request to fix documents. If you want to add a document, you should first create an issue to check if it is reasonable.

1.1.1 Pull Request Workflow

If you're not familiar with Pull Request, don't worry! The following guidance will tell you how to create a Pull Request step by step. If you want to dive into the develop mode of Pull Request, you can refer to the official documents

1. Fork and clone

If you are posting a pull request for the first time, you should fork the OpenMMLab repositories by clicking the **Fork** button in the top right corner of the GitHub page, and the forked repositories will appear under your GitHub profile.

Then, you can clone the repositories to local:

```
git clone git@github.com:{username}/mmediting.git
```

After that, you should ddd official repository as the upstream repository

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

Check whether remote repository has been added successfully by `git remote -v`

```
origin        git@github.com:{username}/mmediting.git (fetch)
origin        git@github.com:{username}/mmediting.git (push)
upstream        git@github.com:open-mmlab/mmediting (fetch)
upstream        git@github.com:open-mmlab/mmediting (push)
```

Note: Here's a brief introduction to origin and upstream. When we use "git clone", we create an "origin" remote by default, which points to the repository cloned from. As for "upstream", we add it ourselves to point to the target repository. Of course, if you don't like the name "upstream", you could name it as you wish. Usually, we'll push the code to "origin". If the pushed code conflicts with the latest code in official("upstream"), we should pull the latest code from upstream to resolve the conflicts, and then push to "origin" again. The posted Pull Request will be updated automatically.

2. Configure pre-commit

You should configure pre-commit in the local development environment to make sure the code style matches that of OpenMMLab. **Note**: The following code should be executed under the mmediting directory.

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

Check that pre-commit is configured successfully, and install the hooks defined in `.pre-commit-config.yaml`.

```
pre-commit run --all-files
```

Note: Chinese users may fail to download the pre-commit hooks due to the network issue. In this case, you could download these hooks from gitee by setting the .pre-commit-config-zh-cn.yaml

pre-commit install -c .pre-commit-config-zh-cn.yaml pre-commit run –all-files -c .pre-commit-config-zh-cn.yaml

If the installation process is interrupted, you can repeatedly run `pre-commit run ...` to continue the installation.

If the code does not conform to the code style specification, pre-commit will raise a warning and fixes some of the errors automatically.

If we want to commit our code bypassing the pre-commit hook, we can use the `--no-verify` option(**only for temporarily commit**.

```
git commit -m "xxx" --no-verify
```

3. Create a development branch

After configuring the pre-commit, we should create a branch based on the master branch to develop the new feature or fix the bug. The proposed branch name is `username/pr_name`

```
git checkout -b yhc/refactor_contributing_doc
```

In subsequent development, if the master branch of the local repository is behind the master branch of "upstream", we need to pull the upstream for synchronization, and then execute the above command:

```
git pull upstream master
```

4. Commit the code and pass the unit test

- MMEditing introduces mypy to do static type checking to increase the robustness of the code. Therefore, we need to add Type Hints to our code and pass the mypy check. If you are not familiar with Type Hints, you can refer to this tutorial.
- The committed code should pass through the unit test

  ```
  # Pass all unit tests
  pytest tests

  # Pass the unit test of runner
  pytest tests/test_runner/test_runner.py
  ```

  If the unit test fails for lack of dependencies, you can install the dependencies referring to the guidance
- If the documents are modified/added, we should check the rendering result referring to guidance

5. Push the code to remote

We could push the local commits to remote after passing through the check of unit test and pre-commit. You can associate the local branch with remote branch by adding `-u` option.

```
git push -u origin {branch_name}
```

This will allow you to use the `git push` command to push code directly next time, without having to specify a branch or the remote repository.

6. Create a Pull Request

(1) Create a pull request in GitHub's Pull request interface

(2) Modify the PR description according to the guidelines so that other developers can better understand your changes

Find more details about Pull Request description in *pull request guidelines*.

note

(a) The Pull Request description should contain the reason for the change, the content of the change, and the impact of the change, and be associated with the relevant Issue (see documentation

(b) If it is your first contribution, please sign the CLA

(c) Check whether the Pull Request pass through the CI

MMEditing will run unit test for the posted Pull Request on different platforms (Linux, Window, Mac), based on different versions of Python, PyTorch, CUDA to make sure the code is correct. We can see the specific test information by clicking `Details` in the above image so that we can modify the code.

(3) If the Pull Request passes the CI, then you can wait for the review from other developers. You'll modify the code based on the reviewer's comments, and repeat the steps 4-5 until all reviewers approve it. Then, we will merge it ASAP.

7. Resolve conflicts

If your local branch conflicts with the latest master branch of "upstream", you'll need to resolove them. There are two ways to do this:

```
git fetch --all --prune
git rebase upstream/master
```

or

```
git fetch --all --prune
git merge upstream/master
```

If you are very good at handling conflicts, then you can use rebase to resolve conflicts, as this will keep your commit logs tidy. If you are not familiar with `rebase`, then you can use `merge` to resolve conflicts.

1.1.2 Guidance

Unit test

We should make sure the committed code will not decrease the coverage of unit test, we could run the following command to check the coverage of unit test:

```
python -m coverage run -m pytest /path/to/test_file
python -m coverage html
# check file in htmlcov/index.html
```

Document rendering

If the documents are modified/added, we should check the rendering result. We could install the dependencies and run the following command to render the documents and check the results:

```
pip install -r requirements/docs.txt
cd docs/zh_cn/
# or docs/en
make html
# check file in ./docs/zh_cn/_build/html/index.html
```

1.1.3 Code style

Python

We adopt PEP8 as the preferred code style.

We use the following tools for linting and formatting:

- flake8: A wrapper around some linter tools.
- isort: A Python utility to sort imports.
- yapf: A formatter for Python files.
- codespell: A Python utility to fix common misspellings in text files.
- mdformat: Mdformat is an opinionated Markdown formatter that can be used to enforce a consistent style in Markdown files.
- docformatter: A formatter to format docstring.

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

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

C++ and CUDA

We follow the Google C++ Style Guide.

1.1.4 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

1.2 MMEditing projects

Welcome to the MMEditing community! The MMEditing ecosystem consists of tutorials, libraries, and projects from a broad set of researchers in academia and industry, ML and application engineers. The goal of this ecosystem is to support, accelerate, and aid in your exploration with MMEditing for image, video, 3D content generation, editing and processing.

Here are a few projects that are built upon MMEditing. They are examples of how to use MMEditing as a library, to make your projects more maintainable. Please find more projects in MMEditing Ecosystem.

1.2.1 Show your projects on OpenMMLab Ecosystem

You can submit your project so that it can be shown on the homepage of OpenMMLab.

1.2.2 Add example projects to MMEditing

Here is an example project about how to add your projects to MMEditing. You can copy and create your own project from the example project.

We also provide some documentation listed below for your reference:

- Contribution Guide

  The guides for new contributors about how to add your projects to MMEditing.

- New Model Guide

  The documentation of adding new models.

- Discussions

  Welcome to start a discussion!

1.2.3 Projects of libraries and toolboxes

- PowerVQE: Open framework for quality enhancement of compressed videos based on PyTorch and MMEditing.
- VR-Baseline: Video Restoration Toolbox.
- Derain-Toolbox: Single Image Deraining Toolbox and Benchmark

1.2.4 Projects of research papers

- Towards Interpretable Video Super-Resolution via Alternating Optimization, ECCV 2022[github]
- SepLUT:Separable Image-adaptive Lookup Tables for Real-time Image Enhancement, ECCV 2022[github]
- TTVSR: Learning Trajectory-Aware Transformer for Video Super-Resolution, CVPR 2022[github]
- Arbitrary-Scale Image Synthesis, CVPR 2022[github]
- Investigating Tradeoffs in Real-World Video Super-Resolution(RealBasicVSR), CVPR 2022[github]
- BasicVSR++: Improving Video Super-Resolution with Enhanced Propagation and Alignment, CVPR 2022[github]
- Multi-Scale Memory-Based Video Deblurring, CVPR 2022[github]
- AdaInt:Learning Adaptive Intervals for 3D Lookup Tables on Real-time Image Enhancement, CVPR 2022[github]
- A New Dataset and Transformer for Stereoscopic Video Super-Resolution, CVPRW 2022[github]
- Liquid warping GAN with attention: A unified framework for human image synthesis, TPAMI 2021[github]
- BasicVSR:The Search for Essential Components in Video Super-Resolution and Beyond, CVPR 2021[github]
- GLEAN:Generative Latent Bank for Large-Factor Image Super-Resolution, CVPR 2021[github]
- DAN:Unfolding the Alternating Optimization for Blind Super Resolution, NeurIPS 2020[github]

1.3 Overview

Welcome to MMEditing! In this section, you will know about

- *What is MMEditing?*
- *Why should I use MMEditing?*
- *Get started*
- *User guides*
- *Advanced guides*

1.3.1 What is MMEditing?

MMEditing is an open-source toolbox for professional AI researchers and machine learning engineers to explore image and video processing, editing and synthesis.

MMEditing allows researchers and engineers to use pre-trained state-of-the-art models, train and develop new customized models easily.

MMEditing supports various foundamental generative models, including:

- Unconditional Generative Adversarial Networks (GANs)
- Conditional Generative Adversarial Networks (GANs)
- Internal Learning
- Diffusion Models
- And many other generative models are coming soon!

MMEditing supports various applications, including:

- Image super-resolution
- Video super-resolution
- Video frame interpolation
- Image inpainting
- Image matting
- Image-to-image translation
- And many other applications are coming soon!

1.3.2 Why should I use MMEditing?

- **State of the Art**

  MMEditing provides state-of-the-art generative models to process, edit and synthesize images and videos.

- **Powerful and Popular Applications**

  MMEditing supports popular and contemporary *inpainting*, *matting*, *super-resolution* and *generation* applications. Specifically, MMEditing supports GAN interpolation, GAN projection, GAN manipulations and many other popular GAN's applications. It's time to play with your GANs!

- **New Modular Design for Flexible Combination:**

  We decompose the editing framework into different modules and one can easily construct a customized editor framework by combining different modules. Specifically, a new design for complex loss modules is proposed for customizing the links between modules, which can achieve flexible combinations among different modules.(Tutorial for *losses*)

- **Efficient Distributed Training:**

  With the support of MMSeparateDistributedDataParallel, distributed training for dynamic architectures can be easily implemented.

1.3.3 Get started

For installation instructions, please see *Installation*.

1.3.4 User guides

For beginners, we suggest learning the basic usage of MMEditing from *user_guides*.

Advanced guides

For users who are familiar with MMEditing, you may want to learn the design of MMEditing, as well as how to extend the repo, how to use multiple repos and other advanced usages, please refer to *advanced_guides*.

How to

For users who want to use MMEditing to do something, please refer to *How to*.

1.4 Installation

In this section, you will know about:

- *Installation*
  - *Prerequisites*
  - *Best practices*
  - *Customize installation*
  - *Developing with multiple MMEditing versions*

1.4.1 Installation

We recommend that users follow our *Best practices* to install MMEditing 1.x. However, the whole process is highly customizable. See *Customize installation* section for more information.

Prerequisites

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

MMEditing works on Linux, Windows, and macOS. It requires:

- Python >= 3.6
- PyTorch >= 1.5
- MMCV >= 2.0.0rc1

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

Step 0. Download and install Miniconda from official website.

Step 1. Create a conda environment and activate it

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

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

- On GPU platforms:

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

- On CPU platforms:

```
conda install pytorch=1.10 torchvision cpuonly -c pytorch
```

Best practices

Step 0. Install MMCV using MIM.

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

Step 1. Install MMEngine.

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

Step 2. Install MMEditing 1.x . Install MMEditing from the source code.

```
git clone -b 1.x https://github.com/open-mmlab/mmediting.git
cd mmediting
pip3 install -e . -v
```

Step 5. Verification.

```
cd ~
python -c "import mmedit; print(mmedit.__version__)"
# Example output: 1.0.0rc1
```

The installation is successful if the version number is output correctly.

Note: You may be curious about what `-e .` means when supplied with `pip install`. Here is the description:

- `-e` means editable mode. When `import mmedit`, modules under the cloned directory are imported. If `pip install` without `-e`, pip will copy cloned codes to somewhere like `lib/python/site-package`. Consequently, modified code under the cloned directory takes no effect unless `pip install` again. Thus, `pip install` with `-e` is particularly convenient for developers. If some codes are modified, new codes will be imported next time without reinstallation.
- `.` means code in this directory

You can also use `pip install -e .[all]`, which will install more dependencies, especially for pre-commit hooks and unittests.

Customize installation

CUDA Version

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

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

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

note Installing CUDA runtime libraries is enough if you follow our best practices, because no CUDA code will be compiled locally. However, if you hope to compile MMCV from source or develop other CUDA operators, you need to install the complete CUDA toolkit from NVIDIA's website, and its version should match the CUDA version of PyTorch. i.e., the specified version of cudatoolkit in `conda install` command.

Install MMCV without MIM

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

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

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

```
pip install 'mmcv>=2.0.0rc1' -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.10/
↪index.html
```

Using MMEditing with Docker

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

```
# build an image with PyTorch 1.8, CUDA 11.1
# If you prefer other versions, just modified the Dockerfile
docker build -t mmediting docker/
```

Run it with

```
docker run --gpus all --shm-size=8g -it -v {DATA_DIR}:/mmediting/data mmediting
```

Trouble shooting

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

Developing with multiple MMEditing versions

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

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

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

1.5 Quick run

After installing MMEditing successfully, now you are able to play with MMEditing!

To synthesize an image of a church, you only need several lines of codes by MMEditing!

```
from mmedit.apis import init_model, sample_unconditional_model

config_file = 'configs/styleganv2/stylegan2_c2_8xb4-800kiters_lsun-church-256x256.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmediting/stylegan2/official_weights/
↪stylegan2-church-config-f-official_20210327_172657-1d42b7d1.pth'
device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images
fake_imgs = sample_unconditional_model(model, 4)
```

Or you can just run the following command.

```
python demo/unconditional_demo.py \
configs/styleganv2/stylegan2_c2_lsun-church_256_b4x8_800k.py \
https://download.openmmlab.com/mmediting/stylegan2/official_weights/stylegan2-church-
↪config-f-official_20210327_172657-1d42b7d1.pth
```

You will see a new image `unconditional_samples.png` in folder `work_dirs/demos/`, which contained generated samples.

What's more, if you want to make these photos much more clear, you only need several lines of codes for image super-resolution by MMEditing!

```
import mmcv
from mmedit.apis import init_model, restoration_inference
from mmedit.utils import tensor2img

config = 'configs/esrgan/esrgan_x4c64b23g32_1xb16-400k_div2k.py'
checkpoint = 'https://download.openmmlab.com/mmediting/restorers/esrgan/esrgan_
↪x4c64b23g32_1x16_400k_div2k_20200508-f8ccaf3b.pth'
img_path = 'tests/data/image/lq/baboon_x4.png'
model = init_model(config, checkpoint)
output = restoration_inference(model, img_path)
output = tensor2img(output)
mmcv.imwrite(output, 'output.png')
```

Now, you can check your fancy photos in `output.png`.

1.6 Tutorial 1: Learn about Configs in MMEditing

We incorporate modular and inheritance design into our config system, which is convenient to conduct various experiments. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config.

You can learn about the usage of our config system according to the following tutorials.

- *Modify config*
- *Config file structure*
- *Config name style*
- *An example of EDSR*
- *An example of StyleGAN2*
- *Other examples*
  - *An example of config system for inpainting*
  - *An example of config system for matting*
  - *An example of config system for restoration*

1.6.1 Modify config through script arguments

When submitting jobs using `tools/train.py` or `tools/test.py`, you may specify `--cfg-options` to in-place modify the config.

- Update config keys of dict chains.

  The config options can be specified following the order of the dict keys in the original config. For example, `--cfg-options test_cfg.use_ema=False` changes the default sampling model to the original generator, and `--cfg-options train_dataloader.batch_size=8` changes the batch size of train dataloader.

- Update keys inside a list of configs.

  Some config dicts are composed as a list in your config. For example, the training pipeline `train_dataloader.dataset.pipeline` is normally a list e.g. `[dict(type='LoadImageFromFile'), ...]`. If you want to change `'LoadImageFromFile'` to `'LoadImageFromWebcam'` in the pipeline, you may specify `--cfg-options train_dataloader.dataset.pipeline.0.type=LoadImageFromWebcam`. The training pipeline `train_pipeline` is normally a list e.g. `[dict(type='LoadImageFromFile'), ...]`. If you want to change `'LoadImageFromFile'` to `'LoadMask'` in the pipeline, you may specify `--cfg-options train_pipeline.0.type=LoadMask`.

- Update values of list/tuples.

  If the value to be updated is a list or a tuple. You can set `--cfg-options key="[a,b]"` or `--cfg-options key=a,b`. It also allows nested list/tuple values, e.g., `--cfg-options key="[(a,b),(c,d)]"`. Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks in the specified value.

1.6.2 Config file structure

There are 3 basic component types under `config/_base_`: datasets, models and default_runtime. Many methods could be easily constructed with one of each like AOT-GAN, EDVR, GLEAN, StyleGAN2, CycleGAN, SinGAN, etc. Configs consisting of components from `_base_` are called *primitive*.

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

For easy understanding, we recommend contributors to inherit from existing methods. For example, if some modification is made base on BasicVSR, user may first inherit the basic BasicVSR structure by specifying `_base_ = ../basicvsr/basicvsr_reds4.py`, then modify the necessary fields in the config files. If some modification is made base on StyleGAN2, user may first inherit the basic StyleGAN2 structure by specifying `_base_ = ../styleganv2/stylegan2_c2_ffhq_256_b4x8_800k.py`, then modify the necessary fields in the config files.

If you are building an entirely new method that does not share the structure with any of the existing methods, you may create a folder `xxx` under `configs`,

Please refer to MMEngine for detailed documentation.

1.6.3 Config name style

```
{model}_[module setting]_{training schedule}_{dataset}
```

`{xxx}` is required field and `[yyy]` is optional.

- `{model}`: model type like `stylegan`, `dcgan`, `basicvsr`, `dim`, etc. Settings referred in the original paper are included in this field as well (e.g., `Stylegan2-config-f`, `edvrm` of `edvrm_8xb4-600k_reds`.)
- `[module setting]`: specific setting for some modules, including Encoder, Decoder, Generator, Discriminator, Normalization, loss, Activation, etc. E.g. `c64n7` of `basicvsr-pp_c64n7_8xb1-600k_reds4`, learning rate `Glr4e-4_Dlr1e-4` for dcgan, `gamma32.8` for stylegan3, `woReLUInplace` in sagan. In this section, information from different submodules (e.g., generator and discriminator) are connected with `_`.
- `{training_scheduler}`: specific setting for training, including batch_size, schedule, etc. For example, learning rate (e.g., `lr1e-3`), number of gpu and batch size is used (e.g., `8xb32`), and total iterations (e.g., `160kiter`) or number of images shown in the discriminator (e.g., `12Mimgs`).
- `{dataset}`: dataset name and data size info like `celeba-256x256` of `deepfillv1_4xb4_celeba-256x256`, `reds4` of `basicvsr_2xb4_reds4`, `ffhq`, `lsun-car`, `celeba-hq`.

1.6.4 An example of EDSR

To help the users have a basic idea of a complete config, we make a brief comments on the config of the EDSR model we implemented as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation and the tutorial in MMEngine.

Model config

In MMEditing's config, we use model fields to set up a model.

```
model = dict(
    type='BaseEditModel',  # Name of the model
    generator=dict(  # Config of the generator
        type='EDSRNet',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')  # Config for␣
↪pixel loss
    train_cfg=dict(),  # Config of training model.
    test_cfg=dict(),  # Config of testing model.
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='EditDataPreprocessor', mean=[0., 0., 0.], std=[255., 255.,
                                                             255.]))
```

Data config

Dataloaders are required for the training, validation, and testing of the runner. Dataset and data pipeline need to be set to build the dataloader. Due to the complexity of this part, we use intermediate variables to simplify the writing of dataloader configs.

Data pipeline

```
train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find the corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find the corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='SetValues', dictionary=dict(scale=scale)),  # Set value to destination␣
↪keys
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
```

(continues on next page)

(continued from previous page)

```
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]
test_pipeline = [  # Test pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]
```

Dataloader

```
dataset_type = 'BasicImageDataset'  # The type of dataset
data_root = 'data'  # Root path of data
train_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type,  # Type of dataset
        ann_file='meta_info_DIV2K800sub_GT.txt',  # Path of annotation file
        metainfo=dict(dataset_type='div2k', task_name='sisr'),
        data_root=data_root + '/DIV2K',  # Root path of data
        data_prefix=dict(  # Prefix of image path
            img='DIV2K_train_LR_bicubic/X2_sub', gt='DIV2K_train_HR_sub'),
        filename_tmpl=dict(img='{}', gt='{}'),  # Filename template
        pipeline=train_pipeline))
val_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        metainfo=dict(dataset_type='set5', task_name='sisr'),
```

(continues on next page)

(continued from previous page)

```
        data_root=data_root + '/Set5',  # Root path of data
        data_prefix=dict(img='LRbicx2', gt='GTmod12'),  # Prefix of image path
        pipeline=test_pipeline))
test_dataloader = val_dataloader
```

Evaluation config

Evaluators are used to compute the metrics of the trained model on the validation and testing datasets. The config of evaluators consists of one or a list of metric configs:

```
val_evaluator = [
    dict(type='MAE'),  # The name of metrics to evaluate
    dict(type='PSNR', crop_border=scale),  # The name of metrics to evaluate
    dict(type='SSIM', crop_border=scale),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator # The config for testing evaluator
```

Training and testing config

MMEngine's runner uses Loop to control the training, validation, and testing processes. Users can set the maximum training iteration and validation intervals with these fields.

```
train_cfg = dict(
    type='IterBasedTrainLoop',  # The name of train loop type
    max_iters=300000,  # The number of total iterations
    val_interval=5000,  # The number of validation interval iterations
)
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type
```

Optimization config

`optim_wrapper` is the field to configure optimization related settings. The optimizer wrapper not only provides the functions of the optimizer, but also supports functions such as gradient clipping, mixed precision training, etc. Find more in optimizer wrapper tutorial.

```
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=0.00001),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.
```

`param_scheduler` is a field that configures methods of adjusting optimization hyper-parameters such as learning rate and momentum. Users can combine multiple schedulers to create a desired parameter adjustment strategy. Find more in parameter scheduler tutorial.

```
param_scheduler = dict(  # Config of learning policy
    type='MultiStepLR', by_epoch=False, milestones=[200000], gamma=0.5)
```

Hook config

Users can attach hooks to training, validation, and testing loops to insert some operations during running. There are two different hook fields, one is `default_hooks` and the other is `custom_hooks`.

`default_hooks` is a dict of hook configs. `default_hooks` are the hooks must required at runtime. They have default priority which should not be modified. If not set, runner will use the default values. To disable a default hook, users can set its config to `None`.

```
default_hooks = dict(  # Used to build default hooks
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=5000,  # The save interval is 5000 iterations
        save_optimizer=True,
        by_epoch=False,  # Count by iterations
        out_dir=save_dir,
    ),
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)
```

`custom_hooks` is a list of hook configs. Users can develop there own hooks and insert them in this field.

```
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)] # Config of␣
↪visualization hook
```

Runtime config

```
default_scope = 'mmedit' # Used to set registries location
env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
)
log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', window_size=100, by_epoch=False)  # Used to␣
↪build log processor
load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

1.6.5 An example of StyleGAN2

Taking Stylegan2 at 1024x1024 scale as an example, we introduce each field in the config according to different function modules.

Model config

In addition to neural network components such as generator, discriminator etc, it also requires `data_preprocessor`, `loss_config`, and some of them contains `ema_config`. `data_preprocessor` is responsible for processing a batch of data output by dataloader. `loss_config` is responsible for weight of loss terms. `ema_config` is responsible for exponential moving average (EMA) operation for generator.

```
model = dict(
    type='StyleGAN2',  # The name of the model
    data_preprocessor=dict(type='GANDataPreprocessor'),  # The config of data␣
↪preprocessor, usually includs image normalization and padding
    generator=dict(  # The config for generator
        type='StyleGANv2Generator',  # The name of the generator
        out_size=1024,  # The output resolution of the generator
        style_channels=512),  # The number of style channels of the generator
    discriminator=dict(  # The config for discriminator
        type='StyleGAN2Discriminator',  # The name of the discriminator
        in_size=1024),  # The input resolution of the discriminator
    ema_config=dict(  # The config for EMA
        type='ExponentialMovingAverage',  # Specific the type of Average model
        interval=1,  # The interval of EMA operation
        momentum=0.9977843871238888),  # The momentum of EMA operation
    loss_config=dict(  # The config for loss terms
        r1_loss_weight=80.0,  # The weight for r1 gradient penalty
        r1_interval=16,  # The interval of r1 gradient penalty
        norm_mode='HWC',  # The normalization mode for r1 gradient penalty
        g_reg_interval=4,  # The interval for generator's regularization
        g_reg_weight=8.0,  # The weight for generator's regularization
        pl_batch_shrink=2))  # The factor of shrinking the batch size in path length␣
↪regularization
```

Dataset and evaluator config

Dataloaders are required for the training, validation, and testing of the runner. Dataset and data pipeline need to be set to build the dataloader. Due to the complexity of this part, we use intermediate variables to simplify the writing of dataloader configs.

```
dataset_type = 'BasicImageDataset'  # Dataset type, this will be used to define the␣
↪dataset
data_root = './data/ffhq/'  # Root path of data

train_pipeline = [  # Training data process pipeline
    dict(type='LoadImageFromFile', key='img'),  # First pipeline to load images from␣
↪file path
    dict(type='Flip', keys=['img'], direction='horizontal'),  # Argumentation pipeline␣
↪that flip the images
    dict(type='PackEditInputs', keys=['img'])  # The last pipeline that formats the␣
↪annotation data (if have) and decides which keys in the data should be packed into␣
↪data_samples
```

(continues on next page)

(continued from previous page)

```
]
val_pipeline = [
    dict(type='LoadImageFromFile', key='img'),  # First pipeline to load images from␣
↪file path
    dict(type='PackEditInputs', keys=['img'])  # The last pipeline that formats the␣
↪annotation data (if have) and decides which keys in the data should be packed into␣
↪data_samples
]
train_dataloader = dict(  # The config of train dataloader
    batch_size=4,  # Batch size of a single GPU
    num_workers=8,  # Worker to pre-fetch data for each single GPU
    persistent_workers=True,  # If ``True``, the dataloader will not shutdown the worker␣
↪processes after an epoch end, which can accelerate training speed.
    sampler=dict(  # The config of training data sampler
        type='InfiniteSampler',  # InfiniteSampler for iteratiion-based training. Refers␣
↪to https://github.com/open-mmlab/mmengine/blob/
↪fe0eb0a5bbc8bf816d5649bfdd34908c258eb245/mmengine/dataset/sampler.py#L107
        shuffle=True),  # Whether randomly shuffle the training data
    dataset=dict(  # The config of the training dataset
        type=dataset_type,
        data_root=data_root,
        pipeline=train_pipeline))
val_dataloader = dict(  # The config of validation dataloader
    batch_size=4,  # Batch size of a single GPU
    num_workers=8,  # Worker to pre-fetch data for each single GPU
    dataset=dict(  # The config of the validation dataset
        type=dataset_type,
        data_root=data_root,
        pipeline=val_pipeline),
    sampler=dict(  # The config of validatioin data sampler
        type='DefaultSampler',  # DefaultSampler which supports both distributed and non-
↪distributed training. Refer to https://github.com/open-mmlab/mmengine/blob/
↪fe0eb0a5bbc8bf816d5649bfdd34908c258eb245/mmengine/dataset/sampler.py#L14
        shuffle=False),  # Whether randomly shuffle the validation data
    persistent_workers=True)
test_dataloader = val_dataloader  # The config of the testing dataloader
```

Evaluators are used to compute the metrics of the trained model on the validation and testing datasets. The config of evaluators consists of one or a list of metric configs:

```
val_evaluator = dict(  # The config for validation evaluator
    type='GenEvaluator',  # The type of evaluation
    metrics=[  # The config for metrics
        dict(
            type='FrechetInceptionDistance',
            prefix='FID-Full-50k',
            fake_nums=50000,
            inception_style='StyleGAN',
            sample_model='ema'),
        dict(type='PrecisionAndRecall', fake_nums=50000, prefix='PR-50K'),
        dict(type='PerceptualPathLength', fake_nums=50000, prefix='ppl-w')
    ])
```

(continues on next page)

(continued from previous page)

```
test_evaluator = val_evaluator  # The config for testing evaluator
```

Training and testing config

MMEngine's runner uses Loop to control the training, validation, and testing processes. Users can set the maximum training iteration and validation intervals with these fields.

```
train_cfg = dict(  # The config for training
    by_epoch=False,  # Set `by_epoch` as False to use iteration-based training
    val_begin=1,  # Which iteration to start the validation
    val_interval=10000,  # Validation intervals
    max_iters=800002)  # Maximum training iterations
val_cfg = dict(type='GenValLoop')  # The validation loop type
test_cfg = dict(type='GenTestLoop')  # The testing loop type
```

Optimization config

`optim_wrapper` is the field to configure optimization related settings. The optimizer wrapper not only provides the functions of the optimizer, but also supports functions such as gradient clipping, mixed precision training, etc. Find more in optimizer wrapper tutorial.

```
optim_wrapper = dict(
    constructor='GenOptimWrapperConstructor',
    generator=dict(
        optimizer=dict(type='Adam', lr=0.0016, betas=(0, 0.9919919678228657))),
    discriminator=dict(
        optimizer=dict(
            type='Adam',
            lr=0.0018823529411764706,
            betas=(0, 0.9905854573074332))))
```

`param_scheduler` is a field that configures methods of adjusting optimization hyperparameters such as learning rate and momentum. Users can combine multiple schedulers to create a desired parameter adjustment strategy. Find more in parameter scheduler tutorial. Since StyleGAN2 do not use parameter scheduler, we use config in CycleGAN as an example:

```
# parameter scheduler in CycleGAN config
param_scheduler = dict(
    type='LinearLrInterval',  # The type of scheduler
    interval=400,  # The interval to update the learning rate
    by_epoch=False,  # The scheduler is called by iteration
    start_factor=0.0002,  # The number we multiply parameter value in the first iteration
    end_factor=0,  # The number we multiply parameter value at the end of linear␣
↪changing process.
    begin=40000,  # The start iteration of the scheduler
    end=80000)  # The end iteration of the scheduler
```

Hook config

Users can attach hooks to training, validation, and testing loops to insert some operations during running. There are two different hook fields, one is `default_hooks` and the other is `custom_hooks`.

`default_hooks` is a dict of hook configs. `default_hooks` are the hooks must required at runtime. They have default priority which should not be modified. If not set, runner will use the default values. To disable a default hook, users can set its config to `None`.

```
default_hooks = dict(
    timer=dict(type='GenIterTimerHook'),
    logger=dict(type='LoggerHook', interval=100, log_metric_by_epoch=False),
    checkpoint=dict(
        type='CheckpointHook',
        interval=10000,
        by_epoch=False,
        less_keys=['FID-Full-50k/fid'],
        greater_keys=['IS-50k/is'],
        save_optimizer=True,
        save_best='FID-Full-50k/fid'))
```

`custom_hooks` is a list of hook configs. Users can develop there own hooks and insert them in this field.

```
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        vis_kwargs_list=dict(type='GAN', name='fake_img'))
]
```

Runtime config

```
default_scope = 'mmedit'  # The default registry scope to find modules. Refer to https://
↪mmengine.readthedocs.io/en/latest/tutorials/registry.html

# config for environment
env_cfg = dict(
    cudnn_benchmark=True,  # whether to enable cudnn benchmark.
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),  # set multi process␣
↪parameters.
    dist_cfg=dict(backend='nccl'),  # set distributed parameters.
)

log_level = 'INFO'  # The level of logging
log_processor = dict(
    type='GenLogProcessor',  # log processor to process runtime logs
    by_epoch=False)  # print log by iteration
load_from = None  # load model checkpoint as a pre-trained model for a given path
resume = False  # Whether to resume from the checkpoint define in `load_from`. If `load_
↪from` is `None`, it will resume the latest checkpoint in `work_dir`
```

1.6.6 Other examples

An example of config system for inpainting

To help the users have a basic idea of a complete config and the modules in a inpainting system, we make brief comments on the config of Global&Local as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation.

```
model = dict(
    type='GLInpaintor', # The name of inpaintor
    data_preprocessor=dict(
        type='EditDataPreprocessor', # The name of data preprocessor
        mean=[127.5], # Mean value used in data normalization
        std=[127.5], # Std value used in data normalization
    ),
    encdec=dict(
        type='GLEncoderDecoder', # The name of encoder-decoder
        encoder=dict(type='GLEncoder', norm_cfg=dict(type='SyncBN')), # The config of␣
↪encoder
        decoder=dict(type='GLDecoder', norm_cfg=dict(type='SyncBN')), # The config of␣
↪decoder
        dilation_neck=dict(
            type='GLDilationNeck', norm_cfg=dict(type='SyncBN'))), # The config of␣
↪dilation neck
    disc=dict(
        type='GLDiscs', # The name of discriminator
        global_disc_cfg=dict(
            in_channels=3, # The input channel of discriminator
            max_channels=512, # The maximum middle channel in discriminator
            fc_in_channels=512 * 4 * 4, # The input channel of last fc layer
            fc_out_channels=1024, # The output channel of last fc channel
            num_convs=6, # The number of convs used in discriminator
            norm_cfg=dict(type='SyncBN') # The config of norm layer
        ),
        local_disc_cfg=dict(
            in_channels=3, # The input channel of discriminator
            max_channels=512, # The maximum middle channel in discriminator
            fc_in_channels=512 * 4 * 4, # The input channel of last fc layer
            fc_out_channels=1024, # The output channel of last fc channel
            num_convs=5, # The number of convs used in discriminator
            norm_cfg=dict(type='SyncBN') # The config of norm layer
        ),
    ),
    loss_gan=dict(
        type='GANLoss', # The name of GAN loss
        gan_type='vanilla', # The type of GAN loss
        loss_weight=0.001 # The weight of GAN loss
    ),
    loss_l1_hole=dict(
        type='L1Loss', # The type of l1 loss
        loss_weight=1.0 # The weight of l1 loss
    ))
```

(continues on next page)

(continued from previous page)

```
train_cfg = dict(
    type='IterBasedTrainLoop',# The name of train loop type
    max_iters=500002, # The number of total iterations
    val_interval=50000, # The number of validation interval iterations
)
val_cfg = dict(type='ValLoop') # The name of validation loop type
test_cfg = dict(type='TestLoop') # The name of test loop type

val_evaluator = [
    dict(type='MAE', mask_key='mask', scaling=100), # The name of metrics to evaluate
    dict(type='PSNR'), # The name of metrics to evaluate
    dict(type='SSIM'), # The name of metrics to evaluate
]
test_evaluator = val_evaluator

input_shape = (256, 256) # The shape of input image

train_pipeline = [
    dict(type='LoadImageFromFile', key='gt'), # The config of loading image
    dict(
        type='LoadMask', # The type of loading mask pipeline
        mask_mode='bbox', # The type of mask
        mask_config=dict(
            max_bbox_shape=(128, 128), # The shape of bbox
            max_bbox_delta=40, # The changing delta of bbox height and width
            min_margin=20,  # The minimum margin from bbox to the image border
            img_shape=input_shape)),  # The input image shape
    dict(
        type='Crop', # The type of crop pipeline
        keys=['gt'],  # The keys of images to be cropped
        crop_size=(384, 384),  # The size of cropped patch
        random_crop=True,  # Whether to use random crop
    ),
    dict(
        type='Resize',  # The type of resizing pipeline
        keys=['gt'],  # They keys of images to be resized
        scale=input_shape,  # The scale of resizing function
        keep_ratio=False,  # Whether to keep ratio during resizing
    ),
    dict(
        type='Normalize',  # The type of normalizing pipeline
        keys=['gt_img'],  # The keys of images to be normed
        mean=[127.5] * 3,  # Mean value used in normalization
        std=[127.5] * 3,  # Std value used in normalization
        to_rgb=False),  # Whether to transfer image channels to rgb
    dict(type='GetMaskedImage'), # The config of getting masked image pipeline
    dict(type='PackEditInputs'), # The config of collecting data from the current␣
↪pipeline
]

test_pipeline = train_pipeline  # Constructing testing/validation pipeline
```

(continues on next page)

(continued from previous page)

```
dataset_type = 'BasicImageDataset' # The type of dataset
data_root = 'data/places'  # Root path of data

train_dataloader = dict(
    batch_size=12, # Batch size of a single GPU
    num_workers=4, # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False, # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=False), # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type, # Type of dataset
        data_root=data_root, # Root path of data
        data_prefix=dict(gt='data_large'), # Prefix of image path
        ann_file='meta/places365_train_challenge.txt', # Path of annotation file
        test_mode=False,
        pipeline=train_pipeline,
    ))

val_dataloader = dict(
    batch_size=1, # Batch size of a single GPU
    num_workers=4, # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False, # Whether maintain the workers Dataset instances alive
    drop_last=False, # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False), # The type of data sampler
    dataset=dict( # Validation dataset config
        type=dataset_type, # Type of dataset
        data_root=data_root, # Root path of data
        data_prefix=dict(gt='val_large'), # Prefix of image path
        ann_file='meta/places365_val.txt', # Path of annotation file
        test_mode=True,
        pipeline=test_pipeline,
    ))

test_dataloader = val_dataloader

model_wrapper_cfg = dict(type='MMSeparateDistributedDataParallel') # The name of model␣
↪wrapper

optim_wrapper = dict( # Config used to build optimizer, support all the optimizers in␣
↪PyTorch whose arguments are also the same as those in PyTorch
    constructor='MultiOptimWrapperConstructor',
    generator=dict(
        type='OptimWrapper', optimizer=dict(type='Adam', lr=0.0004)),
    disc=dict(type='OptimWrapper', optimizer=dict(type='Adam', lr=0.0004)))

default_scope = 'mmedit' # Used to set registries location
save_dir = './work_dirs' # Directory to save the model checkpoints and logs for the␣
↪current experiments
exp_name = 'gl_places'  # The experiment name

default_hooks = dict( # Used to build default hooks
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100), # Config to register logger hook
```

(continues on next page)

(continued from previous page)

```
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict( # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=50000,
        by_epoch=False,
        out_dir=save_dir),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)

env_cfg = dict( # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),
    dist_cfg=dict(backend='nccl'),
)

vis_backends = [dict(type='LocalVisBackend')] # The name of visualization backend
visualizer = dict( # Config used to build visualizer
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
    bgr2rgb=True)
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)] # Used to build custom␣
↪hooks

log_level = 'INFO' # The level of logging
log_processor = dict(type='LogProcessor', by_epoch=False) # Used to build log processor

load_from = None # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False # Resume checkpoints from a given path, the training will be resumed from␣
↪the epoch when the checkpoint's is saved.

find_unused_parameters = False  # Whether to set find unused parameters in ddp
```

An example of config system for matting

To help the users have a basic idea of a complete config, we make a brief comments on the config of the original DIM model we implemented as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation.

```
# model settings
model = dict(
    type='DIM',  # The name of model (we call mattor).
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='MattorPreprocessor',
        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        bgr_to_rgb=True,
        proc_inputs='normalize',
        proc_trimap='rescale_to_zero_one',
```

(continues on next page)

(continued from previous page)

```
        proc_gt='rescale_to_zero_one',
    ),
    backbone=dict(  # The config of the backbone.
        type='SimpleEncoderDecoder',  # The type of the backbone.
        encoder=dict(  # The config of the encoder.
            type='VGG16'),  # The type of the encoder.
        decoder=dict(  # The config of the decoder.
            type='PlainDecoder')),  # The type of the decoder.
    pretrained='./weights/vgg_state_dict.pth',  # The pretrained weight of the encoder␣
↪to be loaded.
    loss_alpha=dict(  # The config of the alpha loss.
        type='CharbonnierLoss',  # The type of the loss for predicted alpha matte.
        loss_weight=0.5),  # The weight of the alpha loss.
    loss_comp=dict(  # The config of the composition loss.
        type='CharbonnierCompLoss',  # The type of the composition loss.
        loss_weight=0.5), # The weight of the composition loss.
    train_cfg=dict(  # Config of training DIM model.
        train_backbone=True,  # In DIM stage1, backbone is trained.
        train_refiner=False),  # In DIM stage1, refiner is not trained.
    test_cfg=dict(  # Config of testing DIM model.
        refine=False,  # Whether use refiner output as output, in stage1, we don't use it.
        resize_method='pad',
        resize_mode='reflect',
        size_divisor=32,
    ),
)

# data settings
dataset_type = 'AdobeComp1kDataset'  # Dataset type, this will be used to define the␣
↪dataset.
data_root = 'data/adobe_composition-1k'  # Root path of data.

train_pipeline = [  # Training data processing pipeline.
    dict(
        type='LoadImageFromFile',  # Load alpha matte from file.
        key='alpha',  # Key of alpha matte in annotation file. The pipeline will read␣
↪alpha matte from path `alpha_path`.
        color_type='grayscale'),  # Load as grayscale image which has shape (height,␣
↪width).
    dict(
        type='LoadImageFromFile',  # Load image from file.
        key='fg'),  # Key of image to load. The pipeline will read fg from path `fg_path`.
    dict(
        type='LoadImageFromFile',  # Load image from file.
        key='bg'),  # Key of image to load. The pipeline will read bg from path `bg_path`.
    dict(
        type='LoadImageFromFile',  # Load image from file.
        key='merged'),  # Key of image to load. The pipeline will read merged from path␣
↪`merged_path`.
    dict(
        type='CropAroundUnknown',  # Crop images around unknown area (semi-transparent␣
↪area).
```

(continues on next page)

(continued from previous page)

```
        keys=['alpha', 'merged', 'fg', 'bg'],  # Images to crop.
        crop_sizes=[320, 480, 640]),  # Candidate crop size.
    dict(
        type='Flip',  # Augmentation pipeline that flips the images.
        keys=['alpha', 'merged', 'fg', 'bg']),  # Images to be flipped.
    dict(
        type='Resize',  # Augmentation pipeline that resizes the images.
        keys=['alpha', 'merged', 'fg', 'bg'],  # Images to be resized.
        scale=(320, 320),  # Target size.
        keep_ratio=False),  # Whether to keep the ratio between height and width.
    dict(
        type='GenerateTrimap',  # Generate trimap from alpha matte.
        kernel_size=(1, 30)),  # Kernel size range of the erode/dilate kernel.
    dict(type='PackEditInputs'),  # The config of collecting data from the current␣
↪pipeline
]
test_pipeline = [
    dict(
        type='LoadImageFromFile',  # Load alpha matte.
        key='alpha',  # Key of alpha matte in annotation file. The pipeline will read␣
↪alpha matte from path `alpha_path`.
        color_type='grayscale',
        save_original_img=True),
    dict(
        type='LoadImageFromFile',  # Load image from file
        key='trimap',  # Key of image to load. The pipeline will read trimap from path␣
↪`trimap_path`.
        color_type='grayscale',  # Load as grayscale image which has shape (height,␣
↪width).
        save_original_img=True),  # Save a copy of trimap for calculating metrics. It␣
↪will be saved with key `ori_trimap`
    dict(
        type='LoadImageFromFile',  # Load image from file
        key='merged'),  # Key of image to load. The pipeline will read merged from path␣
↪`merged_path`.
    dict(type='PackEditInputs'),  # The config of collecting data from the current␣
↪pipeline
]

train_dataloader = dict(
    batch_size=1,  # Batch size of a single GPU
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type,  # Type of dataset
        data_root=data_root,  # Root path of data
        ann_file='training_list.json',  # Path of annotation file
        test_mode=False,
        pipeline=train_pipeline,
    ))
```

(continues on next page)

(continued from previous page)

```
val_dataloader = dict(
    batch_size=1,  # Batch size of a single GPU
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        data_root=data_root,  # Root path of data
        ann_file='test_list.json',  # Path of annotation file
        test_mode=True,
        pipeline=test_pipeline,
    ))

test_dataloader = val_dataloader

val_evaluator = [
    dict(type='SAD'),  # The name of metrics to evaluate
    dict(type='MattingMSE'),  # The name of metrics to evaluate
    dict(type='GradientError'),  # The name of metrics to evaluate
    dict(type='ConnectivityError'),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator

train_cfg = dict(
    type='IterBasedTrainLoop',  # The name of train loop type
    max_iters=1_000_000,  # The number of total iterations
    val_interval=40000,  # The number of validation interval iterations
)
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type

# optimizer
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=0.00001),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.

default_scope = 'mmedit'  # Used to set registries location
save_dir = './work_dirs'  # Directory to save the model checkpoints and logs for the␣
↪current experiments.

default_hooks = dict(  # Used to build default hooks
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=40000,  # The save interval is 40000 iterations.
```

(continues on next page)

(continued from previous page)

```
        by_epoch=False,  # Count by iterations.
        out_dir=save_dir),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)

env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
)

log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', by_epoch=False)  # Used to build log processor

load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

An example of config system for restoration

To help the users have a basic idea of a complete config, we make a brief comments on the config of the EDSR model we implemented as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation.

```
exp_name = 'edsr_x2c64b16_1x16_300k_div2k'  # The experiment name
work_dir = f'./work_dirs/{experiment_name}'
save_dir = './work_dirs/'

load_from = None  # based on pre-trained x2 model

scale = 2  # Scale factor for upsampling
# model settings
model = dict(
    type='BaseEditModel',  # Name of the model
    generator=dict(  # Config of the generator
        type='EDSRNet',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')  # Config for␣
↪pixel loss
    train_cfg=dict(),  # Config of training model.
    test_cfg=dict(),  # Config of testing model.
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='EditDataPreprocessor', mean=[0., 0., 0.], std=[255., 255.,
```

(continues on next page)

(continued from previous page)

```
                                                    255.]))

train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='SetValues', dictionary=dict(scale=scale)),  # Set value to destination␣
↪keys
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='ToTensor', keys=['img', 'gt']),  # Convert images to tensor
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]
test_pipeline = [  # Test pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='ToTensor', keys=['img', 'gt']),  # Convert images to tensor
    dict(type='PackEditInputs')  # The config of collecting data from the current␣
↪pipeline
]

# dataset settings
dataset_type = 'BasicImageDataset'  # The type of dataset
data_root = 'data'  # Root path of data
```

(continues on next page)

(continued from previous page)

```
train_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type,  # Type of dataset
        ann_file='meta_info_DIV2K800sub_GT.txt',  # Path of annotation file
        metainfo=dict(dataset_type='div2k', task_name='sisr'),
        data_root=data_root + '/DIV2K',  # Root path of data
        data_prefix=dict(  # Prefix of image path
            img='DIV2K_train_LR_bicubic/X2_sub', gt='DIV2K_train_HR_sub'),
        filename_tmpl=dict(img='{}', gt='{}'),  # Filename template
        pipeline=train_pipeline))
val_dataloader = dict(
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        metainfo=dict(dataset_type='set5', task_name='sisr'),
        data_root=data_root + '/Set5',  # Root path of data
        data_prefix=dict(img='LRbicx2', gt='GTmod12'),  # Prefix of image path
        pipeline=test_pipeline))
test_dataloader = val_dataloader

val_evaluator = [
    dict(type='MAE'),  # The name of metrics to evaluate
    dict(type='PSNR', crop_border=scale),  # The name of metrics to evaluate
    dict(type='SSIM', crop_border=scale),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator

train_cfg = dict(
    type='IterBasedTrainLoop', max_iters=300000, val_interval=5000)  # Config of train␣
↪loop type
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type

# optimizer
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=0.00001),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.

param_scheduler = dict(  # Config of learning policy
    type='MultiStepLR', by_epoch=False, milestones=[200000], gamma=0.5)

default_hooks = dict(  # Used to build default hooks
```

(continues on next page)

(continued from previous page)

```
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=5000,  # The save interval is 5000 iterations
        save_optimizer=True,
        by_epoch=False,  # Count by iterations
        out_dir=save_dir,
    ),
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)

default_scope = 'mmedit'  # Used to set registries location
save_dir = './work_dirs'  # Directory to save the model checkpoints and logs for the␣
↪current experiments.

env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
)

log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', window_size=100, by_epoch=False)  # Used to␣
↪build log processor

load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

1.7 Tutorial 2: Prepare datasets

In this section, we will detail how to prepare data and adopt the proper dataset in our repo for different methods.

We support multiple datasets of different tasks. There are two ways to use datasets for training and testing models in MMEditing:

1. Using downloaded datasets directly
2. Preprocessing downloaded datasets before using them.

The structure of this guide is as follows:

1.7.1 Download datasets

You are supposed to download datasets from their homepage first. Most datasets are available after downloaded, so you only need to make sure the folder structure is correct and further preparation is not necessary. For example, you can simply prepare Vimeo90K-triplet datasets by downloading datasets from homepage.

1.7.2 Prepare datasets

Some datasets need to be preprocessed before training or testing. We support many scripts to prepare datasets in tools/dataset_converters. And you can follow the tutorials of every dataset to run scripts. For example, we recommend cropping the DIV2K images to sub-images. We provide a script to prepare cropped DIV2K dataset. You can run the following command:

```
python tools/dataset_converters/super-resolution/div2k/preprocess_div2k_dataset.py --
↪data-root ./data/DIV2K
```

1.7.3 The overview of the datasets in MMEditing

We support detailed tutorials and split them according to different tasks.

Please check our dataset zoo for data preparation of different tasks.

If you're interested in more details of datasets in MMEditing, please check the *advanced guides*.

1.8 Tutorial 3: Inference with pre-trained models

MMEditing provides APIs for you to easily play with state-of-the-art models on your own images or videos. Specifically, MMEditing supports various fundamental generative models, including: unconditional Generative Adversarial Networks (GANs), conditional GANs, internal learning, diffusion models, etc. MMEditing also supports various applications, including: image super-resolution, video super-resolution, video frame interpolation, image inpainting, image matting, image-to-image translation, etc.

In this section, we will specify how to play with our pre-trained models.

- *Tutorial 3: Inference with Pre-trained Models*
  - *Sample images with unconditional GANs*
  - *Sample images with conditional GANs*
  - *Sample images with diffusion models*
  - *Run a demo of image inpainting*
  - *Run a demo of image matting*
  - *Run a demo of image super-resolution*
  - *Run a demo of facial restoration*
  - *Run a demo of video super-resolution*
  - *Run a demo of video frame interpolation*
  - *Run a demo of image translation models*

1.8.1 Sample images with unconditional GANs

MMEditing provides high-level APIs for sampling images with unconditional GANs. Here is an example of building StyleGAN2-256 and obtaining the synthesized images.

```
from mmedit.apis import init_model, sample_unconditional_model

# Specify the path to model config and checkpoint file
config_file = 'configs/styleganv2/stylegan2_c2_8xb4_ffhq-1024x1024.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmediting/stylegan2/stylegan2_c2_ffhq_
↪1024_b4x8_20210407_150045-618c9024.pth'

device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images
fake_imgs = sample_unconditional_model(model, 4)
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/unconditional_demo.py with the following commands:

```
python demo/unconditional_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

Note that more arguments are also offered to customize your sampling procedure. Please use `python demo/unconditional_demo.py --help` to check more details.

1.8.2 Sample images with conditional GANs

MMEditing provides high-level APIs for sampling images with conditional GANs. Here is an example for building SAGAN-128 and obtaining the synthesized images.

```
from mmedit.apis import init_model, sample_conditional_model

# Specify the path to model config and checkpoint file
config_file = 'configs/sagan/sagan_woReLUinplace-Glr1e-4_Dlr4e-4_noaug-ndisc1-8xb32-
↪bigGAN-sch_imagenet1k-128x128.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmediting/sagan/sagan_128_
↪woReLUinplace_noaug_bigGAN_imagenet1k_b32x8_Glr1e-4_Dlr-4e-4_ndisc1_20210818_210232-
↪3f5686af.pth'

device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images with random label
fake_imgs = sample_conditional_model(model, 4)

# sample images with the same label
```

(continues on next page)

(continued from previous page)

```
fake_imgs = sample_conditional_model(model, 4, label=0)

# sample images with specific labels
fake_imgs = sample_conditional_model(model, 4, label=[0, 1, 2, 3])
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/conditional_demo.py with the following commands:

```
python demo/conditional_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    [--label] ${LABEL} \
    [--samples-per-classes] ${SAMPLES_PER_CLASSES} \
    [--sample-all-classes] \
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

If `--label` is not passed, images with random labels would be generated. If `--label` is passed, we would generate `${SAMPLES_PER_CLASSES}` images for each input label. If `sample_all_classes` is set true in command line, `--label` would be ignored and the generator will output images for all categories.

Note that more arguments are also offered to customizing your sampling procedure. Please use `python demo/conditional_demo.py --help` to check more details.

1.8.3 Sample images with diffusion models

MMEditing provides high-level APIs for sampling images with diffusion models. Here is an example for building I-DDPM and obtaining the synthesized images.

```
from mmedit.apis import init_model, sample_ddpm_model

# Specify the path to model config and checkpoint file
config_file = 'configs/improved_ddpm/ddpm_cosine-hybird-timestep-4k_16xb8-1500kiters_
↪imagenet1k-64x64.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmediting/improved_ddpm/ddpm_cosine_
↪hybird_timestep-4k_imagenet1k_64x64_b8x16_1500k_20220103_223919-b8f1a310.pth'
device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# sample images
fake_imgs = sample_ddpm_model(model, 4)
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/ddpm_demo.py with the following commands:

```
python demo/ddpm_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

Note that more arguments are also offered to customizing your sampling procedure. Please use `python demo/ddpm_demo.py --help` to check more details.

1.8.4 Run a demo of image inpainting

You can use the following commands to test images for inpainting.

```
python demo/inpainting_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${MASKED_IMAGE_FILE} \
    ${MASK_FILE} \
    ${SAVE_FILE} \
    [--imshow] \
    [--device ${GPU_ID}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/inpainting_demo.py \
    configs/global_local/gl_256x256_8x12_celeba.py \
    https://download.openmmlab.com/mmediting/inpainting/global_local/gl_256x256_8x12_
↪celeba_20200619-5af0493f.pth \
    tests/data/image/celeba_test.png \
    tests/data/image/bbox_mask.png \
    tests/data/pred/inpainting_celeba.png
```

The predicted inpainting result will be saved in `tests/data/pred/inpainting_celeba.png`.

1.8.5 Run a demo of image matting

You can use the following commands to test a pair of images and trimap.

```
python demo/matting_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${IMAGE_FILE} \
    ${TRIMAP_FILE} \
    ${SAVE_FILE} \
    [--imshow] \
    [--device ${GPU_ID}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/matting_demo.py \
    configs/dim/dim_stage3_v16_pln_1x1_1000k_comp1k.py \
    https://download.openmmlab.com/mmediting/mattors/dim/dim_stage3_v16_pln_1x1_1000k_
↪comp1k_SAD-50.6_20200609_111851-647f24b6.pth \
    tests/data/matting_dataset/merged/GT05.jpg \
    tests/data/matting_dataset/trimap/GT05.png \
    tests/data/pred/GT05.png
```

The predicted alpha matte will be saved in `tests/data/pred/GT05.png`.

1.8.6 Run a demo of image super-resolution

You can use the following commands to test an image for restoration.

```
python demo/restoration_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${IMAGE_FILE} \
    ${SAVE_FILE} \
    [--imshow] \
    [--device ${GPU_ID}] \
    [--ref-path ${REF_PATH}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/restoration_demo.py \
    configs/esrgan/esrgan_x4c64b23g32_g1_400k_div2k.py \
    https://download.openmmlab.com/mmediting/restorers/esrgan/esrgan_x4c64b23g32_1x16_
↪400k_div2k_20200508-f8ccaf3b.pth \
    tests/data/image/lq/baboon_x4.png \
    demo/demo_out_baboon.png
```

You can test Ref-SR by providing `--ref-path`. Examples:

```
python demo/restoration_demo.py \
    configs/ttsr/ttsr-gan_x4_c64b16_g1_500k_CUFED.py \
    https://download.openmmlab.com/mmediting/restorers/ttsr/ttsr-gan_x4_c64b16_g1_500k_
↪CUFED_20210626-2ab28ca0.pth \
    tests/data/frames/sequence/gt/sequence_1/00000000.png \
    demo/demo_out.png \
    --ref-path tests/data/frames/sequence/gt/sequence_1/00000001.png
```

1.8.7 Run a demo of facial restoration

You can use the following commands to test a face image for restoration.

```
python demo/restoration_face_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${IMAGE_FILE} \
    ${SAVE_FILE} \
    [--upscale-factor] \
    [--face-size] \
    [--imshow] \
    [--device ${GPU_ID}]
```

If `--imshow` is specified, the demo will also show image with opencv. Examples:

```
python demo/restoration_face_demo.py \
    configs/glean/glean_in128out1024_4xb2-300k_ffhq-celeba-hq.py \
    https://download.openmmlab.com/mmediting/restorers/glean/glean_in128out1024_4x2_300k_
↪ffhq_celebahq_20210812-acbcb04f.pth \
    tests/data/image/face/000001.png \
```

(continues on next page)

(continued from previous page)

```
    tests/data/pred/000001.png \
    --upscale-factor 4
```

1.8.8 Run a demo of video super-resolution

You can use the following commands to test a video for restoration.

```
python demo/restoration_video_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${INPUT_DIR} \
    ${OUTPUT_DIR} \
    [--window-size=${WINDOW_SIZE}] \
    [--device ${GPU_ID}]
```

It supports both the sliding-window framework and the recurrent framework. Examples:

EDVR:

```
python demo/restoration_video_demo.py \
    configs/edvr/edvrm_wotsa_x4_g8_600k_reds.py \
    https://download.openmmlab.com/mmediting/restorers/edvr/edvrm_wotsa_x4_8x4_600k_reds_
↪20200522-0570e567.pth \
    data/Vid4/BIx4/calendar/ \
    demo/output \
    --window-size=5
```

BasicVSR:

```
python demo/restoration_video_demo.py \
    configs/basicvsr/basicvsr_reds4.py \
    https://download.openmmlab.com/mmediting/restorers/basicvsr/basicvsr_reds4_20120409-
↪0e599677.pth \
    data/Vid4/BIx4/calendar/ \
    demo/output
```

The restored video will be saved in `output/`.

1.8.9 Run a demo of video frame interpolation

You can use the following commands to test a video for frame interpolation.

```
python demo/video_interpolation_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${INPUT_DIR} \
    ${OUTPUT_DIR} \
    [--fps-multiplier ${FPS_MULTIPLIER}] \
    [--fps ${FPS}]
```

`${INPUT_DIR}` / `${OUTPUT_DIR}` can be a path of video file or the folder of a sequence of ordered images. If `${OUTPUT_DIR}` is a path of video file, its frame rate can be determined by the frame rate of input video and `fps_multiplier`, or be determined by `fps` directly (the former has higher priority). Examples:

The frame rate of output video is determined by the frame rate of input video and `fps_multiplier`

```
python demo/video_interpolation_demo.py \
    configs/cain/cain_b5_g1b32_vimeo90k_triplet.py \
    https://download.openmmlab.com/mmediting/video_interpolators/cain/cain_b5_320k_vimeo-
↪triple_20220117-647f3de2.pth \
    tests/data/test_inference.mp4 \
    tests/data/test_inference_vfi_out.mp4 \
    --fps-multiplier 2.0
```

The frame rate of output video is determined by `fps`:

```
python demo/video_interpolation_demo.py \
    configs/cain/cain_b5_g1b32_vimeo90k_triplet.py \
    https://download.openmmlab.com/mmediting/video_interpolators/cain/cain_b5_320k_vimeo-
↪triple_20220117-647f3de2.pth \
    tests/data/test_inference.mp4 \
    tests/data/test_inference_vfi_out.mp4 \
    --fps 60.0
```

1.8.10 Run a demo of image translation models

MMEditing provides high-level APIs for translating images by using image translation models. Here is an example of building Pix2Pix and obtaining the translated images.

```
from mmedit.apis import init_model, sample_img2img_model

# Specify the path to model config and checkpoint file
config_file = 'configs/pix2pix/pix2pix_vanilla-unet-bn_wo-jitter-flip-4xb1-190kiters_
↪edges2shoes.py'
# you can download this checkpoint in advance and use a local file path.
checkpoint_file = 'https://download.openmmlab.com/mmediting/pix2pix/refactor/pix2pix_
↪vanilla_unet_bn_wo_jitter_flip_1x4_186840_edges2shoes_convert-bgr_20210902_170902-
↪0c828552.pth'
# Specify the path to image you want to translate
image_path = 'tests/data/paired/test/33_AB.jpg'
device = 'cuda:0'
# init a generative model
model = init_model(config_file, checkpoint_file, device=device)
# translate a single image
translated_image = sample_img2img_model(model, image_path, target_domain='photo')
```

Indeed, we have already provided a more friendly demo script to users. You can use demo/translation_demo.py with the following commands:

```
python demo/translation_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT} \
    ${IMAGE_PATH}
```

(continues on next page)

(continued from previous page)

```
    [--save-path ${SAVE_PATH}] \
    [--device ${GPU_ID}]
```

Note that more customized arguments are also offered to customize your sampling procedure. Please use `python demo/translation_demo.py --help` to check more details.

1.9 Tutorial 4: Train and test in MMEditing

In this section, we introduce how to test and train models in MMEditing.

In this section, we provide the following guides:

1.9.1 Prerequisite

Users need to *prepare dataset* first to enable training and testing models in MMEditing.

1.9.2 Test a model in MMEditing

Test with a single GPUs

You can use the following commands to test a pre-trained model with single GPUs.

```
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE}
```

For example,

```
python tools/test.py configs/example_config.py work_dirs/example_exp/example_model_
↪20200202.pth
```

Test with multiple GPUs

MMEditing supports testing with multiple GPUs, which can largely save your time in testing models. You can use the following commands to test a pre-trained model with multiple GPUs.

```
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM}
```

For example,

```
./tools/dist_test.sh configs/example_config.py work_dirs/example_exp/example_model_
↪20200202.pth
```

Test with Slurm

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

```
[GPUS=${GPUS}] ./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} $
↪{CHECKPOINT_FILE}
```

Here is an example of using 8 GPUs to test an example model on the 'dev' partition with the job name 'test'.

```
GPUS=8 ./tools/slurm_test.sh dev test configs/example_config.py work_dirs/example_exp/
↪example_model_20200202.pth
```

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

Test with specific metrics

MMEditing provides various **evaluation metrics**, i.e., MS-SSIM, SWD, IS, FID, Precision&Recall, PPL, Equivarience, TransFID, TransIS, etc. We have provided unified evaluation scripts in tools/test.py for all models. If users want to evaluate their models with some metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/styleganv2/stylegan2_c2_ffhq_256_b4x8_800k.py
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
        sample_model='ema'),
    dict(type='PrecisionAndRecall', fake_nums=50000, prefix='PR-50K'),
    dict(type='PerceptualPathLength', fake_nums=50000, prefix='ppl-w')
]
```

As above, `metrics` consist of multiple metric dictionaries. Each metric will contain `type` to indicate the category of the metric. `fake_nums` denotes the number of images generated by the model. Some metrics will output a dictionary of results, you can also set `prefix` to specify the prefix of the results. If you set the prefix of FID as `FID-Full-50k`, then an example of output may be

```
FID-Full-50k/fid: 3.6561  FID-Full-50k/mean: 0.4263  FID-Full-50k/cov: 3.2298
```

Then users can test models with the command below:

```
bash tools/dist_test.sh ${CONFIG_FILE} ${CKPT_FILE}
```

If you are in slurm environment, please switch to the tools/slurm_test.sh by using the following commands:

```
sh slurm_test.sh ${PLATFORM} ${JOBNAME} ${CONFIG_FILE} ${CKPT_FILE}
```

1.9.3 Train a model in MMEditing

MMEditing supports multiple ways of training:

1. *Train with a single GPU*
2. *Train with multiple GPUs*
3. *Train with multiple nodes*
4. *Train with Slurm*

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

Train with a single GPU

```
CUDA_VISIBLE=0 python tools/train.py configs/example_config.py --work-dir work_dirs/
↪example
```

Train with multiple nodes

To launch distributed training on multiple machines, which can be accessed via IPs, run the following commands:

On the first machine:

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

On the second machine:

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

To speed up network communication, high speed network hardware, such as Infiniband, is recommended. Please refer to PyTorch docs for more information.

Train with multiple GPUs

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

Train with Slurm

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

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

Here is an example of using 8 GPUs to train an inpainting model on the dev partition.

```
GPUS=8 ./tools/slurm_train.sh dev configs/inpainting/gl_places.py /nfs/xxxx/gl_places_256
```

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

Optional arguments

- `--amp`: This argument is used for fixed-precision training.
- `--resume`: This argument is used for auto resume if the training is aborted.

1.9.4 Train with specific evaluation metrics

Benefit from the `mmengine`'s `Runner`. We can evaluate model during training in a simple way as below.

```
# define metrics
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN')
]

# define dataloader
val_dataloader = dict(
    batch_size=128,
    num_workers=8,
    dataset=dict(
        type='BasicImageDataset',
        data_root='data/celeba-cropped/',
        pipeline=[
            dict(type='LoadImageFromFile', key='img'),
            dict(type='Resize', scale=(64, 64)),
            dict(type='PackEditInputs')
        ]),
    sampler=dict(type='DefaultSampler', shuffle=False),
    persistent_workers=True)

# define val interval
train_cfg = dict(by_epoch=False, val_begin=1, val_interval=10000)

# define val loop and evaluator
val_cfg = dict(type='GenValLoop')
val_evaluator = dict(type='GenEvaluator', metrics=metrics)
```

You can set `val_begin` and `val_interval` to adjust when to begin validation and interval of validation.

For details of metrics, refer to *metrics' guide*.

1.10 Tutorial 5: Using metrics in MMEditing

MMEditing supports **17 metrics** to assess the quality of models.

Please refer to *Train and Test in MMEditing* for usages.

Here, we will specify the details of different metrics one by one.

The structure of this guide are as follows:

1. *MAE*
2. *MSE*
3. *PSNR*
4. *SNR*
5. *SSIM*
6. *NIQE*
7. *SAD*
8. *MattingMSE*
9. *GradientError*
10. *ConnectivityError*
11. *FID and TransFID*
12. *IS and TransIS*
13. *Precision and Recall*
14. *PPL*
15. *SWD*
16. *MS-SSIM*
17. *Equivarience*

1.10.1 MAE

MAE is Mean Absolute Error metric for image. To evaluate with MAE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='MAE'),
]
```

1.10.2 MSE

MSE is Mean Squared Error metric for image. To evaluate with MSE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='MSE'),
]
```

1.10.3 PSNR

PSNR is Peak Signal-to-Noise Ratio. Our implement refers to https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio. To evaluate with PSNR, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='PSNR'),
]
```

1.10.4 SNR

SNR is Signal-to-Noise Ratio. Our implementation refers to https://en.wikipedia.org/wiki/Signal-to-noise_ratio. To evaluate with SNR, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='SNR'),
]
```

1.10.5 SSIM

SSIM is structural similarity for image, proposed in Image quality assessment: from error visibility to structural similarity. The results of our implementation are the same as that of the official released MATLAB code in https://ece.uwaterloo.ca/~z70wang/research/ssim/. To evaluate with SSIM, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='SSIM'),
]
```

1.10.6 NIQE

NIQE is Natural Image Quality Evaluator metric, proposed in Making a "Completely Blind" Image Quality Analyzer. Our implementation could produce almost the same results as the official MATLAB codes: http://live.ece.utexas.edu/research/quality/niqe_release.zip.

To evaluate with NIQE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='NIQE'),
]
```

1.10.7 SAD

SAD is Sum of Absolute Differences metric for image matting. This metric compute per-pixel absolute difference and sum across all pixels. To evaluate with SAD, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='SAD'),
]
```

1.10.8 MattingMSE

MattingMSE is Mean Squared Error metric for image matting. To evaluate with MattingMSE, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='MattingMSE'),
]
```

1.10.9 GradientError

GradientError is Gradient error for evaluating alpha matte prediction. To evaluate with GradientError, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='GradientError'),
]
```

1.10.10 ConnectivityError

ConnectivityError is Connectivity error for evaluating alpha matte prediction. To evaluate with ConnectivityError, please add the following configuration in the config file:

```
val_evaluator = [
    dict(type='ConnectivityError'),
]
```

1.10.11 FID and TransFID

Fréchet Inception Distance is a measure of similarity between two datasets of images. It was shown to correlate well with the human judgment of visual quality and is most often used to evaluate the quality of samples of Generative Adversarial Networks. FID is calculated by computing the Fréchet distance between two Gaussians fitted to feature representations of the Inception network.

In `MMEditing`, we provide two versions for FID calculation. One is the commonly used PyTorch version and the other one is used in StyleGAN paper. Meanwhile, we have compared the difference between these two implementations in the StyleGAN2-FFHQ1024 model (the details can be found here). Fortunately, there is a marginal difference in the final results. Thus, we recommend users adopt the more convenient PyTorch version.

About PyTorch version and Tero's version: The commonly used PyTorch version adopts the modified InceptionV3 network to extract features for real and fake images. However, Tero's FID requires a script module for Tensorflow InceptionV3. Note that applying this script module needs `PyTorch >= 1.6.0`.

About extracting real inception data: For the users' convenience, the real features will be automatically extracted at test time and saved locally, and the stored features will be automatically read at the next test. Specifically, we will calculate a hash value based on the parameters used to calculate the real features, and use the hash value to mark the feature file, and when testing, if the `inception_pkl` is not set, we will look for the feature in `MMEDIT_CACHE_DIR` (~/.cache/openmmlab/mmedit/). If cached inception pkl is not found, then extracting will be performed.

To use the FID metric, you should add the metric in a config file like this:

```
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
        sample_model='ema')
]
```

If you work on an new machine, then you can copy the `pkl` files in `MMEDIT_CACHE_DIR` and copy them to new machine and set `inception_pkl` field.

```
metrics = [
    dict(
        type='FrechetInceptionDistance',
        prefix='FID-Full-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
        inception_pkl=
        'work_dirs/inception_pkl/inception_state-capture_mean_cov-full-
↪33ad4546f8c9152e4b3bdb1b0c08dbaf.pkl',  # copied from old machine
        sample_model='ema')
]
```

`TransFID` has same usage as `FID`, but it's designed for translation models like `Pix2Pix` and `CycleGAN`, which is adapted for our evaluator. You can refer to *evaluation* for details.

1.10.12 IS and TransIS

Inception score is an objective metric for evaluating the quality of generated images, proposed in Improved Techniques for Training GANs. It uses an InceptionV3 model to predict the class of the generated images, and suppose that 1) If an image is of high quality, it will be categorized into a specific class. 2) If images are of high diversity, the range of images' classes will be wide. So the KL-divergence of the conditional probability and marginal probability can indicate the quality and diversity of generated images. You can see the complete implementation in `metrics.py`, which refers to https://github.com/sbarratt/inception-score-pytorch/blob/master/inception_score.py. If you want to evaluate models with `IS` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/biggan/biggan_2xb25-500kiters_cifar10-32x32.py
metrics = [
    xxx,
    dict(
        type='IS',
        prefix='IS-50k',
        fake_nums=50000,
        inception_style='StyleGAN',
```

(continues on next page)

(continued from previous page)

```
        sample_model='ema')
]
```

To be noted that, the selection of Inception V3 and image resize method can significantly influence the final IS score. Therefore, we strongly recommend users may download the Tero's script model of Inception V3 (load this script model need torch >= 1.6) and use `Bicubic` interpolation with `Pillow` backend.

Corresponding to config, you can set `resize_method` and `use_pillow_resize` for image resizing. You can also set `inception_style` as `StyleGAN` for recommended tero's inception model, or `PyTorch` for torchvision's implementation. For environment without internet, you can download the inception's weights, and set `inception_path` to your inception model.

We also perform a survey on the influence of data loading pipeline and the version of pretrained Inception V3 on the IS result. All IS are evaluated on the same group of images which are randomly selected from the ImageNet dataset.

`TransIS` has same usage as `IS`, but it's designed for translation models like `Pix2Pix` and `CycleGAN`, which is adapted for our evaluator. You can refer to *evaluation* for details.

1.10.13 Precision and Recall

Our `Precision and Recall` implementation follows the version used in StyleGAN2. In this metric, a VGG network will be adopted to extract the features for images. Unfortunately, we have not found a PyTorch VGG implementation leading to similar results with Tero's version used in StyleGAN2. (About the differences, please see this file.) Thus, in our implementation, we adopt Teor's VGG network by default. Importantly, applying this script module needs `PyTorch >= 1.6.0`. If with a lower PyTorch version, we will use the PyTorch official VGG network for feature extraction.

To evaluate with `P&R`, please add the following configuration in the config file:

```
metrics = [
    dict(type='PrecisionAndRecall', fake_nums=50000, prefix='PR-50K')
]
```

1.10.14 PPL

Perceptual path length measures the difference between consecutive images (their VGG16 embeddings) when interpolating between two random inputs. Drastic changes mean that multiple features have changed together and that they might be entangled. Thus, a smaller PPL score appears to indicate higher overall image quality by experiments.
As a basis for our metric, we use a perceptually-based pairwise image distance that is calculated as a weighted difference between two VGG16 embeddings, where the weights are fit so that the metric agrees with human perceptual similarity judgments. If we subdivide a latent space interpolation path into linear segments, we can define the total perceptual length of this segmented path as the sum of perceptual differences over each segment, and a natural definition for the perceptual path length would be the limit of this sum under infinitely fine subdivision, but in practice we approximate it using a small subdivision $`\epsilon=10^{-4}`$. The average perceptual path length in latent `space` Z, over all possible endpoints, is therefore

$$`L_Z = E[\frac{1}{\epsilon^2}d(G(slerp(z_1,z_2;t))), G(slerp(z_1,z_2;t+\epsilon)))]`$$

Computing the average perceptual path length in latent `space` W is carried out in a similar fashion:

$$`L_Z = E[\frac{1}{\epsilon^2}d(G(slerp(z_1,z_2;t))), G(slerp(z_1,z_2;t+\epsilon)))]`$$

Where $`z_1, z_2 \sim P(z)`$, and $` t \sim U(0,1)`$ if we set `sampling` to full, $` t \in \{0,1\}`$ if we set `sampling` to end. $` G`$ is the generator(i.e. $` g \circ f`$ for style-based networks), and $` d(., .)`$ evaluates the perceptual distance between the resulting images.We compute the expectation by taking 100,000 samples (set `num_images` to 50,000 in our code).

You can find the complete implementation in `metrics.py`, which refers to https://github.com/rosinality/stylegan2-pytorch/blob/master/ppl.py. If you want to evaluate models with `PPL` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/styleganv2/stylegan2_c2_ffhq_1024_b4x8.py
metrics = [
    xxx,
    dict(type='PerceptualPathLength', fake_nums=50000, prefix='ppl-w')
]
```

1.10.15 SWD

Sliced Wasserstein distance is a discrepancy measure for probability distributions, and smaller distance indicates generated images look like the real ones. We obtain the Laplacian pyramids of every image and extract patches from the Laplacian pyramids as descriptors, then SWD can be calculated by taking the sliced Wasserstein distance of the real and fake descriptors. You can see the complete implementation in `metrics.py`, which refers to https://github.com/tkarras/progressive_growing_of_gans/blob/master/metrics/sliced_wasserstein.py. If you want to evaluate models with `SWD` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py
metrics = [
    dict(
        type='SWD',
        prefix='swd',
        fake_nums=16384,
        sample_model='orig',
        image_shape=(3, 64, 64))
]
```

1.10.16 MS-SSIM

Multi-scale structural similarity is used to measure the similarity of two images. We use MS-SSIM here to measure the diversity of generated images, and a low MS-SSIM score indicates the high diversity of generated images. You can see the complete implementation in `metrics.py`, which refers to https://github.com/tkarras/progressive_growing_of_gans/blob/master/metrics/ms_ssim.py. If you want to evaluate models with `MS-SSIM` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py
metrics = [
    dict(
        type='MS_SSIM', prefix='ms-ssim', fake_nums=10000,
        sample_model='orig')
]
```

1.10.17 Equivarience

Equivarience of generative models refer to the exchangeability of model forward and geometric transformations. Currently this metric is only calculated for StyleGANv3, you can see the complete implementation in `metrics.py`, which refers to https://github.com/NVlabs/stylegan3/blob/main/metrics/equivariance.py. If you want to evaluate models with `Equivarience` metrics, you can add the `metrics` into your config file like this:

```
# at the end of the configs/styleganv3/stylegan3-t_gamma2.0_8xb4-fp16-noaug_ffhq-256x256.
↪py
metrics = [
    dict(
        type='Equivariance',
        fake_nums=50000,
        sample_mode='ema',
        prefix='EQ',
        eq_cfg=dict(
            compute_eqt_int=True, compute_eqt_frac=True, compute_eqr=True))
]
```

1.11 Tutorial 6: Visualization

The visualization of images is an important way to measure the quality of image processing, editing and synthesis. Using `visualizer` in config file can save visual results when training or testing. You can follow MMEngine Documents to learn the usage of visualization. MMEditing provides a rich set of visualization functions. In this tutorial, we introduce the usage of the visualization functions provided by MMEditing.

- *Overview*
- *Visualization hook*
- *Visualizer*
- *VisBackend*

1.11.1 Overview

In MMEditing, the visualization of the training or testing process requires the configuration of three components: VisualizationHook, Visualizer, and VisBackend.

VisualizationHook fetches the visualization results of the model output in fixed intervals during training and passes them to Visualizer. **Visualizer** is responsible for converting the original visualization results into the desired type (png, gif, etc.) and then transferring them to **VisBackend** for storage or display.

Visualization configuration of GANs

For GAN models, such as StyleGAN and SAGAN, a usual configuration is shown below:

```
# VisualizationHook
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,  # visualization interval
```

(continues on next page)

(continued from previous page)

```
        fixed_input=True,  # whether use fixed noise input to generate images
        vis_kwargs_list=dict(type='GAN', name='fake_img')  # pre-defined visualization␣
↪arguments for GAN models
    )
]
# VisBackend
vis_backends = [
    dict(type='GenVisBackend'),  # vis_backend for saving images to file system
    dict(type='WandbGenVisBackend',  # vis_backend for uploading images to Wandb
        init_kwargs=dict(
            project='MMEditing',   # project name for Wandb
            name='GAN-Visualization-Demo'  # name of the experiment for Wandb
        ))
]
# Visualizer
visualizer = dict(type='GenVisualizer', vis_backends=vis_backends)
```

If you apply Exponential Moving Average (EMA) to a generator and want to visualize the EMA model, you can modify config of `VisualizationHook` as below:

```
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        # vis ema and orig in `fake_img` at the same time
        vis_kwargs_list=dict(
            type='Noise',
            name='fake_img',  # save images with prefix `fake_img`
            sample_model='ema/orig',  # specified kwargs for `NoiseSampler`
            target_keys=['ema.fake_img', 'orig.fake_img']  # specific key to␣
↪visualization
        ))
]
```

Visualization configuration of image translation models

For Translation models, such as CycleGAN and Pix2Pix, visualization configs can be formed as below:

```
# VisualizationHook
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        vis_kwargs_list=[
            dict(
                type='Translation',  # Visualize results on the training set
                name='trans'),  #  save images with prefix `trans`
            dict(
                type='Translationval',  # Visualize results on the validation set
```

(continues on next page)

(continued from previous page)

```
                name='trans_val'),  #  save images with prefix `trans_val`
        ])
]
# VisBackend
vis_backends = [
    dict(type='GenVisBackend'),  # vis_backend for saving images to file system
    dict(type='WandbGenVisBackend',  # vis_backend for uploading images to Wandb
        init_kwargs=dict(
            project='MMEditing',   # project name for Wandb
            name='Translation-Visualization-Demo'  # name of the experiment for Wandb
        ))
]
# Visualizer
visualizer = dict(type='GenVisualizer', vis_backends=vis_backends)
```

Visualization configuration of diffusion models

For Diffusion models, such as Improved-DDPM, we can use the following configuration to visualize the denoising process through a gif:

```
# VisualizationHook
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        vis_kwargs_list=dict(type='DDPMDenoising'))  # pre-defined visualization␣
↪argument for DDPM models
]
# VisBackend
vis_backends = [
    dict(type='GenVisBackend'),  # vis_backend for saving images to file system
    dict(type='WandbGenVisBackend',  # vis_backend for uploading images to Wandb
        init_kwargs=dict(
            project='MMEditing',   # project name for Wandb
            name='Diffusion-Visualization-Demo'  # name of the experiment for Wandb
        ))
]
# Visualizer
visualizer = dict(type='GenVisualizer', vis_backends=vis_backends)
```

Visualization configuration of inpainting models

For inpainting models, such as AOT-GAN and Global&Local, a usual configuration is shown below:

```
# VisBackend
vis_backends = [dict(type='LocalVisBackend')]
# Visualizer
visualizer = dict(
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
    bgr2rgb=True)
# VisualizationHook
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

Visualization configuration of matting models

For matting models, such as DIM and GCA, a usual configuration is shown below:

```
# VisBackend
vis_backends = [dict(type='LocalVisBackend')]
# Visualizer
visualizer = dict(
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='trimap_path',
    img_keys=['pred_alpha', 'trimap', 'gt_merged', 'gt_alpha'],
    bgr2rgb=True)
# VisualizationHook
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

Visualization configuration of SISR/VSR/VFI models

For SISR/VSR/VFI models, such as EDSR, EDVR and CAIN, a usual configuration is shown below:

```
# VisBackend
vis_backends = [dict(type='LocalVisBackend')]
# Visualizer
visualizer = dict(
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
    bgr2rgb=False)
# VisualizationHook
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

The specific configuration of the `VisualizationHook`, `Visualizer` and `VisBackend` components are described below

1.11.2 Visualization Hook

In MMEditing, we use `BasicVisualizationHook` and `GenVisualizationHook` as `VisualizationHook`. `GenVisualizationHook` supports three following cases.

(1) Modify `vis_kwargs_list` to visualize the output of the model under specific inputs , which is suitable for visualization of the generated results of GAN and translation results of Image-to-Image-Translation models under specific data input, etc. Below are two typical examples:

```
# input as dict
vis_kwargs_list = dict(
    type='Noise',  # use 'Noise' sampler to generate model input
    name='fake_img',  # define prefix of saved images
)

# input as list of dict
vis_kwargs_list = [
    dict(type='Arguments',  # use `Arguments` sampler to generate model input
         name='arg_output',  # define prefix of saved images
         vis_mode='gif',  # specific visualization mode as GIF
         forward_kwargs=dict(forward_mode='sampling', sample_kwargs=dict(show_
↪pbar=True))  # specific kwargs for `Arguments` sampler
    ),
    dict(type='Data',  # use `Data` sampler to feed data in dataloader to model as input
         n_samples=36,  # specific how many samples want to generate
         fixed_input=False,  # specific do not use fixed input for each visualization␣
↪process
    )
]
```

`vis_kwargs_list` takes dict or list of dict as input. Each of dict must contain a `type` field indicating the **type of sampler** used to generate the model input, and each of the dict must also contain the keyword fields necessary for the sampler (e.g. `ArgumentSampler` requires that the argument dictionary contain `forward_kwargs`).

To be noted that, this content is checked by the corresponding sampler and is not restricted by `GenVisHook`.

In addition, the other fields are generic fields (e.g. `n_samples`, `n_row`, `name`, `fixed_input`, etc.). If not passed in, the default values from the GenVisHook initialization will be used.

For the convenience of users, MMEditing has pre-defined visualization parameters for **GAN**, **Translation models**, **SinGAN** and **Diffusion models**, and users can directly use the predefined visualization methods by using the following configuration:

```
vis_kwargs_list = dict(type='GAN')
vis_kwargs_list = dict(type='SinGAN')
vis_kwargs_list = dict(type='Translation')
vis_kwargs_list = dict(type='TranslationVal')
vis_kwargs_list = dict(type='TranslationTest')
vis_kwargs_list = dict(type='DDPMDenoising')
```

1.11.3 Visualizer

In MMEditing, we implement `ConcatImageVisualizer` and `GenVisualizer`, which inherit from `mmengine.Visualizer`. The base class of `Visualizer` is `ManagerMixin` and this makes `Visualizer` a globally unique object. After being instantiated, `Visualizer` can be called at anywhere of the code by `Visualizer.get_current_instance()`, as shown below:

```
# configs
vis_backends = [dict(type='GenVisBackend')]
visualizer = dict(
    type='GenVisualizer', vis_backends=vis_backends, name='visualizer')
```

```
# `get_instance()` is called for globally unique instantiation
VISUALIZERS.build(cfg.visualizer)

# Once instantiated by the above code, you can call the `get_current_instance` method at␣
↪any location to get the visualizer
visualizer = Visualizer.get_current_instance()
```

The core interface of `Visualizer` is `add_datasample`. Through this interface, This interface will call the corresponding drawing function according to the corresponding `vis_mode` to obtain the visualization result in `np.ndarray` type. Then `show` or `add_image` will be called to directly show the results or pass the visualization result to the predefined vis_backend.

1.11.4 VisBackend

In general, users do not need to manipulate `VisBackend` objects, only when the current visualization storage can not meet the needs, users will want to manipulate the storage backend directly. MMEditing supports a variety of different visualization backends, including:

- Basic VisBackend of MMEngine: including LocalVisBackend, TensorboardVisBackend and WandbVisBackend. You can follow MMEngine Documents to learn more about them
- GenVisBackend: Backend for **File System**. Save the visualization results to the corresponding position.
- TensorboardGenVisBackend: Backend for **Tensorboard**. Send the visualization results to Tensorboard.
- PaviGenVisBackend: Backend for **Pavi**. Send the visualization results to Tensorboard.
- WandbGenVisBackend: Backend for **Wandb**. Send the visualization results to Tensorboard.

One `Visualizer` object can have access to any number of VisBackends and users can access to the backend by their class name in their code.

```
# configs
vis_backends = [dict(type='GenVisualizer'), dict(type='WandbVisBackend')]
visualizer = dict(
    type='GenVisualizer', vis_backends=vis_backends, name='visualizer')
```

```
# code
VISUALIZERS.build(cfg.visualizer)
visualizer = Visualizer.get_current_instance()

# access to the backend by class name
gen_vis_backend = visualizer.get_backend('GenVisBackend')
gen_wandb_vis_backend = visualizer.get_backend('GenWandbVisBackend')
```

When there are multiply VisBackend with the same class name, user must specific name for each VisBackend.

```
# configs
vis_backends = [
    dict(type='GenVisBackend', name='gen_vis_backend_1'),
    dict(type='GenVisBackend', name='gen_vis_backend_2')
]
visualizer = dict(
    type='GenVisualizer', vis_backends=vis_backends, name='visualizer')
```

```
# code
VISUALIZERS.build(cfg.visualizer)
visualizer = Visualizer.get_current_instance()

local_vis_backend_1 = visualizer.get_backend('gen_vis_backend_1')
local_vis_backend_2 = visualizer.get_backend('gen_vis_backend_2')
```

1.12 Tutorial 7: Useful tools

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

The structure of this guide is as follows:

- *Get the FLOPs and params*
- *Publish a model*
- *Print full config*

1.12.1 Get the FLOPs and params

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

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

For example,

```
python tools/analysis_tools/get_flops.py configs/resotorer/srresnet.py --shape 40 40
```

You will get the result like this.

```
==============================
Input shape: (3, 40, 40)
Flops: 4.07 GMac
Params: 1.52 M
==============================
```

Note: This tool is still experimental and we do not guarantee that the number is correct. You may well use the result for simple comparisons, but double check it before you adopt it in technical reports or papers.

(1) FLOPs are related to the input shape while parameters are not. The default input shape is (1, 3, 250, 250). (2) Some operators are not counted in FLOPs like GN and custom operators. You can add support for new operators by modifying `mmcv/cnn/utils/flops_counter.py`.

1.12.2 Publish a model

Before you upload a model to AWS, you may want to

1. convert model weights to CPU tensors
2. delete the optimizer states and
3. compute the hash of the checkpoint file and append time and the hash id to the filename.

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

E.g.,

```
python tools/model_converters/publish_model.py work_dirs/stylegan2/latest.pth stylegan2_
↪c2_8xb4_ffhq-1024x1024.pth
```

The final output filename will be `stylegan2_c2_8xb4_ffhq-1024x1024_{time}-{hash id}.pth`.

1.12.3 Print full config

MMGeneration incorporates config mechanism to set parameters used for training and testing models. With our *config* mechanism, users can easily conduct extensive experiments without hard coding. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config.

An Example:

```
python tools/misc/print_config.py configs/styleganv2/stylegan2_c2-PL_8xb4-fp16-partial-
↪GD-no-scaler-800kiters_ffhq-256x256.py
```

1.13 Tutorial 8: Deploy models in MMEditing

The deployment of OpenMMLab codebases, including MMClassification, MMDetection, MMEditing and so on are supported by MMDeploy. The latest deployment guide for MMEditing can be found from here.

This tutorial is organized as follows:

- *Installation*
- *Convert model*
- *Model specification*
- *Model inference*
  - *Backend model inference*
  - *SDK model inference*
- *Supported models*

1.13.1 Installation

Please follow the *guide* to install mmedit. And then install mmdeploy from source by following this guide.

Note: If you install mmdeploy prebuilt package, please also clone its repository by 'git clone https://github.com/open-mmlab/mmdeploy.git –depth=1' to get the deployment config files.

1.13.2 Convert model

Suppose mmediting and mmdeploy repositories are in the same directory, and the working directory is the root path of mmediting.

Take ESRGAN model as an example. You can download its checkpoint from here, and then convert it to onnx model as follows:

```
from mmdeploy.apis import torch2onnx
from mmdeploy.backend.sdk.export_info import export2SDK

img = 'tests/data/image/face/000001.png'
work_dir = 'mmdeploy_models/mmedit/onnx'
save_file = 'end2end.onnx'
deploy_cfg = '../mmdeploy/configs/mmedit/super-resolution/super-resolution_onnxruntime_
↪dynamic.py'
model_cfg = 'configs/esrgan/esrgan_psnr-x4c64b23g32_1xb16-1000k_div2k.py'
model_checkpoint = 'esrgan_psnr_x4c64b23g32_1x16_1000k_div2k_20200420-bf5c993c.pth'
device = 'cpu'

# 1. convert model to onnx
torch2onnx(img, work_dir, save_file, deploy_cfg, model_cfg,
  model_checkpoint, device)

# 2. extract pipeline info for inference by MMDeploy SDK
export2SDK(deploy_cfg, model_cfg, work_dir, pth=model_checkpoint, device=device)
```

It is crucial to specify the correct deployment config during model conversion.MMDeploy has already provided builtin deployment config files of all supported backends for mmedit, under which the config file path follows the pattern:

```
{task}/{task}_{backend}-{precision}_{static | dynamic}_{shape}.py
```

- **{task}:** task in mmedit.
- **{backend}:** inference backend, such as onnxruntime, tensorrt, pplnn, ncnn, openvino, coreml etc.
- **{precision}:** fp16, int8. When it's empty, it means fp32
- **{static | dynamic}:** static shape or dynamic shape
- **{shape}:** input shape or shape range of a model

Therefore, in the above example, you can also convert `ESRGAN` to other backend models by changing the deployment config file, e.g., converting to tensorrt-fp16 model by `super-resolution_tensorrt-fp16_dynamic-32x32-512x512.py`.

Tip: When converting mmedit models to tensorrt models, –device should be set to "cuda"

1.13.3 Model specification

Before moving on to model inference chapter, let's know more about the converted model structure which is very important for model inference.

The converted model locates in the working directory like `mmdeploy_models/mmedit/onnx` in the previous example. It includes:

```
mmdeploy_models/mmedit/onnx
├── deploy.json
├── detail.json
├── end2end.onnx
└── pipeline.json
```

in which,

- **end2end.onnx**: backend model which can be inferred by ONNX Runtime
- ***xxx*.json**: the necessary information for mmdeploy SDK

The whole package **mmdeploy_models/mmedit/onnx** is defined as **mmdeploy SDK model**, i.e., **mmdeploy SDK model** includes both backend model and inference meta information.

1.13.4 Model inference

Backend model inference

Take the previous converted `end2end.onnx` model as an example, you can use the following code to inference the model.

```python
from mmdeploy.apis.utils import build_task_processor
from mmdeploy.utils import get_input_shape, load_config
import torch

deploy_cfg = '../mmdeploy/configs/mmedit/super-resolution/super-resolution_onnxruntime_
↪dynamic.py'
model_cfg = 'configs/esrgan/esrgan_psnr-x4c64b23g32_1xb16-1000k_div2k.py'
device = 'cpu'
backend_model = ['mmdeploy_models/mmedit/onnx/end2end.onnx']
image = 'tests/data/image/lq/baboon_x4.png'

# read deploy_cfg and model_cfg
deploy_cfg, model_cfg = load_config(deploy_cfg, model_cfg)

# build task and backend model
task_processor = build_task_processor(model_cfg, deploy_cfg, device)
model = task_processor.build_backend_model(backend_model)

# process input image
input_shape = get_input_shape(deploy_cfg)
model_inputs, _ = task_processor.create_input(image, input_shape)

# do model inference
with torch.no_grad():
```

(continues on next page)

(continued from previous page)

```
    result = model.test_step(model_inputs)

# visualize results
task_processor.visualize(
    image=image,
    model=model,
    result=result[0],
    window_name='visualize',
    output_file='output_restorer.bmp')
```

SDK model inference

You can also perform SDK model inference like following,

```
from mmdeploy_python import Restorer
import cv2

img = cv2.imread('tests/data/image/lq/baboon_x4.png')

# create a predictor
restorer = Restorer(model_path='mmdeploy_models/mmedit/onnx', device_name='cpu', device_
↪id=0)
# perform inference
result = restorer(img)

# visualize inference result
cv2.imwrite('output_restorer.bmp', result)
```

Besides python API, MMDeploy SDK also provides other FFI (Foreign Function Interface), such as C, C++, C#, Java and so on. You can learn their usage from demos.

1.13.5 Supported models

Please refer to here for the supported model list.

1.14 Evaluator [Coming Soon!]

We're improving this documentation. Don't hesitate to join us!

Make a pull request or discuss with us!

1.15 Data structure in MMEditing [Coming Soon!]

We're improving this documentation. Don't hesitate to join us!

Make a pull request or discuss with us!

1.16 Data pre-processor [Coming Soon!]

We're improving this documentation. Don't hesitate to join us!

Make a pull request or discuss with us!

1.17 Data flow in MMEditing [Coming Soon!]

We're improving this documentation. Don't hesitate to join us!

Make a pull request or discuss with us!

1.18 How to design your own models

MMEditing is built upon MMEngine and MMCV, which enables users to design new models quickly, train and evaluate them easily. In this section, you will learn how to design your own models.

The structure of this guide are as follows:

1.18.1 Overview of models in MMEditing

In MMEditing, one algorithm can be splited two compents: **Model** and **Module**.

- **Model** are topmost wrappers and always inherint from `BaseModel` provided in MMEngine. **Model** is responsible to network forward, loss calculation and backward, parameters updating, etc. In MMEditing, **Model** should be registered as `MODELS`.
- **Module** includes the neural network **architectures** to train or inference, pre-defined **loss classes**, and **data preprocessors** to preprocess the input data batch. **Module** always present as elements of **Model**. In MMEditing, **Module** should be registered as **MODULES**.

Take DCGAN model as an example, generator and discriminator are the **Module**, which generate images and discriminate real or fake images. `DCGAN` is the **Model**, which take data from dataloader and train generator and discriminator alternatively.

You can find the implementation of **Model** and **Module** by the following link.

- **Model**:
  - Editors
- **Module**:
  - Layers
  - Losses
  - Data Preprocessor

1.18.2 An example of SRCNN

Here, we take the implementation of the classical image super-resolution model, SRCNN [1], as an example.

Step 1: Define the network of SRCNN

SRCNN is the first deep learning method for single image super-resolution [1]. To implement the network architecture of SRCNN, we need to create a new file `mmedit/models/editors/srgan/sr_resnet.py` and implement `class MSRResNet`.

In this step, we implement `class MSRResNet` by inheriting from `mmengine.models.BaseModule` and define the network architecture in `__init__` function. In particular, we need to use `@MODELS.register_module()` to add the implementation of `class MSRResNet` into the registration of MMEditing.

```
import torch.nn as nn
from mmengine.model import BaseModule
from mmedit.registry import MODELS

from mmedit.models.utils import (PixelShufflePack, ResidualBlockNoBN,
                                 default_init_weights, make_layer)


@MODELS.register_module()
class MSRResNet(BaseModule):
    """Modified SRResNet.

    A compacted version modified from SRResNet in "Photo-Realistic Single
```

(continues on next page)

(continued from previous page)

```
    Image Super-Resolution Using a Generative Adversarial Network".

    It uses residual blocks without BN, similar to EDSR.
    Currently, it supports x2, x3 and x4 upsampling scale factor.

    Args:
        in_channels (int): Channel number of inputs.
        out_channels (int): Channel number of outputs.
        mid_channels (int): Channel number of intermediate features.
            Default: 64.
        num_blocks (int): Block number in the trunk network. Default: 16.
        upscale_factor (int): Upsampling factor. Support x2, x3 and x4.
            Default: 4.
    """
    _supported_upscale_factors = [2, 3, 4]

    def __init__(self,
                 in_channels,
                 out_channels,
                 mid_channels=64,
                 num_blocks=16,
                 upscale_factor=4):

        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.mid_channels = mid_channels
        self.num_blocks = num_blocks
        self.upscale_factor = upscale_factor

        self.conv_first = nn.Conv2d(
            in_channels, mid_channels, 3, 1, 1, bias=True)
        self.trunk_net = make_layer(
            ResidualBlockNoBN, num_blocks, mid_channels=mid_channels)

        # upsampling
        if self.upscale_factor in [2, 3]:
            self.upsample1 = PixelShufflePack(
                mid_channels,
                mid_channels,
                self.upscale_factor,
                upsample_kernel=3)
        elif self.upscale_factor == 4:
            self.upsample1 = PixelShufflePack(
                mid_channels, mid_channels, 2, upsample_kernel=3)
            self.upsample2 = PixelShufflePack(
                mid_channels, mid_channels, 2, upsample_kernel=3)
        else:
            raise ValueError(
                f'Unsupported scale factor {self.upscale_factor}. '
                f'Currently supported ones are '
                f'{self._supported_upscale_factors}.')
```

(continues on next page)

(continued from previous page)

```
        self.conv_hr = nn.Conv2d(
            mid_channels, mid_channels, 3, 1, 1, bias=True)
        self.conv_last = nn.Conv2d(
            mid_channels, out_channels, 3, 1, 1, bias=True)

        self.img_upsampler = nn.Upsample(
            scale_factor=self.upscale_factor,
            mode='bilinear',
            align_corners=False)

        # activation function
        self.lrelu = nn.LeakyReLU(negative_slope=0.1, inplace=True)

        self.init_weights()

    def init_weights(self):
        """Init weights for models.

        Args:
            pretrained (str, optional): Path for pretrained weights. If given
                None, pretrained weights will not be loaded. Defaults to None.
            strict (boo, optional): Whether strictly load the pretrained model.
                Defaults to True.
        """

        for m in [self.conv_first, self.conv_hr, self.conv_last]:
            default_init_weights(m, 0.1)
```

Then, we implement the `forward` function of `class MSRResNet`, which takes as input tensor and then returns the results from `MSRResNet`.

```
    def forward(self, x):
        """Forward function.

        Args:
            x (Tensor): Input tensor with shape (n, c, h, w).

        Returns:
            Tensor: Forward results.
        """

        feat = self.lrelu(self.conv_first(x))
        out = self.trunk_net(feat)

        if self.upscale_factor in [2, 3]:
            out = self.upsample1(out)
        elif self.upscale_factor == 4:
            out = self.upsample1(out)
            out = self.upsample2(out)
```

(continues on next page)

(continued from previous page)

```
        out = self.conv_last(self.lrelu(self.conv_hr(out)))
        upsampled_img = self.img_upsampler(x)
        out += upsampled_img
        return out
```

After the implementation of `class MSRResNet`, we need to update the model list in `mmedit/models/editors/__init__.py`, so that we can import and use `class MSRResNet` by `mmedit.models.editors`.

```
from .srgan.sr_resnet import MSRResNet
```

Step 2: Define the model of SRCNN

After the implementation of the network architecture, we need to define our model `class BaseEditModel` and implement the forward loop of `class BaseEditModel`.

To implement `class BaseEditModel`, we create a new file `mmedit/models/base_models/base_edit_model.py`. Specifically, `class BaseEditModel` inherits from `mmengine.model.BaseModel`. In the `__init__` function, we define the loss functions, training and testing configurations, networks of `class BaseEditModel`.

```
from typing import List, Optional

import torch
from mmengine.model import BaseModel

from mmedit.registry import MODELS
from mmedit.structures import EditDataSample, PixelData


@MODELS.register_module()
class BaseEditModel(BaseModel):
    """Base model for image and video editing.

    It must contain a generator that takes frames as inputs and outputs an
    interpolated frame. It also has a pixel-wise loss for training.

    Args:
        generator (dict): Config for the generator structure.
        pixel_loss (dict): Config for pixel-wise loss.
        train_cfg (dict): Config for training. Default: None.
        test_cfg (dict): Config for testing. Default: None.
        init_cfg (dict, optional): The weight initialized config for
            :class:`BaseModule`.
        data_preprocessor (dict, optional): The pre-process config of
            :class:`BaseDataPreprocessor`.

    Attributes:
        init_cfg (dict, optional): Initialization config dict.
        data_preprocessor (:obj:`BaseDataPreprocessor`): Used for
            pre-processing data sampled by dataloader to the format accepted by
            :meth:`forward`. Default: None.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self,
                 generator,
                 pixel_loss,
                 train_cfg=None,
                 test_cfg=None,
                 init_cfg=None,
                 data_preprocessor=None):
        super().__init__(
            init_cfg=init_cfg, data_preprocessor=data_preprocessor)

        self.train_cfg = train_cfg
        self.test_cfg = test_cfg

        # generator
        self.generator = MODELS.build(generator)

        # loss
        self.pixel_loss = MODELS.build(pixel_loss)
```

Since `mmengine.model.BaseModel` provides the basic functions of the algorithmic model, such as weights initialize, batch inputs preprocess, parse losses, and update model parameters. Therefore, the subclasses inherit from BaseModel, i.e., `class BaseEditModel` in this example, only need to implement the forward method, which implements the logic to calculate loss and predictions.

Specifically, the implemented `forward` function of `class BaseEditModel` takes as input `batch_inputs` and `data_samples` and return results according to mode arguments.

```
    def forward(self,
                batch_inputs: torch.Tensor,
                data_samples: Optional[List[EditDataSample]] = None,
                mode: str = 'tensor',
                **kwargs):
        """Returns losses or predictions of training, validation, testing, and
        simple inference process.

        ``forward`` method of BaseModel is an abstract method, its subclasses
        must implement this method.

        Accepts ``batch_inputs`` and ``data_samples`` processed by
        :attr:`data_preprocessor`, and returns results according to mode
        arguments.

        During non-distributed training, validation, and testing process,
        ``forward`` will be called by ``BaseModel.train_step``,
        ``BaseModel.val_step`` and ``BaseModel.val_step`` directly.

        During distributed data parallel training process,
        ``MMSeparateDistributedDataParallel.train_step`` will first call
        ``DistributedDataParallel.forward`` to enable automatic
        gradient synchronization, and then call ``forward`` to get training
        loss.

        Args:
```

(continues on next page)

(continued from previous page)

```
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
                data samples collated by :attr:`data_preprocessor`.
            mode (str): mode should be one of ``loss``, ``predict`` and
                ``tensor``

                - ``loss``: Called by ``train_step`` and return loss ``dict``
                  used for logging
                - ``predict``: Called by ``val_step`` and ``test_step``
                  and return list of ``BaseDataElement`` results used for
                  computing metric.
                - ``tensor``: Called by custom use to get ``Tensor`` type
                  results.

        Returns:
            ForwardResults:

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

        if mode == 'tensor':
            return self.forward_tensor(batch_inputs, data_samples, **kwargs)

        elif mode == 'predict':
            return self.forward_inference(batch_inputs, data_samples, **kwargs)

        elif mode == 'loss':
            return self.forward_train(batch_inputs, data_samples, **kwargs)
```

Specifically, in `forward_tensor`, `class BaseEditModel` returns the forward tensors of the network directly.

```
    def forward_tensor(self, batch_inputs, data_samples=None, **kwargs):
        """Forward tensor.
            Returns result of simple forward.

        Args:
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
                data samples collated by :attr:`data_preprocessor`.

        Returns:
            Tensor: result of simple forward.
        """
```

(continues on next page)

(continued from previous page)

```
        feats = self.generator(batch_inputs, **kwargs)

        return feats
```

In `forward_inference` function, `class BaseEditModel` first converts the forward tensors to images and then returns the images as output.

```
    def forward_inference(self, batch_inputs, data_samples=None, **kwargs):
        """Forward inference.
            Returns predictions of validation, testing, and simple inference.

        Args:
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
                data samples collated by :attr:`data_preprocessor`.

        Returns:
            List[EditDataSample]: predictions.
        """

        feats = self.forward_tensor(batch_inputs, data_samples, **kwargs)
        feats = self.data_preprocessor.destructor(feats)
        predictions = []
        for idx in range(feats.shape[0]):
            predictions.append(
                EditDataSample(
                    pred_img=PixelData(data=feats[idx].to('cpu')),
                    metainfo=data_samples[idx].metainfo))

        return predictions
```

In `forward_train`, `class BaseEditModel` calculate the loss function and returns a dictionary contains the losses as output.

```
    def forward_train(self, batch_inputs, data_samples=None, **kwargs):
        """Forward training.
            Returns dict of losses of training.

        Args:
            batch_inputs (torch.Tensor): batch input tensor collated by
                :attr:`data_preprocessor`.
            data_samples (List[BaseDataElement], optional):
                data samples collated by :attr:`data_preprocessor`.

        Returns:
            dict: Dict of losses.
        """

        feats = self.forward_tensor(batch_inputs, data_samples, **kwargs)
        gt_imgs = [data_sample.gt_img.data for data_sample in data_samples]
        batch_gt_data = torch.stack(gt_imgs)
```

(continues on next page)

(continued from previous page)

```
        loss = self.pixel_loss(feats, batch_gt_data)

        return dict(loss=loss)
```

After the implementation of `class BaseEditModel`, we need to update the model list in `mmedit/models/__init__.py`, so that we can import and use `class BaseEditModel` by `mmedit.models`.

```
from .base_models.base_edit_model import BaseEditModel
```

Step 3: Start training SRCNN

After implementing the network architecture and the forward loop of SRCNN, now we can create a new file `configs/srcnn/srcnn_x4k915_g1_1000k_div2k.py` to set the configurations needed by training SRCNN.

In the configuration file, we need to specify the parameters of our model, `class BaseEditModel`, including the generator network architecture, loss function, additional training and testing configuration, and data preprocessor of input tensors. Please refer to the *Introduction to the loss in MMEditing* for more details of losses in MMEditing.

```
# model settings
model = dict(
    type='BaseEditModel',
    generator=dict(
        type='SRCNNNet',
        channels=(3, 64, 32, 3),
        kernel_sizes=(9, 1, 5),
        upscale_factor=scale),
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean'),
    data_preprocessor=dict(
        type='EditDataPreprocessor',
        mean=[0., 0., 0.],
        std=[255., 255., 255.],
    ))
```

We also need to specify the training dataloader and testing dataloader according to create your own dataloader. Finally we can start training our own model by

```
python train.py configs/srcnn/srcnn_x4k915_g1_1000k_div2k.py
```

1.18.3 An example of DCGAN

Here, we take the implementation of the classical gan model, DCGAN [2], as an example.

Step 1: Define the network of DCGAN

DCGAN is a classical image generative adversarial network [2]. To implement the network architecture of DCGAN, we need to create tow new files `mmedit/models/editors/dcgan/dcgan_generator.py` and `mmedit/models/editors/dcgan/dcgan_discriminator.py`, and implement generator (`class DCGANGenerator`) and discriminator (`class DCGANDiscriminator`).

In this step, we implement `class DCGANGenerator`, `class DCGANDiscriminator` and define the network architecture in `__init__` function. In particular, we need to use `@MODULES.register_module()` to add the generator and discriminator into the registration of MMEditing.

Take the following code as example:

```
import torch.nn as nn
from mmcv.cnn import ConvModule
from mmcv.runner import load_checkpoint
from mmcv.utils.parrots_wrapper import _BatchNorm
from mmengine.logging import MMLogger
from mmengine.model.utils import normal_init

from mmedit.models.builder import MODULES
from ..common import get_module_device


@MODULES.register_module()
class DCGANGenerator(nn.Module):
    def __init__(self,
                 output_scale,
                 out_channels=3,
                 base_channels=1024,
                 input_scale=4,
                 noise_size=100,
                 default_norm_cfg=dict(type='BN'),
                 default_act_cfg=dict(type='ReLU'),
                 out_act_cfg=dict(type='Tanh'),
                 pretrained=None):
        super().__init__()
        self.output_scale = output_scale
        self.base_channels = base_channels
        self.input_scale = input_scale
        self.noise_size = noise_size

        # the number of times for upsampling
        self.num_upsamples = int(np.log2(output_scale // input_scale))

        # output 4x4 feature map
        self.noise2feat = ConvModule(
            noise_size,
            base_channels,
            kernel_size=4,
            stride=1,
            padding=0,
            conv_cfg=dict(type='ConvTranspose2d'),
            norm_cfg=default_norm_cfg,
```

(continues on next page)

(continued from previous page)

```
            act_cfg=default_act_cfg)

        # build up upsampling backbone (excluding the output layer)
        upsampling = []
        curr_channel = base_channels
        for _ in range(self.num_upsamples - 1):
            upsampling.append(
                ConvModule(
                    curr_channel,
                    curr_channel // 2,
                    kernel_size=4,
                    stride=2,
                    padding=1,
                    conv_cfg=dict(type='ConvTranspose2d'),
                    norm_cfg=default_norm_cfg,
                    act_cfg=default_act_cfg))

            curr_channel //= 2

        self.upsampling = nn.Sequential(*upsampling)

        # output layer
        self.output_layer = ConvModule(
            curr_channel,
            out_channels,
            kernel_size=4,
            stride=2,
            padding=1,
            conv_cfg=dict(type='ConvTranspose2d'),
            norm_cfg=None,
            act_cfg=out_act_cfg)
```

Then, we implement the `forward` function of `DCGANGenerator`, which takes as `noise` tensor or `num_batches` and then returns the results from `DCGANGenerator`.

```
    def forward(self, noise, num_batches=0, return_noise=False):
        noise_batch = noise_batch.to(get_module_device(self))
        x = self.noise2feat(noise_batch)
        x = self.upsampling(x)
        x = self.output_layer(x)
        return x
```

If you want to implement specific weights initialization method for you network, you need add `init_weights` function by yourself.

```
    def init_weights(self, pretrained=None):
        if isinstance(pretrained, str):
            logger = MMLogger.get_current_instance()
            load_checkpoint(self, pretrained, strict=False, logger=logger)
        elif pretrained is None:
            for m in self.modules():
                if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
```

(continues on next page)

(continued from previous page)

```
                normal_init(m, 0, 0.02)
            elif isinstance(m, _BatchNorm):
                nn.init.normal_(m.weight.data)
                nn.init.constant_(m.bias.data, 0)
    else:
        raise TypeError('pretrained must be a str or None but'
                        f' got {type(pretrained)} instead.')
```

After the implementation of class `DCGANGenerator`, we need to update the model list in `mmedit/models/editors/__init__.py`, so that we can import and use class `DCGANGenerator` by `mmedit.models.editors`.

Implementation of Class `DCGANDiscriminator` follows the similar logic, and you can find the implementation here.

Step 2: Design the model of DCGAN

After the implementation of the network **Module**, we need to define our **Model** class `DCGAN`.

Your **Model** should inherit from `BaseModel` provided by MMEngine and implement three functions, `train_step`, `val_step` and `test_step`.

- `train_step`: This function is responsible to update the parameters of the network and called by MMEngine's Loop (`IterBasedTrainLoop` or `EpochBasedTrainLoop`). `train_step` take data batch and `OptimWrapper` as input and return a dict of log.
- `val_step`: This function is responsible for getting output for validation during the training process. and is called by `GenValLoop`.
- `test_step`: This function is responsible for getting output in test process and is called by `GenTestLoop`.

  Note that, in `train_step`, `val_step` and `test_step`, `DataPreprocessor` is called to preprocess the input data batch before feed them to the neural network. To know more about `DataPreprocessor` please refer to this file and this tutorial.

For simplify using, we provide `BaseGAN` class in MMEditing, which implements generic `train_step`, `val_step` and `test_step` function for GAN models. With `BaseGAN` as base class, each specific GAN algorithm only need to implement `train_generator` and `train_discriminator`.

In `train_step`, we support data preprocessing, gradient accumulation (realized by `OptimWrapper`) and expontial moving averate (EMA) realized by (`ExponentialMovingAverage`). With `BaseGAN.train_step`, each specific GAN algorithm only need to implement `train_generator` and `train_discriminator`.

```
    def train_step(self, data: dict,
                   optim_wrapper: OptimWrapperDict) -> Dict[str, Tensor]:
        message_hub = MessageHub.get_current_instance()
        curr_iter = message_hub.get_info('iter')
        data = self.data_preprocessor(data, True)
        disc_optimizer_wrapper: OptimWrapper = optim_wrapper['discriminator']
        disc_accu_iters = disc_optimizer_wrapper._accumulative_counts

        # train discriminator, use context manager provided by MMEngine
        with disc_optimizer_wrapper.optim_context(self.discriminator):
            # train_discriminator should be implemented!
            log_vars = self.train_discriminator(
                **data, optimizer_wrapper=disc_optimizer_wrapper)
```

(continues on next page)

(continued from previous page)

```
        # add 1 to `curr_iter` because iter is updated in train loop.
        # Whether to update the generator. We update generator with
        # discriminator is fully updated for `self.n_discriminator_steps`
        # iterations. And one full updating for discriminator contains
        # `disc_accu_counts` times of grad accumulations.
        if (curr_iter + 1) % (self.discriminator_steps * disc_accu_iters) == 0:
            set_requires_grad(self.discriminator, False)
            gen_optimizer_wrapper = optim_wrapper['generator']
            gen_accu_iters = gen_optimizer_wrapper._accumulative_counts

            log_vars_gen_list = []
            # init optimizer wrapper status for generator manually
            gen_optimizer_wrapper.initialize_count_status(
                self.generator, 0, self.generator_steps * gen_accu_iters)
            # update generator, use context manager provided by MMEngine
            for _ in range(self.generator_steps * gen_accu_iters):
                with gen_optimizer_wrapper.optim_context(self.generator):
                    # train_generator should be implemented!
                    log_vars_gen = self.train_generator(
                        **data, optimizer_wrapper=gen_optimizer_wrapper)

                log_vars_gen_list.append(log_vars_gen)
            log_vars_gen = gather_log_vars(log_vars_gen_list)
            log_vars_gen.pop('loss', None)  # remove 'loss' from gen logs

            set_requires_grad(self.discriminator, True)

            # only do ema after generator update
            if self.with_ema_gen and (curr_iter + 1) >= (
                    self.ema_start * self.discriminator_steps *
                    disc_accu_iters):
                self.generator_ema.update_parameters(
                    self.generator.module
                    if is_model_wrapper(self.generator) else self.generator)

            log_vars.update(log_vars_gen)

        # return the log dict
        return log_vars
```

In `val_step` and `test_step`, we call data preprocessing and `BaseGAN.forward` progressively.

```
    def val_step(self, data: dict) -> SampleList:
        data = self.data_preprocessor(data)
        # call `forward`
        outputs = self(**data)
        return outputs

    def test_step(self, data: dict) -> SampleList:
        data = self.data_preprocessor(data)
        # call `orward`
        outputs = self(**data)
```

(continues on next page)

(continued from previous page)

```
        return outputs
```

Then, we implement `train_generator` and `train_discriminator` in `DCGAN` class.

```
from typing import Dict, Tuple

import torch
import torch.nn.functional as F
from mmengine.optim import OptimWrapper
from torch import Tensor

from mmedit.registry import MODELS
from .base_gan import BaseGAN


@MODELS.register_module()
class DCGAN(BaseGAN):
    def disc_loss(self, disc_pred_fake: Tensor,
                  disc_pred_real: Tensor) -> Tuple:
        losses_dict = dict()
        losses_dict['loss_disc_fake'] = F.binary_cross_entropy_with_logits(
            disc_pred_fake, 0. * torch.ones_like(disc_pred_fake))
        losses_dict['loss_disc_real'] = F.binary_cross_entropy_with_logits(
            disc_pred_real, 1. * torch.ones_like(disc_pred_real))

        loss, log_var = self.parse_losses(losses_dict)
        return loss, log_var

    def gen_loss(self, disc_pred_fake: Tensor) -> Tuple:
        losses_dict = dict()
        losses_dict['loss_gen'] = F.binary_cross_entropy_with_logits(
            disc_pred_fake, 1. * torch.ones_like(disc_pred_fake))
        loss, log_var = self.parse_losses(losses_dict)
        return loss, log_var

    def train_discriminator(
            self, inputs, data_sample,
            optimizer_wrapper: OptimWrapper) -> Dict[str, Tensor]:
        real_imgs = inputs['img']

        num_batches = real_imgs.shape[0]

        noise_batch = self.noise_fn(num_batches=num_batches)
        with torch.no_grad():
            fake_imgs = self.generator(noise=noise_batch, return_noise=False)

        disc_pred_fake = self.discriminator(fake_imgs)
        disc_pred_real = self.discriminator(real_imgs)

        parsed_losses, log_vars = self.disc_loss(disc_pred_fake,
                                                 disc_pred_real)
        optimizer_wrapper.update_params(parsed_losses)
```

(continues on next page)

(continued from previous page)

```
        return log_vars

    def train_generator(self, inputs, data_sample,
                        optimizer_wrapper: OptimWrapper) -> Dict[str, Tensor]:
        num_batches = inputs['img'].shape[0]

        noise = self.noise_fn(num_batches=num_batches)
        fake_imgs = self.generator(noise=noise, return_noise=False)

        disc_pred_fake = self.discriminator(fake_imgs)
        parsed_loss, log_vars = self.gen_loss(disc_pred_fake)

        optimizer_wrapper.update_params(parsed_loss)
        return log_vars
```

After the implementation of `class DCGAN`, we need to update the model list in `mmedit/models/__init__.py`, so that we can import and use `class DCGAN` by `mmedit.models`.

Step 3: Start training DCGAN

After implementing the network **Module** and the **Model** of DCGAN, now we can create a new file `configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py` to set the configurations needed by training DCGAN.

In the configuration file, we need to specify the parameters of our model, `class DCGAN`, including the generator network architecture and data preprocessor of input tensors.

```
# model settings
model = dict(
    type='DCGAN',
    noise_size=100,
    data_preprocessor=dict(type='GANDataPreprocessor'),
    generator=dict(type='DCGANGenerator', output_scale=64, base_channels=1024),
    discriminator=dict(
        type='DCGANDiscriminator',
        input_scale=64,
        output_scale=4,
        out_channels=1))
```

We also need to specify the training dataloader and testing dataloader according to *create your own dataloader*. Finally we can start training our own model by

```
python train.py configs/dcgan/dcgan_1xb128-5epoches_lsun-bedroom-64x64.py
```

1.18.4 References

1. Dong, Chao and Loy, Chen Change and He, Kaiming and Tang, Xiaoou. Image Super-Resolution Using Deep Convolutional Networks[J]. IEEE transactions on pattern analysis and machine intelligence, 2015.
2. Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

1.19 How to prepare your own datasets

In this document, we will introduce the design of each datasets in MMEditing and how users can design their own dataset.

1.19.1 Supported Data Format

In 1.x version of MMEditing, all datasets are inherited from `BaseDataset`. Each dataset load the list of data info (e.g., data path) by `load_data_list`. In `__getitem__`, `prepare_data` is called to get the preprocessed data. In `prepare_data`, data loading pipeline consists of the following steps:

1. fetch the data info by passed index, implemented by `get_data_info`
2. apply data transforms to the data, implemented by `pipeline`

BasicImageDataset

BasicImageDataset `mmedit.datasets.BasicImageDataset` General image dataset designed for low-level vision tasks with image, such as image super-resolution, inpainting and unconditional image generation. The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
Case 1 (CelebA-HQ):

    000001.png
    000002.png
```

(continues on next page)

(continued from previous page)

```
    Case 2 (DIV2K):

        0001_s001.png (480,480,3)
        0001_s002.png (480,480,3)
        0001_s003.png (480,480,3)
        0002_s001.png (480,480,3)
        0002_s002.png (480,480,3)

    Case 3 (Vimeo90k):

        00001/0266 (256, 448, 3)
        00001/0268 (256, 448, 3)
```

Here we give several examples showing how to use `BasicImageDataset`. Assume the file structure as the following:

```
mmediting (root)
├── mmedit
├── tools
├── configs
├── data
    ├── DIV2K
        ├── DIV2K_train_HR
            ├── image.png
        ├── DIV2K_train_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
                ├── image_x4.png
        ├── DIV2K_valid_HR
        ├── DIV2K_valid_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
    ├── places
        ├── test_set
        ├── train_set
        ├── meta
            ├── Places365_train.txt
            ├── Places365_val.txt
    ├── celebahq
        ├── imgs_1024
```

Case 1: Loading DIV2K dataset for training a SISR model.

```
    dataset = BasicImageDataset(
        ann_file='',
        metainfo=dict(
            dataset_type='div2k',
            task_name='sisr'),
        data_root='data/DIV2K',
        data_prefix=dict(
            gt='DIV2K_train_HR', img='DIV2K_train_LR_bicubic/X4'),
```

(continues on next page)

(continued from previous page)

```
        filename_tmpl=dict(img='{}_x4', gt='{}'),
        pipeline=[])
```

Case 2: Loading places dataset for training an inpainting model.

```
    dataset = BasicImageDataset(
        ann_file='meta/Places365_train.txt',
        metainfo=dict(
            dataset_type='places365',
            task_name='inpainting'),
        data_root='data/places',
        data_prefix=dict(gt='train_set'),
        pipeline=[])
```

Case 3: Loading CelebA-HQ dataset for training an PGGAN.

```
dataset = BasicImageDataset(
        pipeline=[],
        data_root='./data/celebahq/imgs_1024')
```

BasicFramesDataset

BasicFramesDataset `mmedit.datasets.BasicFramesDataset` General frames dataset designed for low-level vision tasks with frames, such as video super-resolution and video frame interpolation. The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
Case 1 (Vid4):

   calendar 41
   city 34
   foliage 49
   walk 47

Case 2 (REDS):

   000/00000000.png (720, 1280, 3)
   000/00000001.png (720, 1280, 3)

Case 3 (Vimeo90k):

   00001/0266 (256, 448, 3)
   00001/0268 (256, 448, 3)
```

Assume the file structure as the following:

```
mmediting (root)
├── mmedit
├── tools
├── configs
├── data
    ├── Vid4
```

(continues on next page)

(continued from previous page)

```
|   |   ├── BIx4
|   |   |   ├── city
|   |   |   |   ├── img1.png
|   |   ├── GT
|   |   |   ├── city
|   |   |   |   ├── img1.png
|   |   ├── meta_info_Vid4_GT.txt
|   ├── places
|   |   ├── sequences
|   |   |   ├── 00001
|   |   |   |   ├── 0389
|   |   |   |   |   ├── img1.png
|   |   |   |   |   ├── img2.png
|   |   |   |   |   ├── img3.png
|   |   ├── tri_trainlist.txt
```

Case 1: Loading Vid4 dataset for training a VSR model.

```
dataset = BasicFramesDataset(
    ann_file='meta_info_Vid4_GT.txt',
    metainfo=dict(dataset_type='vid4', task_name='vsr'),
    data_root='data/Vid4',
    data_prefix=dict(img='BIx4', gt='GT'),
    pipeline=[],
    depth=2,
    num_input_frames=5)
```

Case 2: Loading Vimeo90k dataset for training a VFI model.

```
dataset = BasicFramesDataset(
    ann_file='tri_trainlist.txt',
    metainfo=dict(dataset_type='vimeo90k', task_name='vfi'),
    data_root='data/vimeo-triplet',
    data_prefix=dict(img='sequences', gt='sequences'),
    pipeline=[],
    depth=2,
    load_frames_list=dict(
        img=['img1.png', 'img3.png'], gt=['img2.png']))
```

BasicConditonalDataset

BasicConditonalDataset `mmedit.datasets.BasicConditonalDataset` is designed for conditional GANs (e.g., SAGAN, BigGAN). This dataset support load label for the annotation file. `BasicConditonalDataset` support three kinds of annotation as follow:

1. Annotation file read by line (e.g., txt)

Sample files structure:

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

Sample annotation file (the first column is the image path and the second column is the index of category):

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

Config example for ImageNet dataset:

```
dataset=dict(
    type='BasicConditionalDataset,
    data_root='./data/imagenet/',
    ann_file='meta/train.txt',
    data_prefix='train',
    pipeline=train_pipeline),
```

2. Dict-based annotation file (e.g., json):

Sample files structure:

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

Sample annotation file (the key is the image path and the value column is the label):

```
    {
        "folder_1/xxx.png": [1, 2, 3, 4],
        "folder_1/xxy.png": [2, 4, 1, 0],
        "folder_2/123.png": [0, 9, 8, 1],
        "folder_2/nsdf3.png", [1, 0, 0, 2],
        ...
    }
```

Config example for EG3D (shapenet-car) dataset:

```
dataset = dict(
    type='BasicConditionalDataset',
    data_root='./data/eg3d/shapenet-car',
    ann_file='annotation.json',
    pipeline=train_pipeline)
```

In this kind of annotation, labels can be any type and not restricted to an index.

3. Folder-based annotation (no annotation file need):

Sample files structure:

```
    data_prefix/
    ├── class_x
    │   ├── xxx.png
    │   ├── xxy.png
    │   └── ...
    │       └── xxz.png
    └── class_y
        ├── 123.png
        ├── nsdf3.png
        ├── ...
        └── asd932_.png
```

If the annotation file is specified, the dataset will be generated by the first two ways, otherwise, try the third way.

ImageNet Dataset and CIFAR10 Dataset

ImageNet Dataset`mmedit.datasets.ImageNet` and **CIFAR10 Dataset**`mmedit.datasets.CIFAR10` are datasets specific designed for ImageNet and CIFAR10 datasets. Both two datasets are encapsulation of `BasicConditionalDataset`. You can used them to load data from ImageNet dataset and CIFAR10 dataset easily.

Config example for ImageNet:

```
pipeline = [
    dict(type='LoadImageFromFile', key='img'),
    dict(type='RandomCropLongEdge', keys=['img']),
    dict(type='Resize', scale=(128, 128), keys=['img'], backend='pillow'),
    dict(type='Flip', keys=['img'], flip_ratio=0.5, direction='horizontal'),
    dict(type='PackEditInputs')
]
```

(continues on next page)

(continued from previous page)

```
dataset=dict(
    type='ImageNet',
    data_root='./data/imagenet/',
    ann_file='meta/train.txt',
    data_prefix='train',
    pipeline=pipeline),
```

Config example for CIFAR10:

```
pipeline = [dict(type='PackEditInputs')]

dataset = dict(
    type='CIFAR10',
    data_root='./data',
    data_prefix='cifar10',
    test_mode=False,
    pipeline=pipeline)
```

AdobeComp1kDataset

AdobeComp1kDataset `mmedit.datasets.AdobeComp1kDataset` Adobe composition-1k dataset.

The dataset loads (alpha, fg, bg) data and apply specified transforms to the data. You could specify whether composite merged image online or load composited merged image in pipeline.

Example for online comp-1k dataset:

```
[
   {
       "alpha": 'alpha/000.png',
       "fg": 'fg/000.png',
       "bg": 'bg/000.png'
   },
   {
       "alpha": 'alpha/001.png',
       "fg": 'fg/001.png',
       "bg": 'bg/001.png'
   },
]
```

Example for offline comp-1k dataset:

```
[
  {
      "alpha": 'alpha/000.png',
      "merged": 'merged/000.png',
      "fg": 'fg/000.png',
      "bg": 'bg/000.png'
  },
  {
      "alpha": 'alpha/001.png',
```

(continues on next page)

(continued from previous page)

```
      "merged": 'merged/001.png',
      "fg": 'fg/001.png',
      "bg": 'bg/001.png'
  },
]
```

GrowScaleImgDataset

`GrowScaleImgDataset` is designed for dynamic GAN models (e.g., PGGAN and StyleGANv1). In this dataset, we support switching the data root during training to load training images of different resolutions. This procedure is implemented by `GrowScaleImgDataset.update_annotations` and is called by `PGGANFetchDataHook.before_train_iter` in the training process.

```
def update_annotations(self, curr_scale):
    # determine if the data root needs to be updated
    if curr_scale == self._actual_curr_scale:
        return False

    # fetch new data root by resolution (scale)
    for scale in self._img_scales:
        if curr_scale <= scale:
            self._curr_scale = scale
            break
        if scale == self._img_scales[-1]:
            assert RuntimeError(
                f'Cannot find a suitable scale for {curr_scale}')
    self._actual_curr_scale = curr_scale
    self.data_root = self.data_roots[str(self._curr_scale)]

    # reload the data list with new data root
    self.load_data_list()

    # print basic dataset information to check the validity
    print_log('Update Dataset: ' + repr(self), 'current')
    return True
```

SinGANDataset

`SinGANDataset` is designed for SinGAN's training. In SinGAN's training, we do not iterate the images in the dataset but return a consistent preprocessed image dict.

Therefore, we bypass the default data loading logic of `BaseDataset` because we do not need to load the corresponding image data based on the given index.

```
def load_data_list(self, min_size, max_size, scale_factor_init):
    # load single image
    real = mmcv.imread(self.data_root)
    self.reals, self.scale_factor, self.stop_scale = create_real_pyramid(
        real, min_size, max_size, scale_factor_init)
```

(continues on next page)

(continued from previous page)

```
    self.data_dict = {}

    # generate multi scale image
    for i, real in enumerate(self.reals):
        self.data_dict[f'real_scale{i}'] = real

    self.data_dict['input_sample'] = np.zeros_like(
        self.data_dict['real_scale0']).astype(np.float32)

def __getitem__(self, index):
    # directly return the transformed data dict
    return self.pipeline(self.data_dict)
```

PairedImageDataset

`PairedImageDataset` is designed for translation models that needs paried training data (e.g., Pix2Pix). The directory structure is shown below. Each image files are the concatenation of the image pair.

```
./data/dataset_name/
├── test
│   └── XXX.jpg
└── train
    └── XXX.jpg
```

In `PairedImageDataset`, we scan the file list in `load_data_list` and save path in `pair_path` field to fit the `LoadPairedImageFromFile` transformation.

```
def load_data_list(self):
    data_infos = []
    pair_paths = sorted(self.scan_folder(self.data_root))
    for pair_path in pair_paths:
        # save path in the specific field
        data_infos.append(dict(pair_path=pair_path))

    return data_infos
```

UnpairedImageDataset

`UnpairedImageDataset` is designed for translation models that do not need paired data (e.g., CycleGAN). The directory structure is shown below.

```
./data/dataset_name/
├── testA
│   └── XXX.jpg
├── testB
│   └── XXX.jpg
├── trainA
│   └── XXX.jpg
└── trainB
    └── XXX.jpg
```

In this dataset, we overwrite `__getitem__` function to load random image pair in the training process.

```
def __getitem__(self, idx):
    if not self.test_mode:
        return self.prepare_train_data(idx)

    return self.prepare_test_data(idx)

def prepare_train_data(self, idx):
    img_a_path = self.data_infos_a[idx % self.len_a]['path']
    idx_b = np.random.randint(0, self.len_b)
    img_b_path = self.data_infos_b[idx_b]['path']
    results = dict()
    results[f'img_{self.domain_a}_path'] = img_a_path
    results[f'img_{self.domain_b}_path'] = img_b_path
    return self.pipeline(results)

def prepare_test_data(self, idx):
    img_a_path = self.data_infos_a[idx % self.len_a]['path']
    img_b_path = self.data_infos_b[idx % self.len_b]['path']
    results = dict()
    results[f'img_{self.domain_a}_path'] = img_a_path
    results[f'img_{self.domain_b}_path'] = img_b_path
    return self.pipeline(results)
```

1.19.2 Design a new dataset

If you want to create a dataset for a new low level CV task (e.g. denoise, derain, defog, and de-reflection) or existing dataset format doesn't meet your need, you can reorganize new data formats to existing format.

Or create a new dataset in `mmedit/datasets` to load the data.

Inheriting from the base class of datasets such as `BasicImageDataset` and `BasicFramesDataset` will make it easier to create a new dataset.

And you can create a new dataset inherited from BaseDataset which is the base class of datasets in MMEngine.

Here is an example of creating a dataset for video frame interpolation:

```
from .basic_frames_dataset import BasicFramesDataset
from mmedit.registry import DATASETS


@DATASETS.register_module()
class NewVFIDataset(BasicFramesDataset):
    """Introduce the dataset

    Examples of file structure.

    Args:
        pipeline (list[dict | callable]): A sequence of data transformations.
        folder (str | :obj:`Path`): Path to the folder.
        ann_file (str | :obj:`Path`): Path to the annotation file.
        test_mode (bool): Store `True` when building test dataset.
```

(continues on next page)

(continued from previous page)

```
            Default: `False`.
    """

    def __init__(self, ann_file, metainfo, data_root, data_prefix,
                 pipeline, test_mode=False):
        super().__init__(ann_file, metainfo, data_root, data_prefix,
                         pipeline, test_mode)
        self.data_infos = self.load_annotations()

    def load_annotations(self):
        """Load annoations for the dataset.

        Returns:
            list[dict]: A list of dicts for paired paths and other information.
        """
        data_infos = []
        ...
        return data_infos
```

Welcome to submit new dataset classes to MMEditing.

Repeat dataset

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

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

You may refer to tutorial in MMEngine.

1.20 How to design your own data transforms

In this tutorial, we introduce the design of transforms pipeline in MMEditing.

The structure of this guide are as follows:

- Design Your Own Data Pipelines
  - *Data pipelines in MMEditing*
    - *A simple example of data transform*
    - *An example of BasicVSR*
    - *An example of Pix2Pix*

1.20.1 Data pipelines in MMEditing

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

The data preparation pipeline and the dataset is decomposed. Usually a dataset defines how to process the annotations and a data pipeline defines all the steps to prepare a data dict.

A pipeline consists of a sequence of operations. Each operation takes a dict as input and also output a dict for the next transform.

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

In 1.x version of MMEditing, all data transformations are inherited from `BaseTransform`. The input and output types of transformations are both dict.

A simple example of data transform

```
>>> from mmgen.transforms import LoadPairedImageFromFile
>>> transforms = LoadPairedImageFromFile(
>>>     key='pair',
>>>     domain_a='horse',
>>>     domain_b='zebra',
>>>     flag='color'),
>>> data_dict = {'pair_path': './data/pix2pix/facades/train/1.png'}
>>> data_dict = transforms(data_dict)
>>> print(data_dict.keys())
dict_keys(['pair_path', 'pair', 'pair_ori_shape', 'img_mask', 'img_photo', 'img_mask_path
↪', 'img_photo_path', 'img_mask_ori_shape', 'img_photo_ori_shape'])
```

Generally, the last step of the transforms pipeline must be `PackEditInputs`. `PackEditInputs` will pack the processed data into a dict containing two fields: `inputs` and `data_samples`. `inputs` is the variable you want to use as the model's input, which can be the type of `torch.Tensor`, dict of `torch.Tensor`, or any type you want. `data_samples` is a list of `EditDataSample`. Each `EditDataSample` contains groundtruth and necessary information for corresponding input.

An example of BasicVSR

Here is a pipeline example for BasicVSR.

```
train_pipeline = [
    dict(type='LoadImageFromFile', key='img', channel_order='rgb'),
    dict(type='LoadImageFromFile', key='gt', channel_order='rgb'),
    dict(type='SetValues', dictionary=dict(scale=scale)),
    dict(type='PairedRandomCrop', gt_patch_size=256),
    dict(
        type='Flip',
        keys=['img', 'gt'],
        flip_ratio=0.5,
        direction='horizontal'),
    dict(
        type='Flip', keys=['img', 'gt'], flip_ratio=0.5, direction='vertical'),
    dict(type='RandomTransposeHW', keys=['img', 'gt'], transpose_ratio=0.5),
    dict(type='MirrorSequence', keys=['img', 'gt']),
    dict(type='PackEditInputs')
]

val_pipeline = [
    dict(type='GenerateSegmentIndices', interval_list=[1]),
    dict(type='LoadImageFromFile', key='img', channel_order='rgb'),
    dict(type='LoadImageFromFile', key='gt', channel_order='rgb'),
    dict(type='PackEditInputs')
]

test_pipeline = [
    dict(type='LoadImageFromFile', key='img', channel_order='rgb'),
    dict(type='LoadImageFromFile', key='gt', channel_order='rgb'),
    dict(type='MirrorSequence', keys=['img']),
    dict(type='PackEditInputs')
]
```

For each operation, we list the related dict fields that are added/updated/removed, the dict fields marked by '*' are optional.

An example of Pix2Pix

Here is a pipeline example for Pix2Pix training on aerial2maps dataset.

```
source_domain = 'aerial'
target_domain = 'map'

pipeline = [
    dict(
        type='LoadPairedImageFromFile',
        io_backend='disk',
        key='pair',
        domain_a=domain_a,
        domain_b=domain_b,
        flag='color'),
```

(continues on next page)

(continued from previous page)

```
    dict(
        type='TransformBroadcaster',
        mapping={'img': [f'img_{domain_a}', f'img_{domain_b}']},
        auto_remap=True,
        share_random_params=True,
        transforms=[
            dict(
                type='mmgen.Resize', scale=(286, 286),
                interpolation='bicubic'),
            dict(type='mmgen.FixedCrop', crop_size=(256, 256))
        ]),
    dict(
        type='Flip',
        keys=[f'img_{domain_a}', f'img_{domain_b}'],
        direction='horizontal'),
    dict(
        type='PackEditInputs',
        keys=[f'img_{domain_a}', f'img_{domain_b}', 'pair'])
```

1.20.2 Supported transforms in MMEditing

Data loading

Pre-processing

Formatting

1.20.3 Extend and use custom pipelines

A simple example of MyTransform

1. Write a new pipeline in a file, e.g., in `my_pipeline.py`. It takes a dict as input and returns a dict.

```
import random
from mmcv.transforms import BaseTransform
from mmedit.registry import TRANSFORMS


@TRANSFORMS.register_module()
class MyTransform(BaseTransform):
    """Add your transform

    Args:
        p (float): Probability of shifts. Default 0.5.
    """

    def __init__(self, p=0.5):
        self.p = p

    def transform(self, results):
```

(continues on next page)

(continued from previous page)

```
        if random.random() > self.p:
            results['dummy'] = True
        return results

    def __repr__(self):

        repr_str = self.__class__.__name__
        repr_str += (f'(p={self.p})')

        return repr_str
```

2. Import and use the pipeline in your config file.

Make sure the import is relative to where your train script is located.

```
train_pipeline = [
    ...
    dict(type='MyTransform', p=0.2),
    ...
]
```

An example of flipping

Here we use a simple flipping transformation as example:

```
import random
import mmcv
from mmcv.transforms import BaseTransform, TRANSFORMS

@TRANSFORMS.register_module()
class MyFlip(BaseTransform):
    def __init__(self, direction: str):
        super().__init__()
        self.direction = direction

    def transform(self, results: dict) -> dict:
        img = results['img']
        results['img'] = mmcv.imflip(img, direction=self.direction)
        return results
```

Thus, we can instantiate a `MyFlip` object and use it to process the data dict.

```
import numpy as np

transform = MyFlip(direction='horizontal')
data_dict = {'img': np.random.rand(224, 224, 3)}
data_dict = transform(data_dict)
processed_img = data_dict['img']
```

Or, we can use `MyFlip` transformation in data pipeline in our config file.

```
pipeline = [
    ...
    dict(type='MyFlip', direction='horizontal'),
    ...
]
```

Note that if you want to use `MyFlip` in config, you must ensure the file containing `MyFlip` is imported during the program run.

1.21 How to design your own loss functions

`losses` are registered as `LOSSES` in `MMEditing`. Customizing losses is similar to customizing any other model. This section is mainly for clarifying the design of loss modules in MMEditing. Importantly, when writing your own loss modules, you should follow the same design, so that the new loss module can be adopted in our framework without extra effort.

This guides includes:

- Design Your Own Loss Functions
  - *Introduction to supported losses*
  - *Design a new loss function*
    - *An example of MSELoss*
    - *An example of DiscShiftLoss*
    - *An example of GANWithCustomizedLoss*
  - *Available losses*
    - *regular losses*
    - *losses components*

1.21.1 Introduction to supported losses

For convenient usage, you can directly use default loss calculation process we set for concrete algorithms like lsgan, biggan, styleganv2 etc. Take `stylegan2` as an example, we use R1 gradient penalty and generator path length regularization as configurable losses, and users can adjust related arguments like `r1_loss_weight` and `g_reg_weight`.

```
# stylegan2_base.py
loss_config = dict(
    r1_loss_weight=10. / 2. * d_reg_interval,
    r1_interval=d_reg_interval,
    norm_mode='HWC',
    g_reg_interval=g_reg_interval,
    g_reg_weight=2. * g_reg_interval,
    pl_batch_shrink=2)

model = dict(
    type='StyleGAN2',
    xxx,
    loss_config=loss_config)
```

1.21.2 Design a new loss function

An example of MSELoss

In general, to implement a loss module, we will write a function implementation and then wrap it with a class implementation. Take the MSELoss as an example:

```
@masked_loss
def mse_loss(pred, target):
    return F.mse_loss(pred, target, reduction='none')

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

    def __init__(self, loss_weight=1.0, reduction='mean', sample_wise=False):
        # codes can be found in ``mmedit/models/losses/pixelwise_loss.py``

    def forward(self, pred, target, weight=None, **kwargs):
        # codes can be found in ``mmedit/models/losses/pixelwise_loss.py``
```

Given the definition of the loss, we can now use the loss by simply defining it in the configuration file:

```
pixel_loss=dict(type='MSELoss', loss_weight=1.0, reduction='mean')
```

Note that `pixel_loss` above must be defined in the model. Please refer to `customize_models` for more details. Similar to model customization, in order to use your customized loss, you need to import the loss in `mmedit/models/losses/__init__.py` after writing it.

An example of DiscShiftLoss

In general, to implement a loss module, we will write a function implementation and then wrap it with a class implementation. However, in `MMEditing`, we provide another unified interface `data_info` for users to define the mapping between the input argument and data items.

```
@weighted_loss
def disc_shift_loss(pred):
    return pred**2

@MODULES.register_module()
class DiscShiftLoss(nn.Module):

    def __init__(self, loss_weight=1.0, data_info=None):
        super(DiscShiftLoss, self).__init__()
        # codes can be found in ``mmgen/models/losses/disc_auxiliary_loss.py``

    def forward(self, *args, **kwargs):
        # codes can be found in ``mmgen/models/losses/disc_auxiliary_loss.py``
```

The goal of this design for loss modules is to allow for using it automatically in the generative models (`MODELS`), without other complex codes to define the mapping between data and keyword arguments. Thus, different from other frameworks in `OpenMMLab`, our loss modules contain a special keyword, `data_info`, which is a dictionary defining the mapping between the input arguments and data from the generative models. Taking the `DiscShiftLoss` as an example, when writing the config file, users may use this loss as follows:

```
dict(type='DiscShiftLoss',
    loss_weight=0.001 * 0.5,
    data_info=dict(pred='disc_pred_real')
```

The information in `data_info` tells the module to use the `disc_pred_real` data as the input tensor for `pred` arguments. Once the `data_info` is not `None`, our loss module will automatically build up the computational graph.

```
@MODULES.register_module()
class DiscShiftLoss(nn.Module):

    def __init__(self, loss_weight=1.0, data_info=None):
        super(DiscShiftLoss, self).__init__()
        self.loss_weight = loss_weight
        self.data_info = data_info

    def forward(self, *args, **kwargs):
        # use data_info to build computational path
        if self.data_info is not None:
            # parse the args and kwargs
            if len(args) == 1:
                assert isinstance(args[0], dict), (
                    'You should offer a dictionary containing network outputs '
                    'for building up computational graph of this loss module.')
                outputs_dict = args[0]
            elif 'outputs_dict' in kwargs:
                assert len(args) == 0, (
                    'If the outputs dict is given in keyworded arguments, no'
                    ' further non-keyworded arguments should be offered.')
                outputs_dict = kwargs.pop('outputs_dict')
            else:
                raise NotImplementedError(
                    'Cannot parsing your arguments passed to this loss module.'
                    ' Please check the usage of this module')
            # link the outputs with loss input args according to self.data_info
            loss_input_dict = {
                k: outputs_dict[v]
                for k, v in self.data_info.items()
            }
            kwargs.update(loss_input_dict)
            kwargs.update(dict(weight=self.loss_weight))
            return disc_shift_loss(**kwargs)
        else:
            # if you have not define how to build computational graph, this
            # module will just directly return the loss as usual.
            return disc_shift_loss(*args, weight=self.loss_weight, **kwargs)

    @staticmethod
    def loss_name():
        return 'loss_disc_shift'
```

As shown in this part of codes, once users set the `data_info`, the loss module will receive a dictionary containing all of the necessary data and modules, which is provided by the `MODELS` in the training procedure. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using a keyword argument,

please name it as `outputs_dict`.

An example of GANWithCustomizedLoss

To build the computational graph, the generative models have to provide a dictionary containing all kinds of data. Having a close look at any generative model, you will find that we collect all kinds of features and modules into a dictionary. We provide a customized `GANWithCustomizedLoss` here to show the process.

```
class GANWithCustomizedLoss(BaseModel):

    def __init__(self, gan_loss, disc_auxiliary_loss, gen_auxiliary_loss,
                 *args, **kwargs):
        # ...
        if gan_loss is not None:
            self.gan_loss = MODULES.build(gan_loss)
        else:
            self.gan_loss = None

        if disc_auxiliary_loss:
            self.disc_auxiliary_losses = MODULES.build(disc_auxiliary_loss)
            if not isinstance(self.disc_auxiliary_losses, nn.ModuleList):
                self.disc_auxiliary_losses = nn.ModuleList(
                    [self.disc_auxiliary_losses])
        else:
            self.disc_auxiliary_loss = None

        if gen_auxiliary_loss:
            self.gen_auxiliary_losses = MODULES.build(gen_auxiliary_loss)
            if not isinstance(self.gen_auxiliary_losses, nn.ModuleList):
                self.gen_auxiliary_losses = nn.ModuleList(
                    [self.gen_auxiliary_losses])
        else:
            self.gen_auxiliary_losses = None

    def train_step(self, data: dict,
                   optim_wrapper: OptimWrapperDict) -> Dict[str, Tensor]:
        # ...

        # get data dict to compute losses for disc
        data_dict_ = dict(
            iteration=curr_iter,
            gen=self.generator,
            disc=self.discriminator,
            disc_pred_fake=disc_pred_fake,
            disc_pred_real=disc_pred_real,
            fake_imgs=fake_imgs,
            real_imgs=real_imgs)

        loss_disc, log_vars_disc = self._get_disc_loss(data_dict_)

        # ...

    def _get_disc_loss(self, outputs_dict):
```

(continues on next page)

(continued from previous page)

```
        # Construct losses dict. If you hope some items to be included in the
        # computational graph, you have to add 'loss' in its name. Otherwise,
        # items without 'loss' in their name will just be used to print
        # information.
        losses_dict = {}
        # gan loss
        losses_dict['loss_disc_fake'] = self.gan_loss(
            outputs_dict['disc_pred_fake'], target_is_real=False, is_disc=True)
        losses_dict['loss_disc_real'] = self.gan_loss(
            outputs_dict['disc_pred_real'], target_is_real=True, is_disc=True)

        # disc auxiliary loss
        if self.with_disc_auxiliary_loss:
            for loss_module in self.disc_auxiliary_losses:
                loss_ = loss_module(outputs_dict)
                if loss_ is None:
                    continue

                # the `loss_name()` function return name as 'loss_xxx'
                if loss_module.loss_name() in losses_dict:
                    losses_dict[loss_module.loss_name(
                    )] = losses_dict[loss_module.loss_name()] + loss_
                else:
                    losses_dict[loss_module.loss_name()] = loss_
        loss, log_var = self.parse_losses(losses_dict)

        return loss, log_var
```

Here, the `_get_disc_loss` will help to combine all kinds of losses automatically.

Therefore, as long as users design the loss module with the same rules, any kind of loss can be inserted in the training of generative models, without other modifications in the code of models. What you only need to do is just defining the `data_info` in the config files.

1.21.3 Available losses

We list available losses with examples in configs as follows.

regular losses

```
# dic gan
loss_gan=dict(
    type='GANLoss',
    gan_type='vanilla',
    loss_weight=0.001,
)
```

```
# deepfillv1
loss_gan=dict(
    type='GANLoss',
    gan_type='wgan',
    loss_weight=0.0001,
)
```

```
# deepfillv2
loss_gan=dict(
    type='GANLoss',
    gan_type='hinge',
    loss_weight=0.1,
)
```

```
# aot-gan
loss_gan=dict(
    type='GANLoss',
    gan_type='smgan',
    loss_weight=0.01,
)
```

```
# deepfillv1
loss_gp=dict(type='GradientPenaltyLoss', loss_weight=10.)
```

```
# deepfillv1
loss_disc_shift=dict(type='DiscShiftLoss', loss_weight=0.001)
```

```
# dim
loss_comp=dict(type='CharbonnierCompLoss', loss_weight=0.5)
```

```
# dic gan
feature_loss=dict(
    type='LightCNNFeatureLoss',
    pretrained=pretrained_light_cnn,
    loss_weight=0.1,
    criterion='l1')
```

```
# dic gan
pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')
```

```
# dic gan
align_loss=dict(type='MSELoss', loss_weight=0.1, reduction='mean')
```

```
# dim
loss_alpha=dict(type='CharbonnierLoss', loss_weight=0.5)
```

```
# partial conv
loss_tv=dict(
    type='MaskedTVLoss',
```

(continues on next page)

(continued from previous page)

```
    loss_weight=0.1
)
```

```
# real_basicvsr
perceptual_loss=dict(
    type='PerceptualLoss',
    layer_weights={
        '2': 0.1,
        '7': 0.1,
        '16': 1.0,
        '25': 1.0,
        '34': 1.0,
    },
    vgg_type='vgg19',
    perceptual_weight=1.0,
    style_weight=0,
    norm_img=False)
```

```
# ttsr
transferal_perceptual_loss=dict(
    type='TransferalPerceptualLoss',
    loss_weight=1e-2,
    use_attention=False,
    criterion='mse')
```

losses components

For `GANWithCustomizedLoss`, we provide several components to build customized loss.

1.22 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. If the contents here do not cover your issue, please create an issue using the provided templates and make sure you fill in all required information in the template.

1.22.1 FAQ

Q1: "xxx: 'yyy is not in the zzz registry'".

A1: The registry mechanism will be triggered only when the file of the module is imported. So you need to import that file somewhere.

Q2: What's the folder structure of xxx dataset?

A2: You can make sure the folder structure is correct following tutorials of *dataset preparation*.

Q3: How to use LMDB data to train the model?

A3: You can use scripts in `tools/data` to make LMDB files. More details are shown in tutorials of *dataset preparation*.

Q4: Why `MMCV==xxx is used but incompatible` is raised when import I try to import `mmgen`?

A4: This is because the version of MMCV and MMGeneration are incompatible. Compatible MMGeneration and MMCV versions are shown as below. Please choose the correct version of MMCV to avoid installation issues.

Note: You need to run `pip uninstall mmcv` first if you have mmcv installed. If mmcv and mmcv-full are both installed, there will be `ModuleNotFoundError`.

Q5: How can I ignore some fields in the base configs?

A5: Sometimes, you may set `_delete_=True` to ignore some of fields in base configs. You may refer to MMEngine for simple illustration.

You may have a careful look at this tutorial for better understanding of this feature.

Q6:: How can I use intermediate variables in configs?

A6: Some intermediate variables are used in the config files, like `train_pipeline/test_pipeline` in datasets. It's worth noting that when modifying intermediate variables in the children configs, users need to pass the intermediate variables into corresponding fields again.

1.23 Overview

- *vimeo90k*
- *vid4*
- *unpaired-cyclegan*
- *reds*
- *unconditional_gans*
- *df2k_ost*
- *div2k*
- *comp1k*
- *celeba-hq*
- *places365*
- *paris-street-view*
- *vimeo90k-triplet*
- *paired-pix2pix*

1.24 Preparing Vimeo90K Dataset

```
@article{xue2019video,
  title={Video Enhancement with Task-Oriented Flow},
  author={Xue, Tianfan and Chen, Baian and Wu, Jiajun and Wei, Donglai and Freeman,␣
↪William T},
  journal={International Journal of Computer Vision (IJCV)},
  volume={127},
  number={8},
  pages={1106--1125},
  year={2019},
  publisher={Springer}
}
```

The training and test datasets can be download from here.

The Vimeo90K dataset has a `clip/sequence/img` folder structure:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── vimeo_triplet
        ├── BDx4
            ├── 00001
                ├── 0001
                    ├── im1.png
                    ├── im2.png
                    ├── ...
                ├── 0002
                ├── 0003
                ├── ...
            ├── 00002
            ├── ...
        ├── BIx4
        ├── GT
        ├── meta_info_Vimeo90K_test_GT.txt
        ├── meta_info_Vimeo90K_train_GT.txt
```

1.24.1 Prepare the annotation files for Vimeo90K dataset

To prepare the annotation file for training, you need to download the official training list path for Vimeo90K from the official website, and run the following command:

```
python tools/dataset_converters/super-resolution/vimeo90k/preprocess_vimeo90k_dataset.py␣
↪./data/Vimeo90K/official_train_list.txt
```

The annotation file for test is generated similarly.

1.24.2 Prepare LMDB dataset for Vimeo90K

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/vimeo90k/preprocess_vimeo90k_dataset.py␣
↪./data/Vimeo90K/official_train_list.txt --gt-path ./data/Vimeo90K/GT --lq-path ./data/
↪Vimeo90K/LQ  --make-lmdb
```

1.25 Preparing Vid4 Dataset

```
@article{xue2019video,
  title={On Bayesian adaptive video super resolution},
  author={Liu, Ce and Sun, Deqing},
  journal={IEEE Transactions on Pattern Analysis and Machine Intelligence},
  volume={36},
  number={2},
  pages={346--360},
  year={2013},
  publisher={IEEE}
}
```

The Vid4 dataset can be downloaded from here. There are two degradations in the dataset.

1. BIx4 contains images downsampled by bicubic interpolation
2. BDx4 contains images blurred by Gaussian kernel with =1.6, followed by a subsampling every four pixels.

1.26 Preparing Unpaired Dataset for CycleGAN

```
@inproceedings{zhu2017unpaired,
  title={Unpaired image-to-image translation using cycle-consistent adversarial networks}
↪,
  author={Zhu, Jun-Yan and Park, Taesung and Isola, Phillip and Efros, Alexei A},
  booktitle={Proceedings of the IEEE international conference on computer vision},
  pages={2223--2232},
  year={2017}
}
```

You can download unpaired datasets from here. Then, you need to unzip and move corresponding datasets to follow the folder structure shown above. The datasets have been well-prepared by the original authors.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── unpaired
        ├── facades
|   |   ├── horse2zebra
|   |   ├── summer2winter_yosemite
```

(continues on next page)

(continued from previous page)

```
|   |   |   ├── trainA
|   |   |   ├── trainB
|   |   |   ├── testA
|   |   |   ├── testB
```

1.27 Preparing REDS Dataset

```
@InProceedings{Nah_2019_CVPR_Workshops_REDS,
  author = {Nah, Seungjun and Baik, Sungyong and Hong, Seokil and Moon, Gyeongsik and␣
↪Son, Sanghyun and Timofte, Radu and Lee, Kyoung Mu},
  title = {NTIRE 2019 Challenge on Video Deblurring and Super-Resolution: Dataset and␣
↪Study},
  booktitle = {The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)␣
↪Workshops},
  month = {June},
  year = {2019}
}
```

- Training dataset: REDS dataset.
- Validation dataset: REDS dataset and Vid4.

Note that we merge train and val datasets in REDS for easy switching between REDS4 partition (used in EDVR) and the official validation partition. The original val dataset (clip names from 000 to 029) are modified to avoid conflicts with training dataset (total 240 clips). Specifically, the clip names are changed to 240, 241, ... 269.

You can prepare the REDS dataset by running:

```
python tools/dataset_converters/super-resolution/reds/preprocess_reds_dataset.py --root-
↪path ./data/REDS
```

```
mmediting
├── mmedit
├── tools
├── configs
├── data
|   ├── REDS
|   |   ├── train_sharp
|   |   |   ├── 000
|   |   |   ├── 001
|   |   |   ├── ...
|   |   ├── train_sharp_bicubic
|   |   |   ├── 000
|   |   |   ├── 001
|   |   |   ├── ...
|   ├── REDS4
|   |   ├── GT
|   |   ├── sharp_bicubic
```

1.27.1 Prepare LMDB dataset for REDS

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/reds/preprocess_reds_dataset.py --root-
↪path ./data/REDS --make-lmdb
```

1.27.2 Crop to sub-images

MMEditing also support cropping REDS images to sub-images for faster IO. We provide such a script:

```
python tools/dataset_converters/super-resolution/reds/crop_sub_images.py --data-root ./
↪data/REDS  -scales 4
```

The generated data is stored under `REDS` and the data structure is as follows, where `_sub` indicates the sub-images.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── REDS
        ├── train_sharp
            ├── 000
            ├── 001
            ├── ...
        ├── train_sharp_sub
            ├── 000_s001
            ├── 000_s002
            ├── ...
            ├── 001_s001
            ├── ...
        ├── train_sharp_bicubic
            ├── X4
                ├── 000
                ├── 001
                ├── ...
            ├── X4_sub
            ├── 000_s001
            ├── 000_s002
            ├── ...
            ├── 001_s001
            ├── ...
```

Note that by default `preprocess_reds_dataset.py` does not make lmdb and annotation file for the cropped dataset. You may need to modify the scripts a little bit for such operations.

1.28 Unconditional GANs Datasets

Data preparation for unconditional model is simple. What you need to do is downloading the images and put them into a directory. Next, you should set a symlink in the `data` directory. For standard unconditional gans with static architectures, like DCGAN and StyleGAN2, `UnconditionalImageDataset` is designed to train such unconditional models. Here is an example config for FFHQ dataset:

```
dataset_type = 'BasicImageDataset'

train_pipeline = [
    dict(type='LoadImageFromFile', key='img'),
    dict(type='Flip', keys=['img'], direction='horizontal'),
    dict(type='PackEditInputs', keys=['img'], meta_keys=['img_path'])
]

# `batch_size` and `data_root` need to be set.
train_dataloader = dict(
    batch_size=4,
    num_workers=8,
    persistent_workers=True,
    sampler=dict(type='InfiniteSampler', shuffle=True),
    dataset=dict(
        type=dataset_type,
        data_root=None,  # set by user
        pipeline=train_pipeline))
```

Here, we adopt `InfinitySampler` to avoid frequent dataloader reloading, which will accelerate the training procedure. As shown in the example, `pipeline` provides important data pipeline to process images, including loading from file system, resizing, cropping, transferring to `torch.Tensor` and packing to `EditDataSample`. All of supported data pipelines can be found in `mmedit/datasets/transforms`.

For unconditional GANs with dynamic architectures like PGGAN and StyleGANv1, `GrowScaleImgDataset` is recommended to use for training. Since such dynamic architectures need real images in different scales, directly adopting `UnconditionalImageDataset` will bring heavy I/O cost for loading multiple high-resolution images. Here is an example we use for training PGGAN in CelebA-HQ dataset:

```
dataset_type = 'GrowScaleImgDataset'

pipeline = [
    dict(type='LoadImageFromFile', key='img'),
    dict(type='Flip', keys=['img'], direction='horizontal'),
    dict(type='PackEditInputs')
]

# `samples_per_gpu` and `imgs_root` need to be set.
train_dataloader = dict(
    num_workers=4,
    batch_size=64,
    dataset=dict(
        type='GrowScaleImgDataset',
        data_roots={
            '1024': './data/ffhq/images',
            '256': './data/ffhq/ffhq_imgs/ffhq_256',
```

(continues on next page)

(continued from previous page)

```
            '64': './data/ffhq/ffhq_imgs/ffhq_64'
        },
        gpu_samples_base=4,
        # note that this should be changed with total gpu number
        gpu_samples_per_scale={
            '4': 64,
            '8': 32,
            '16': 16,
            '32': 8,
            '64': 4,
            '128': 4,
            '256': 4,
            '512': 4,
            '1024': 4
        },
        len_per_stage=300000,
        pipeline=pipeline),
    sampler=dict(type='InfiniteSampler', shuffle=True))
```

In this dataset, you should provide a dictionary of image paths to the `data_roots`. Thus, you should resize the images in the dataset in advance. For the resizing methods in the data pre-processing, we adopt bilinear interpolation methods in all of the experiments studied in MMEditing.

Note that this dataset should be used with `PGGANFetchDataHook`. In this config file, this hook should be added in the customized hooks, as shown below.

```
custom_hooks = [
    dict(
        type='GenVisualizationHook',
        interval=5000,
        fixed_input=True,
        # vis ema and orig at the same time
        vis_kwargs_list=dict(
            type='Noise',
            name='fake_img',
            sample_model='ema/orig',
            target_keys=['ema', 'orig'])),
    dict(type='PGGANFetchDataHook')
]
```

This fetching data hook helps the dataloader update the status of dataset to change the data source and batch size during training.

Here, we provide several download links of datasets frequently used in unconditional models: LSUN, CelebA, CelebA-HQ, FFHQ.

1.29 Preparing DF2K_OST Dataset

```
@inproceedings{wang2021real,
  title={Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic␣
↪Data},
  author={Wang, Xintao and Xie, Liangbin and Dong, Chao and Shan, Ying},
  booktitle={Proceedings of the IEEE/CVF International Conference on Computer Vision},
  pages={1905--1914},
  year={2021}
}
```

- The DIV2K dataset can be downloaded from here (We use the training set only).
- The Flickr2K dataset can be downloaded here (We use the training set only).
- The OST dataset can be downloaded here (We use the training set only).

Please first put all the images into the `GT` folder (naming does not need to be in order):

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── df2k_ost
        ├── GT
            ├── 0001.png
            ├── 0002.png
            ├── ...
...
```

1.29.1 Crop sub-images

For faster IO, we recommend to crop the images to sub-images. We provide such a script:

```
python tools/dataset_converters/super-resolution/df2k_ost/preprocess_df2k_ost_dataset.py␣
↪--data-root ./data/df2k_ost
```

The generated data is stored under `df2k_ost` and the data structure is as follows, where `_sub` indicates the sub-images.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── df2k_ost
        ├── GT
        ├── GT_sub
...
```

1.29.2 Prepare LMDB dataset for DF2K_OST

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/df2k_ost/preprocess_df2k_ost_dataset.py␣
↪--data-root ./data/df2k_ost --make-lmdb
```

1.30 Preparing DIV2K Dataset

```
@InProceedings{Agustsson_2017_CVPR_Workshops,
    author = {Agustsson, Eirikur and Timofte, Radu},
    title = {NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study},
    booktitle = {The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)␣
↪Workshops},
    month = {July},
    year = {2017}
}
```

- Training dataset: DIV2K dataset.
- Validation dataset: Set5 and Set14.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── DIV2K
        ├── DIV2K_train_HR
        ├── DIV2K_train_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
        ├── DIV2K_valid_HR
        ├── DIV2K_valid_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
    ├── Set5
        ├── GTmod12
        ├── LRbicx2
        ├── LRbicx3
        ├── LRbicx4
    ├── Set14
        ├── GTmod12
        ├── LRbicx2
        ├── LRbicx3
        ├── LRbicx4
```

1.30.1 Crop sub-images

For faster IO, we recommend to crop the DIV2K images to sub-images. We provide such a script:

```
python tools/dataset_converters/super-resolution/div2k/preprocess_div2k_dataset.py --
↪data-root ./data/DIV2K
```

The generated data is stored under `DIV2K` and the data structure is as follows, where `_sub` indicates the sub-images.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── DIV2K
        ├── DIV2K_train_HR
        ├── DIV2K_train_HR_sub
        ├── DIV2K_train_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
            ├── X2_sub
            ├── X3_sub
            ├── X4_sub
        ├── DIV2K_valid_HR
        ├── ...
...
```

1.30.2 Prepare annotation list

If you use the annotation mode for the dataset, you first need to prepare a specific `txt` file.

Each line in the annotation file contains the image names and image shape (usually for the ground-truth images), separated by a white space.

Example of an annotation file:

```
0001_s001.png (480,480,3)
0001_s002.png (480,480,3)
```

1.30.3 Prepare LMDB dataset for DIV2K

If you want to use LMDB datasets for faster IO speed, you can make LMDB files by:

```
python tools/dataset_converters/super-resolution/div2k/preprocess_div2k_dataset.py --
↪data-root ./data/DIV2K --make-lmdb
```

1.31 Preparing Composition-1k Dataset

1.31.1 Introduction

```
@inproceedings{xu2017deep,
  title={Deep image matting},
  author={Xu, Ning and Price, Brian and Cohen, Scott and Huang, Thomas},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={2970--2979},
  year={2017}
}
```

The Adobe Composition-1k dataset consists of foreground images and their corresponding alpha images. To get the full dataset, one need to composite the foregrounds with selected backgrounds from the COCO dataset and the Pascal VOC dataset.

1.31.2 Obtain and Extract

Please follow the instructions of paper authors to obtain the Composition-1k (comp1k) dataset.

1.31.3 Composite the full dataset

The Adobe composition-1k dataset contains only `alpha` and `fg` (and `trimap` in test set). It is needed to merge `fg` with COCO data (training) or VOC data (test) before training or evaluation. Use the following script to perform image composition and generate annotation files for training or testing:

```
# The script is run under the root folder of MMEditing
python tools/dataset_converters/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_
↪composition-1k data/coco data/VOCdevkit --composite
```

The generated data is stored under `adobe_composition-1k/Training_set` and `adobe_composition-1k/Test_set` respectively. If you only want to composite test data (since compositing training data is time-consuming), you can skip compositing the training set by removing the `--composite` option:

```
# skip compositing training set
python tools/dataset_converters/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_
↪composition-1k data/coco data/VOCdevkit
```

If you only want to preprocess test data, i.e. for FBA, you can skip the train set by adding the `--skip-train` option:

```
# skip preprocessing training set
python tools/data/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_composition-1k␣
↪data/coco data/VOCdevkit --skip-train
```

> Currently, `GCA` and `FBA` support online composition of training data. But you can modify the data pipeline of other models to perform online composition instead of loading composited images (we called it `merged` in our data pipeline).

1.31.4 Check Directory Structure for DIM

The result folder structure should look like:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
│   ├── adobe_composition-1k
│   │   ├── Test_set
│   │   │   ├── Adobe-licensed images
│   │   │   │   ├── alpha
│   │   │   │   ├── fg
│   │   │   │   └── trimaps
│   │   │   ├── merged  (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   │   └── bg      (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   ├── Training_set
│   │   │   ├── Adobe-licensed images
│   │   │   │   ├── alpha
│   │   │   │   └── fg
│   │   │   ├── Other
│   │   │   │   ├── alpha
│   │   │   │   └── fg
│   │   │   ├── merged  (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   │   └── bg      (generated by tools/dataset_converters/matting/comp1k/preprocess_
↪comp1k_dataset.py)
│   │   ├── test_list.json     (generated by tools/dataset_converters/matting/comp1k/
↪preprocess_comp1k_dataset.py)
│   │   ├── training_list.json (generated by tools/dataset_converters/matting/comp1k/
↪preprocess_comp1k_dataset.py)
│   ├── coco
│   │   ├── train2014   (or train2017)
│   ├── VOCdevkit
│   │   ├── VOC2012
```

1.31.5 Prepare the dataset for FBA

FBA adopts dynamic dataset augmentation proposed in Learning-base Sampling for Natural Image Matting. In addition, to reduce artifacts during augmentation, it uses the extended version of foreground as foreground. We provide scripts to estimate foregrounds.

Prepare the test set as follows:

```
# skip preprocessing training set, as it composites online during training
python tools/dataset_converters/matting/comp1k/preprocess_comp1k_dataset.py data/adobe_
↪composition-1k data/coco data/VOCdevkit --skip-train
```

Extend the foreground of training set as follows:

```
python tools/dataset_converters/matting/comp1k/extend_fg.py data/adobe_composition-1k
```

1.31.6 Check Directory Structure for DIM

The final folder structure should look like:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── adobe_composition-1k
        ├── Test_set
            ├── Adobe-licensed images
                ├── alpha
                ├── fg
                ├── trimaps
            ├── merged  (generated by tools/data/matting/comp1k/preprocess_comp1k_
↪dataset.py)
            ├── bg      (generated by tools/data/matting/comp1k/preprocess_comp1k_
↪dataset.py)
        ├── Training_set
            ├── Adobe-licensed images
                ├── alpha
                ├── fg
                ├── fg_extended (generated by tools/data/matting/comp1k/extend_fg.py)
            ├── Other
                ├── alpha
                ├── fg
                ├── fg_extended (generated by tools/data/matting/comp1k/extend_fg.py)
        ├── test_list.json         (generated by tools/data/matting/comp1k/preprocess_
↪comp1k_dataset.py)
        ├── training_list_fba.json (generated by tools/data/matting/comp1k/extend_fg.py)
    ├── coco
        ├── train2014   (or train2017)
    ├── VOCdevkit
        ├── VOC2012
```

1.32 Preparing CelebA-HQ Dataset

```
@article{karras2017progressive,
  title={Progressive growing of gans for improved quality, stability, and variation},
  author={Karras, Tero and Aila, Timo and Laine, Samuli and Lehtinen, Jaakko},
  journal={arXiv preprint arXiv:1710.10196},
  year={2017}
}
```

Follow the instructions here to prepare the dataset.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── CelebA-HQ
        ├── train_256
|   |   ├── test_256
|   |   ├── train_celeba_img_list.txt
|   |   ├── val_celeba_img_list.txt
```

1.33 Preparing Places365 Dataset

```
 @article{zhou2017places,
   title={Places: A 10 million Image Database for Scene Recognition},
   author={Zhou, Bolei and Lapedriza, Agata and Khosla, Aditya and Oliva, Aude and␣
↪Torralba, Antonio},
   journal={IEEE Transactions on Pattern Analysis and Machine Intelligence},
   year={2017},
   publisher={IEEE}
 }
```

Prepare the data from Places365.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── Places
        ├── data_large
        ├── val_large
|   |   ├── meta
|   |   |   ├── places365_train_challenge.txt
|   |   |   ├── places365_val.txt
```

1.34 Preparing Paris Street View Dataset

```
@inproceedings{pathak2016context,
  title={Context encoders: Feature learning by inpainting},
  author={Pathak, Deepak and Krahenbuhl, Philipp and Donahue, Jeff and Darrell, Trevor␣
↪and Efros, Alexei A},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={2536--2544},
  year={2016}
}
```

Obtain the dataset here.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── paris_street_view
        ├── train
|   |   ├── val
```

1.35 Preparing Vimeo90K-triplet Dataset

```
@article{xue2019video,
  title={Video Enhancement with Task-Oriented Flow},
  author={Xue, Tianfan and Chen, Baian and Wu, Jiajun and Wei, Donglai and Freeman,␣
↪William T},
  journal={International Journal of Computer Vision (IJCV)},
  volume={127},
  number={8},
  pages={1106--1125},
  year={2019},
  publisher={Springer}
}
```

The training and test datasets can be download from here.

The Vimeo90K-triplet dataset has a `clip/sequence/img` folder structure:

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── vimeo_triplet
        ├── tri_testlist.txt
        ├── tri_trainlist.txt
        ├── sequences
            ├── 00001
                ├── 0001
                    ├── im1.png
                    ├── im2.png
                    └── im3.png
                ├── 0002
                ├── 0003
                ├── ...
            ├── 00002
            ├── ...
```

1.36 Preparing Paired Dataset for Pix2pix

```
@inproceedings{isola2017image,
  title={Image-to-image translation with conditional adversarial networks},
  author={Isola, Phillip and Zhu, Jun-Yan and Zhou, Tinghui and Efros, Alexei A},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={1125--1134},
  year={2017}
}
```

You can download paired datasets from here. Then, you need to unzip and move corresponding datasets to follow the folder structure shown above. The datasets have been well-prepared by the original authors.

```
mmediting
├── mmedit
├── tools
├── configs
├── data
    ├── paired
        ├── facades
        ├── maps
|   |   ├── edges2shoes
|   |   |   ├── train
|   |   |   ├── test
```

1.37 Changelog

1.37.1 v1.0.0rc5 (04/01/2023)

Highlights We are excited to announce the release of MMEditing 1.0.0rc5. This release supports 49+ models, 180+ configs and 177+ checkpoints in MMGeneration and MMEditing. We highlight the following new features

- Support Restormer
- Support GLIDE
- Support SwinIR
- Support Stable Diffusion

New Features & Improvements

- Disco notebook.(#1507)
- Revise test requirements and CI.(#1514)
- Recursive generate summary and docstring.(#1517)
- Enable projects.(#1526)
- Support mscoco dataset.(#1520)
- Improve Chinese documents.(#1532)
- Type hints.(#1481)

- Update download link.(#1554)
- Update deployment guide.(#1551)

Bug Fixes

- Fix documentation link checker.(#1522)
- Fix ssim first channel bug.(#1515)
- Fix restormer ut.(#1550)
- Fix extract_gt_data of realesrgan.(#1542)
- Fix model index.(#1559)
- Fix config path in disco-diffusion.(#1553)
- Fix text2image inferencer.(#1523)

Contributors A total of 16 developers contributed to this release. Thanks @plyfager, @LeoXing1996, @Z-Fran, @zengyh1900, @VongolaWu, @liuwenran, @AlexZou14, @lvhan028, @xiaomile, @ldr426, @austin273, @whu-lee, @willaty, @curiosity654, @Zdafeng, @Taited

New Contributors

- @xiaomile made their first contribution in https://github.com/open-mmlab/mmediting/pull/1481
- @ldr426 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1542
- @austin273 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1553
- @whu-lee made their first contribution in https://github.com/open-mmlab/mmediting/pull/1539
- @willaty made their first contribution in https://github.com/open-mmlab/mmediting/pull/1541
- @curiosity654 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1556
- @Zdafeng made their first contribution in https://github.com/open-mmlab/mmediting/pull/1476
- @Taited made their first contribution in https://github.com/open-mmlab/mmediting/pull/1534

1.37.2 v1.0.0rc4 (05/12/2022)

Highlights

We are excited to announce the release of MMEditing 1.0.0rc4. This release supports 45+ models, 176+ configs and 175+ checkpoints in MMGeneration and MMEditing. We highlight the following new features

- Support High-level APIs.
- Support diffusion models.
- Support Text2Image Task.
- Support 3D-Aware Generation.

New Features & Improvements

- Refactor high-level APIs. (#1410)
- Support disco-diffusion text-2-image. (#1234, #1504)
- Support EG3D. (#1482, #1493, #1494, #1499)
- Support NAFNet model. (#1369)

Bug Fixes

- fix srgan train config. (#1441)
- fix cain config. (#1404)
- fix rdn and srcnn train configs. (#1392)
- Revise config and pretrain model loading in esrgan. (#1407)

Contributors A total of 14 developers contributed to this release. Thanks @plyfager, @LeoXing1996, @Z-Fran, @zengyh1900, @VongolaWu, @gaoyang07, @ChangjianZhao, @zxczrx123, @jackghosts, @liuwenran, @CCODING04, @RoseZhao929, @shaocongliu, @liangzelong.

New Contributors

- @gaoyang07 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1372
- @ChangjianZhao made their first contribution in https://github.com/open-mmlab/mmediting/pull/1461
- @zxczrx123 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1462
- @jackghosts made their first contribution in https://github.com/open-mmlab/mmediting/pull/1463
- @liuwenran made their first contribution in https://github.com/open-mmlab/mmediting/pull/1410
- @CCODING04 made their first contribution in https://github.com/open-mmlab/mmediting/pull/783
- @RoseZhao929 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1474
- @shaocongliu made their first contribution in https://github.com/open-mmlab/mmediting/pull/1470
- @liangzelong made their first contribution in https://github.com/open-mmlab/mmediting/pull/1488

1.37.3 v1.0.0rc3 (03/11/2022)

Highlights

We are excited to announce the release of MMEditing 1.0.0rc3. This release supports 43+ models, 170+ configs and 169+ checkpoints in MMGeneration and MMEditing. We highlight the following new features

- convert `mmdet` and `clip` to optional requirements.

New Features & Improvements

- Support `try_import` for `mmdet`. (#1408)
- Support `try_import` for `flip`. (#1420)
- Complete requirements (#1419)
- Update `.gitignore`. ($1416)
- Set `real_feat` to cpu in `inception_utils`. (#1415)
- Modify README and configs of StyleGAN2 and PEGAN (#1418)
- Improve the rendering of Docs-API (#1373)

Bug Fixes

- Revise config and pretrain model loading in ESRGAN (#1407)
- Revise config of LSGAN (#1409)
- Revise config of CAIN (#1404)

Contributors

A total of 5 developers contributed to this release. @Z-Fran, @zengyh1900, @plyfager, @LeoXing1996, @ruoningYu.

1.37.4 v1.0.0rc2 (02/11/2022)

Highlights

We are excited to announce the release of MMEditing 1.0.0rc2. This release supports 43+ models, 170+ configs and 169+ checkpoints in MMGeneration and MMEditing. We highlight the following new features

- patch-based and slider-based image and video comparison viewer.
- image colorization.

We want to sincerely thank our community for continuously improving MMEditing.

New Features & Improvements

- Support qualitative comparison tools. (#1303)
- Support instance aware colorization. (#1370)
- Support multi-metrics with different sample-model. (#1171)
- Improve the implementation
  - refactoring evaluation metrics. (#1164)
  - Save gt images in PGGAN's `forward`. (#1332)
  - Improve type and change default number of `preprocess_div2k_dataset.py`. (#1380)
  - Support pixel value clip in visualizer. (#1365)
  - Support SinGAN Dataset and SinGAN demo. (#1363)
  - Avoid cast int and float in GenDataPreprocessor. (#1385)
- Improve the documentation
  - Update a menu switcher. (#1162)
  - Fix TTSR's README. (#1325)
  - Revise docs (change `PackGenInputs` and `GenDataSample`). (#1382)

Bug Fixes

- Fix PPL bug. (#1172)
- Fix RDN number of channels. (#1328)
- Fix types of exceptions in demos. (#1372)
- Fix realesrgan ema. (#1341)
- Improve the assertion to ensuer `GenerateFacialHeatmap` as `np.float32`. (#1310)
- Fix sampling behavior of `unpaired_dataset.py` and urls in cyclegan's README. (#1308)
- Fix vsr models in pytorch2onnx. (#1300)
- Fix incorrect settings in configs. (#1167,#1200,#1236,#1293,#1302,#1304,#1319,#1331,#1336,#1349,#1352,#1353,#1358,#1364,

New Contributors

- @gaoyang07 made their first contribution in https://github.com/open-mmlab/mmediting/pull/1372

Contributors

A total of 7 developers contributed to this release. Thanks @LeoXing1996, @Z-Fran, @zengyh1900, @plyfager, @ryanxingql, @ruoningYu, @gaoyang07.

1.37.5 v1.0.0rc1(23/9/2022)

MMEditing 1.0.0rc1 has merged MMGeneration 1.x.

- Support 42+ algorithms, 169+ configs and 168+ checkpoints.
- Support 26+ loss functions, 20+ metrics.
- Support tensorboard, wandb.
- Support unconditional GANs, conditional GANs, image2image translation and internal learning.

1.37.6 v1.0.0rc0(31/8/2022)

MMEditing 1.0.0rc0 is the first version of MMEditing 1.x, a part of the OpenMMLab 2.0 projects.

Built upon the new training engine, MMEditing 1.x unifies the interfaces of dataset, models, evaluation, and visualization.

And there are some BC-breaking changes. Please check the migration tutorial for more details.

1.37.7 v0.15.0 (01/06/2022)

Highlights

1. Support FLAVR
2. Support AOT-GAN
3. Support CAIN with ReduceLROnPlateau Scheduler

New Features

- Add configs for AOT-GAN (#681)
- Support Vimeo90k-triplet dataset (#810)
- Add default config for mm-assistant (#827)
- Support CPU demo (#848)
- Support `use_cache` and `backend` in LoadImageFromFileList (#857)
- Support VFIVimeo90K7FramesDataset (#858)
- Support ColorJitter for VFI (#859)
- Support ReduceLrUpdaterHook (#860)
- Support `after_val_epoch` in IterBaseRunner (#861)
- Support FLAVR Net (#866, #867, #897)
- Support MAE metric (#871)
- Use mdformat (#888)
- Support CAIN with ReduceLROnPlateau Scheduler (#906)

Bug Fixes

- Change - to _ for restoration_demo.py (#834)
- Remove recommonmark in requirements/docs.txt (#844)
- Move EDVR to VSR category in README.md (#849)

- Remove ， in multi-line F-string in crop.py (#855)
- Modify double `lq_path` to `gt_path` in test_pipeline (#862)
- Fix unittest of TOF-VFI (#873)
- Fix wrong frames in VFI demo (#891)
- Fix logo & contrib guideline on README (#898)
- Normalizing trimap in indexnet_dimaug_mobv2_1x16_78k_comp1k.py (#901)

Improvements

- Add `--cfg-options` in train/test scripts (#826)
- Update MMCV_MAX to 1.6 (#829)
- Update TOFlow in README (#835)
- Recover beirf installation steps & merge optional requirements (#836)
- Use {MMEditing Contributors} in citation (#838)
- Add tutorial for customizing losses (#839)
- Add installation guide (wiki ver) in README (#845)
- Add 'need help to traslate' note on Chinese documentation (#850)
- Add wechat QR code in README_zh-CN.md (#851)
- Support non-zero frame index for SRFolderVideoDataset & Fix Typos (#853)
- Create README.md for docker (#856)
- Optimize IO for flow_warp (#881)
- Move wiki/installation to docs (#883)
- Add `myst_heading_anchors` (#887)
- Use checkpoint link in inpainting demo (#892)

Contributors

@wangruohui @quincylin1 @nijkah @jayagami @ckkelvinchan @ryanxingql @NK-CS-ZZL @Yshuo-Li

1.37.8 v0.14.0 (01/04/2022)

Highlights

1. Support TOFlow in video frame interpolation

New Features

- Support AOT-GAN (#677)
- Use `--diff-seed` to set different torch seed on different rank (#781)
- Support streaming reading of frames in video interpolation demo (#790)
- Support dist_train without slurm (#791)
- Put LQ into CPU for restoration_video_demo (#792)
- Support gray normalization constant in EDSR (#793)
- Support TOFlow in video frame interpolation (#806, #811)

- Support seed in DistributedSampler and sync seed across ranks (#815)

Bug Fixes

- Update link in README files (#782, #786, #819, #820)
- Fix matting tutorial, and fix links to colab (#795)
- Invert `flip_ratio` in `RandomAffine` pipeline (#799)
- Update preprocess_div2k_dataset.py (#801)
- Update SR Colab Demo Installation Method and Set5 link (#807)
- Fix Y/GRB mistake in EDSR README (#812)
- Replace pytorch install command to conda in README(_zh-CN).md (#816)

Improvements

- Update CI (#650)
- Update requirements.txt (#725, #817)
- Add Tutorial of dataset (#758), pipeline (#779), model (#766)
- Update index and TOC tree (#767)
- Make update_model_index.py compatible on windows (#768)
- Update doc build system (#769)
- Update keyword and classifier for setuptools (#773)
- Renovate installation (#776, #800)
- Update BasicVSR++ and RealBasicVSR docs (#778)
- Update citation (#785, #787)
- Regroup docs (#788)
- Use full name of config as 'Name' in metafile (#798)
- Update figure and video demo in README (#802)
- Add `clamp(0, 1)` in test of video frame interpolation (#805)
- Use hyphen for command line args in demo & tools (#808), and keep underline for required arguments in python files (#822)
- Make dataset.pipeline a dedicated section in doc (#813)
- Update mmcv-full>=1.3.13 to support DCN on CPU (#823)

Contributors

@wangruohui @ckkelvinchan @Yshuo-Li @nijkah @wdmwhh @freepoet @quincylin1

1.37.9 v0.13.0 (01/03/2022)

Highlights

1. Support CAIN
2. Support EDVR-L
3. Support running in Windows

New Features

- Add test-time ensemble for images and videos and support ensemble in BasicVSR series (#585)
- Support AOT-GAN (work in progress) (#674, #675, #676)
- Support CAIN (#683, #691, #709, #713)
- Add basic interpolater (#687)
- Add BaseVFIDataset and VFIVimeo90KDataset (#695, #697)
- Add video interpolation demo (#688, #717)
- Support various scales in RRDBNet (#699)
- Support Ref-SR inference (#716)
- Support EDVR-L on REDS (#719)
- Support CPU training (#720)
- Support running in Windows (#732, #738)
- Support DCN on CPU (#735)

Bug Fixes

- Fix link address in docs (#703, #704)
- Fix ARG `MMCV` in Dockerfile (#708)
- Fix file permission of non-executable files (#718)
- Fix some deprecation warning related to numpy (#728)
- Delete `__init__` in `TestVFIDataset` (#731)
- Fix data type in docstring of several Datasets (#739)
- Fix math notation in docstring (#741)
- Fix missing folders in copyright commit hook (#754)
- Delete duplicate test in loading (#756)

Improvements

- Update Pillow from 6.2.2 to 8.4 in CI (#693)
- Add argument 'repeat' to SRREDSMultipleGTDataset (#672)
- Deprecate the support for "python setup.py test" (#701)
- Add setup multi-processing both in train and test (#707)
- Add OpenMMLab website and platform links (#710)
- Refact README files of all methods (#712)
- Replace string version comparison with `package.version.parse` (#723)

- Add docs of Ref-SR demo and video frame interpolation demo (#724)
- Add interpolation and refact README.md (#726)
- Update isort version in pre-commit hook (#727)
- Redesign CI for Linux (#734)
- Update install.md (#763)
- Reorganizing OpenMMLab projects in readme (#764)
- Add deprecation message for deploy tools (#765)

Contributors

@wangruohui @ckkelvinchan @Yshuo-Li @quincylin1 @Juggernaut93 @anse3832 @nijkah

1.37.10 v0.12.0 (31/12/2021)

Highlights

1. Support RealBasicVSR
2. Support Real-ESRGAN checkpoint

New Features

- Support video input and output in restoration demo (#622)
- Support RealBasicVSR (#632, #633, #647, #680)
- Support Real-ESRGAN checkpoint (#635)
- Support conversion to y-channel when loading images (643)
- Support random video compression during training (#646)
- Support crop sequence (#648)
- Support pixel_unshuffle (#684)

Bug Fixes

- Change 'target_size' for RandomResize from list to tuple (#617)
- Fix folder creation in preprocess_df2k_ost_dataset.py (#623)
- Change TDAN config path in README (#625)
- Change 'radius' to 'kernel_size' for UnsharpMasking in Real-ESRNet config (#626)
- Fix bug in MATLABLikeResize (#630)
- Fix 'flow_warp' comment (#655)
- Fix the error of Model Zoo and Datasets in docs (#664)
- Fix bug in 'random_degradations' (#673)
- Limit opencv-python version (#689)

Improvements

- Translate docs to Chinese (#576, #577, #578, #579, #581, #582, #584, #587, #588, #589, #590, #591, #592, #593, #594, #595, #596, #641, #647, #656, #665, #666)
- Add UNetDiscriminatorWithSpectralNorm (#605)

- Use PyTorch sphinx theme (#607, #608)
- Update mmcv (#609), mmflow (#621), mmfewshot (#634) and mmhuman3d (#649) in docs
- Convert minimum GCC version to 5.4 (#612)
- Add tiff in SRDataset IMG_EXTENSIONS (#614)
- Update metafile and update_model_index.py (#615)
- Update preprocess_df2k_ost_dataset.py (#624)
- Add Abstract to README (#628, #636)
- Align NIQE to MATLAB results (#631)
- Add official markdown lint hook (#639)
- Skip CI when some specific files were changed (#640)
- Update docs/conf.py (#644, #651)
- Try to create a symbolic link on windows (#645)
- Cancel previous runs that are not completed (#650)
- Update path of configs in demo.md and getting_started.md (#658, #659)
- Use mmcv root model registry (#660)
- Update README.md (#654, #663)
- Refactor the structure of documentation (#668)
- Add script to crop REDS images into sub-images for faster IO (#669)
- Capitalize the first letter of the task name in the metafile (#678)
- Update FixedCrop for cropping image sequence (#682)

1.37.11 v0.11.0 (03/11/2021)

Highlights

- GLEAN for blind face image restoration #530
- Real-ESRGAN model #546

New Features

- Exponential Moving Average Hook #542
- Support DF2K_OST dataset #566

Improvements

- Add MATLAB-like bicubic interpolation #507
- Support random degradations during training #504
- Support torchserve #568

1.37.12 v0.10.0 (12/08/2021).

Highlights

1. Support LIIF-RDN (CVPR'2021)
2. Support BasicVSR++ (NTIRE'2021)

New Features

- Support loading annotation from file for video SR datasets (#423)
- Support persistent worker (#426)
- Support LIIF-RDN (#428, #440)
- Support BasicVSR++ (#451, #467)
- Support mim (#455)

Bug Fixes

- Fix bug in stat.py (#420)
- Fix astype error in function tensor2img (#429)
- Fix device error caused by torch.new_tensor when pytorch >= 1.7 (#465)
- Fix _non_dist_train in .mmedit/apis/train.py (#473)
- Fix multi-node distributed test (#478)

Breaking Changes

- Refactor LIIF for pytorch2onnx (#425)

Improvements

- Update Chinese docs (#415, #416, #418, #421, #424, #431, #442)
- Add CI of pytorch 1.9.0 (#444)
- Refactor README.md of configs (#452)
- Avoid loading pretrained VGG in unittest (#466)
- Support specifying scales in preprocessing div2k dataset (#472)
- Support all formats in readthedocs (#479)
- Use version_info of mmcv (#480)
- Remove unnecessary codes in restoration_video_demo.py (#484)
- Change priority of DistEvalIterHook to 'LOW' (#489)
- Reset resource limit (#491)
- Update QQ QR code in README_CN.md (#494)
- Add `myst_parser` (#495)
- Add license header (#496)
- Fix typo of StyleGAN modules (#427)
- Fix typo in docs/demo.md (#453, #454)
- Fix typo in tools/data/super-resolution/reds/README.md (#469)

1.37.13 v0.9.0 (30/06/2021).

Highlights

1. Support DIC and DIC-GAN (CVPR'2020)
2. Support GLEAN Cat 8x (CVPR'2021)
3. Support TTSR-GAN (CVPR'2020)
4. Add colab tutorial for super-resolution

New Features

- Add DIC (#342, #345, #348, #350, #351, #357, #363, #365, #366)
- Add SRFolderMultipleGTDataset (#355)
- Add GLEAN Cat 8x (#367)
- Add SRFolderVideoDataset (#370)
- Add colab tutorial for super-resolution (#380)
- Add TTSR-GAN (#372, #381, #383, #398)
- Add DIC-GAN (#392, #393, #394)

Bug Fixes

- Fix bug in restoration_video_inference.py (#379)
- Fix Config of LIIF (#368)
- Change the path to pre-trained EDVR-M (#396)
- Fix normalization in restoration_video_inference (#406)
- Fix [brush_stroke_mask] error in unittest (#409)

Breaking Changes

- Change mmcv minimum version to v1.3 (#378)

Improvements

- Correct Typos in code (#371)
- Add Custom_hooks (#362)
- Refactor unittest folder structure (#386)
- Add documents and download link for Vid4 (#399)
- Update model zoo for documents (#400)
- Update metafile (407)

1.37.14 v0.8.0 (31/05/2021).

Highlights

1. Support GLEAN (CVPR'2021)
2. Support TTSR (CVPR'2020)
3. Support TDAN (CVPR'2020)

New Features

- Add GLEAN (#296, #332)
- Support PWD metafile (#298)
- Support CropLike in pipeline (#299)
- Add TTSR (#301, #304, #307, #311, #311, #312, #313, #314, #321, #326, #327)
- Add TDAN (#316, #334, #347)
- Add onnx2tensorrt (#317)
- Add tensorrt evaluation (#328)
- Add SRFacicalLandmarkDataset (#329)
- Add key point auxiliary model for DIC (#336, #341)
- Add demo for video super-resolution methods (#275)
- Add SR Folder Ref Dataset (#292)
- Support FLOPs calculation of video SR models (#309)

Bug Fixes

- Fix find_unused_parameters in PyTorch 1.8 for BasicVSR (#290)
- Fix error in publish_model.py for pt>=1.6 (#291)
- Fix PSNR when input is uint8 (#294)

Improvements

- Support backend in LoadImageFromFile (#293, #303)
- Update `metric_average_mode` of video SR dataset (#319)
- Add error message in restoration_demo.py (324)
- Minor correction in getting_started.md (#339)
- Update description for Vimeo90K (#349)
- Support start_index in GenerateSegmentIndices (#338)
- Support different filename templates in GenerateSegmentIndices (#325)
- Support resize by scale-factor (#295, #310)

1.37.15 v0.7.0 (30/04/2021).

Highlights

1. Support BasicVSR (CVPR'2021)
2. Support IconVSR (CVPR'2021)
3. Support RDN (CVPR'2018)
4. Add onnx evaluation tool

New Features

- Add RDN (#233, #260, #280)
- Add MultipleGT datasets (#238)
- Add BasicVSR and IconVSR (#245, #252, #253, #254, #264, #274, #258, #252, #264)
- Add onnx evaluation tool (#279)

Bug Fixes

- Fix onnx conversion of maxunpool2d (#243)
- Fix inpainting in `demo.md` (#248)
- Tiny fix of config file of EDSR (#251)
- Fix link in README (#256)
- Fix restoration_inference key missing bug (#270)
- Fix the usage of channel_order in `loading.py` (#271)
- Fix the command of inpainting (#278)
- Fix `preprocess_vimeo90k_dataset.py` args name (#281)

Improvements

- Support `empty_cache` option in `test.py` (#261)
- Update projects in README (#249, #276)
- Support Y-channel PSNR and SSIM (#250)
- Add zh-CN README (#262)
- Update pytorch2onnx doc (#265)
- Remove extra quotation in English readme (#268)
- Change tags to comment (#269)
- List `model zoo` in README (#284, #285, #286)

1.37.16 v0.6.0 (08/04/2021).

Highlights

1. Support Local Implicit Image Function (LIIF)
2. Support exporting DIM and GCA from Pytorch to ONNX

New Features

- Add readthedocs config files and fix docstring (#92)
- Add github action file (#94)
- Support exporting DIM and GCA from Pytorch to ONNX (#105)
- Support concatenating datasets (#106)
- Support `non_dist_train` validation (#110)
- Add matting colab tutorial (#111)
- Support niqe metric (#114)
- Support PoolDataLoader for parrots (#134)
- Support collect-env (#137, #143)
- Support pt1.6 cpu/gpu in CI (#138)
- Support fp16 (139, #144)
- Support publishing to pypi (#149)
- Add modelzoo statistics (#171, #182, #186)
- Add doc of datasets (194)
- Support extended foreground option. (#195, #199, #200, #210)
- Support nn.MaxUnpool2d (#196)
- Add some FBA components (#203, #209, #215, #220)
- Support random down sampling in pipeline (#222)
- Support SR folder GT Dataset (#223)
- Support Local Implicit Image Function (LIIF) (#224, #226, #227, #234, #239)

Bug Fixes

- Fix `_non_dist_train` in train api (#104)
- Fix setup and CI (#109)
- Fix redundant loop bug in Normalize (#121)
- Fix `get_hash` in `setup.py` (#124)
- Fix `tool/preprocess_reds_dataset.py` (#148)
- Fix slurm train tutorial in `getting_started.md` (#162)
- Fix pip install bug (#173)
- Fix bug in config file (#185)
- Fix broken links of datasets (#236)
- Fix broken links of model zoo (#242)

Breaking Changes

- Refactor data loader configs (#201)

Improvements

- Updata requirements.txt (#95, #100)
- Update teaser (#96)
- Updata README (#93, #97, #98, #152)
- Updata model_zoo (#101)
- Fix typos (#102, #188, #191, #197, #208)
- Adopt adjust_gamma from skimage and reduce dependencies (#112)
- remove `.gitlab-ci.yml` (#113)
- Update import of first party (#115)
- Remove citation and contact (#122)
- Update version file (#136)
- Update download url (#141)
- Update `setup.py` (#150)
- Update the highest version of supported mmcv (#153, #154)
- modify `Crop` to handle a sequence of video frames (#164)
- Add links to other mm projects (#179, #180)
- Add config type (#181)
- Refactor docs (#184)
- Add config link (#187)
- Update file structure (#192)
- Update config doc (#202)
- Update `slurm_train.md` script (#204)
- Improve code style (#206, #207)
- Use `file_client` in CompositeFg (#212)
- Replace `random` with `numpy.random` (#213)
- Refactor `loader_cfg` (#214)

1.37.17 v0.5.0 (09/07/2020).

Note that **MMSR** has been merged into this repo, as a part of MMEditing. With elaborate designs of the new framework and careful implementations, hope MMEditing could provide better experience.

1.38 mmedit.apis.inferencers

1.38.1 Package Contents

Classes

MMEditInferencer	Class to assign task to different inferencers.

Functions

calculate_grid_size(→ int)	Calculate the number of images per row (nrow) to make the grid closer to
colorization_inference(model, img)	Inference image with the model.
delete_cfg(cfg[, key])	Delete key from config object.
init_model(config[, checkpoint, device])	Initialize a model from config file.
inpainting_inference(model, masked_img, mask)	Inference image with the model.
matting_inference(model, img, trimap)	Inference image(s) with the model.
restoration_face_inference(model, img[, ...])	Inference image with the model.
restoration_inference(model, img[, ref])	Inference image with the model.
restoration_video_inference(model, img_dir, ...[, ...])	Inference image with the model.
sample_conditional_model(model[, num_samples, ...])	Sampling from conditional models.
sample_img2img_model(model, image_path[, target_domain])	Sampling from translation models.
sample_unconditional_model(model[, num_samples, ...])	Sampling from unconditional models.
set_random_seed(seed[, deterministic, use_rank_shift])	Set random seed.
video_interpolation_inference(model, input_dir, output_dir)	Inference image with the model.

mmedit.apis.inferencers.**calculate_grid_size**(*num_batches: int = 1*, *aspect_ratio: int = 1*) → int

Calculate the number of images per row (nrow) to make the grid closer to square when formatting a batch of images to grid.

Parameters

- **num_batches** (*int, optional*) – Number of images per batch. Defaults to 1.
- **aspect_ratio** (*int, optional*) – The aspect ratio (width / height) of each image sample. Defaults to 1.

Returns Calculated number of images per row.

Return type int

mmedit.apis.inferencers.**colorization_inference**(*model*, *img*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.

- **img** (*str*) – Image file path.

Returns The predicted colorization result.

Return type Tensor

mmedit.apis.inferencers.**delete_cfg**(*cfg*, *key='init_cfg'*)

Delete key from config object.

Parameters

- **cfg** (str or mmengine.Config) – Config object.
- **key** (*str*) – Which key to delete.

mmedit.apis.inferencers.**init_model**(*config*, *checkpoint=None*, *device='cuda:0'*)

Initialize a model from config file.

Parameters

- **config** (str or mmengine.Config) – Config file path or the config object.
- **checkpoint** (*str, optional*) – Checkpoint path. If left as None, the model will not load any weights.
- **device** (*str*) – Which device the model will deploy. Default: 'cuda:0'.

Returns The constructed model.

Return type nn.Module

mmedit.apis.inferencers.**inpainting_inference**(*model*, *masked_img*, *mask*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **masked_img** (*str*) – File path of image with mask.
- **mask** (*str*) – Mask file path.

Returns The predicted inpainting result.

Return type Tensor

mmedit.apis.inferencers.**matting_inference**(*model*, *img*, *trimap*)

Inference image(s) with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **img** (*str*) – Image file path.
- **trimap** (*str*) – Trimap file path.

Returns The predicted alpha matte.

Return type np.ndarray

mmedit.apis.inferencers.**restoration_face_inference**(*model*, *img*, *upscale_factor=1*, *face_size=1024*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.

- **img** (*str*) – File path of input image.
- **upscale_factor** (*int, optional*) – The number of times the input image is upsampled. Default: 1.
- **face_size** (*int, optional*) – The size of the cropped and aligned faces. Default: 1024.

Returns The predicted restoration result.

Return type Tensor

mmedit.apis.inferencers.**restoration_inference**(*model*, *img*, *ref=None*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **img** (*str*) – File path of input image.
- **ref** (*str | None*) – File path of reference image. Default: None.

Returns The predicted restoration result.

Return type Tensor

mmedit.apis.inferencers.**restoration_video_inference**(*model*, *img_dir*, *window_size*, *start_idx*, *filename_tmpl*, *max_seq_len=None*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **img_dir** (*str*) – Directory of the input video.
- **window_size** (*int*) – The window size used in sliding-window framework. This value should be set according to the settings of the network. A value smaller than 0 means using recurrent framework.
- **start_idx** (*int*) – The index corresponds to the first frame in the sequence.
- **filename_tmpl** (*str*) – Template for file name.
- **max_seq_len** (*int | None*) – The maximum sequence length that the model processes. If the sequence length is larger than this number, the sequence is split into multiple segments. If it is None, the entire sequence is processed at once.

Returns The predicted restoration result.

Return type Tensor

mmedit.apis.inferencers.**sample_conditional_model**(*model*, *num_samples=16*, *num_batches=4*, *sample_model='ema'*, *label=None*, ***kwargs*)

Sampling from conditional models.

Parameters

- **model** (*nn.Module*) – Conditional models in MMGeneration.
- **num_samples** (*int, optional*) – The total number of samples. Defaults to 16.
- **num_batches** (*int, optional*) – The number of batch size for inference. Defaults to 4.
- **sample_model** (*str, optional*) – Which model you want to use. [‘ema’, ‘orig’]. Defaults to ‘ema’.

- **label** (*int | torch.Tensor | list[int], optional*) – Labels used to generate images. Default to None.,

Returns Generated image tensor.

Return type Tensor

mmedit.apis.inferencers.**sample_img2img_model**(*model*, *image_path*, *target_domain=None*, ***kwargs*)

Sampling from translation models.

Parameters

- **model** (*nn.Module*) – The loaded model.
- **image_path** (*str*) – File path of input image.
- **style** (*str*) – Target style of output image.

Returns Translated image tensor.

Return type Tensor

mmedit.apis.inferencers.**sample_unconditional_model**(*model*, *num_samples=16*, *num_batches=4*, *sample_model='ema'*, ***kwargs*)

Sampling from unconditional models.

Parameters

- **model** (*nn.Module*) – Unconditional models in MMGeneration.
- **num_samples** (*int, optional*) – The total number of samples. Defaults to 16.
- **num_batches** (*int, optional*) – The number of batch size for inference. Defaults to 4.
- **sample_model** (*str, optional*) – Which model you want to use. [‘ema’, ‘orig’]. Defaults to ‘ema’.

Returns Generated image tensor.

Return type Tensor

mmedit.apis.inferencers.**set_random_seed**(*seed*, *deterministic=False*, *use_rank_shift=True*)

Set random seed.

In this function, we just modify the default behavior of the similar function defined in MMCV.

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: True.

mmedit.apis.inferencers.**video_interpolation_inference**(*model*, *input_dir*, *output_dir*, *start_idx=0*, *end_idx=None*, *batch_size=4*, *fps_multiplier=0*, *fps=0*, *filename_tmpl='{:08d}.png'*)

Inference image with the model.

Parameters

- **model** (*nn.Module*) – The loaded model.

- **input_dir** (*str*) – Directory of the input video.
- **output_dir** (*str*) – Directory of the output video.
- **start_idx** (*int*) – The index corresponding to the first frame in the sequence. Default: 0
- **end_idx** (*int | None*) – The index corresponding to the last interpolated frame in the sequence. If it is None, interpolate to the last frame of video or sequence. Default: None
- **batch_size** (*int*) – Batch size. Default: 4
- **fps_multiplier** (*float*) – multiply the fps based on the input video. Default: 0.
- **fps** (*float*) – frame rate of the output video. Default: 0.
- **filename_tmpl** (*str*) – template of the file names. Default: '{:08d}.png'

class mmedit.apis.inferencers.**MMEditInferencer**(*task: Optional[str] = None*, *config: Optional[Union[mmedit.utils.ConfigType, str]] = None*, *ckpt: Optional[str] = None*, *device: torch.device = None*, *extra_parameters: Optional[Dict] = None*, *seed: int = 2022*)

Class to assign task to different inferencers.

Parameters

- **task** (*str*) – Inferencer task.
- **config** (*str or ConfigType*) – Model config or the path to it.
- **ckpt** (*str, optional*) – Path to the checkpoint.
- **device** (*str, optional*) – Device to run inference. If None, the best device will be automatically used.
- **seed** (*int*) – The random seed used in inference. Defaults to 2022.

__call__(***kwargs*) → Union[Dict, List[Dict]]

Call the inferencer.

Parameters **kwargs** – Keyword arguments for the inferencer.

Returns Results of inference pipeline.

Return type Union[Dict, List[Dict]]

get_extra_parameters() → List[str]

Each inferencer may has its own parameters. Call this function to get these parameters.

Returns List of unique parameters.

Return type List[str]

1.39 mmedit.structures

1.39.1 Package Contents

Classes

EditDataSample	A data structure interface of MMEditing. They are used as interfaces
PixelData	Data structure for pixel-level annnotations or predictions.

class mmedit.structures.**EditDataSample**(*, *metainfo: Optional[dict] = None*, ***kwargs*)

Bases: mmengine.structures.BaseDataElement

A data structure interface of MMEditing. They are used as interfaces between different components.

The attributes in EditDataSample are divided into several parts:

- gt_img: Ground truth image(s).
- pred_img: Image(s) of model predictions.
- ref_img: Reference image(s).
- mask: Mask in Inpainting.
- trimap: Trimap in Matting.
- gt_alpha: Ground truth alpha image in Matting.
- pred_alpha: Predicted alpha image in Matting.
- gt_fg: Ground truth foreground image in Matting.
- pred_fg: Predicted foreground image in Matting.
- gt_bg: Ground truth background image in Matting.
- pred_bg: Predicted background image in Matting.
- gt_merged: Ground truth merged image in Matting.

Examples:

```
>>> import torch
>>> import numpy as np
>>> from mmedit.structures import EditDataSample, PixelData
>>> data_sample = EditDataSample()
>>> img_meta = dict(img_shape=(800, 1196, 3))
>>> img = torch.rand((3, 800, 1196))
>>> gt_img = PixelData(data=img, metainfo=img_meta)
>>> data_sample.gt_img = gt_img
>>> assert 'img_shape' in data_sample.gt_img.metainfo_keys()
<EditDataSample(

    META INFORMATION

    DATA FIELDS
    _gt_img: <PixelData(

            META INFORMATION
            img_shape: (800, 1196, 3)
```

(continues on next page)

(continued from previous page)

```
            DATA FIELDS
            data: tensor([[[0.8069, 0.4279,  ..., 0.6603, 0.0292],

                           ...,

                           [0.8139, 0.0908,  ..., 0.4964, 0.9672]]])
        ) at 0x1f6ae000af0>
    gt_img: <PixelData(

            META INFORMATION
            img_shape: (800, 1196, 3)

            DATA FIELDS
            data: tensor([[[0.8069, 0.4279,  ..., 0.6603, 0.0292],

                           ...,

                           [0.8139, 0.0908,  ..., 0.4964, 0.9672]]])
        ) at 0x1f6ae000af0>
) at 0x1f6a5a99a00>
```

property gt_img: mmedit.structures.pixel_data.PixelData

This is the function to fetch gt_img in PixelData.

Returns data element

Return type *PixelData*

property gt_samples: *EditDataSample*

This is the function to fetch gt_samples.

Returns gt samples.

Return type *EditDataSample*

property noise: torch.Tensor

This is the function to fetch noise.

Returns noise.

Return type torch.Tensor

property pred_img: mmedit.structures.pixel_data.PixelData

This is the function to fetch pred_img in PixelData.

Returns data element

Return type *PixelData*

property fake_img: Union[mmedit.structures.pixel_data.PixelData, torch.Tensor]

This is the function to fetch fake_img.

Returns The fake img.

Return type Union[*PixelData*, Tensor]

property img_lq: mmedit.structures.pixel_data.PixelData

This is the function to fetch img_lq in PixelData.

Returns data element

Return type *PixelData*

property ref_img: mmedit.structures.pixel_data.PixelData

This is the function to fetch ref_img.

Returns data element

Return type *PixelData*

property ref_lq: mmedit.structures.pixel_data.PixelData

This is the function to fetch ref_lq.

Returns data element

Return type *PixelData*

property gt_unsharp: mmedit.structures.pixel_data.PixelData

This is the function to fetch gt_unsharp in PixelData.

Returns data element

Return type *PixelData*

property mask: mmedit.structures.pixel_data.PixelData

This is the function to fetch mask.

Returns data element

Return type *PixelData*

property gt_heatmap: mmedit.structures.pixel_data.PixelData

This is the function to fetch gt_heatmap.

Returns data element

Return type *PixelData*

property pred_heatmap: mmedit.structures.pixel_data.PixelData

This is the function to fetch pred_heatmap.

Returns data element

Return type *PixelData*

property trimap: mmedit.structures.pixel_data.PixelData

This is the function to fetch trimap.

Returns data element

Return type *PixelData*

property gt_alpha: mmedit.structures.pixel_data.PixelData

This is the function to fetch gt_alpha.

Returns data element

Return type *PixelData*

property pred_alpha: mmedit.structures.pixel_data.PixelData

This is the function to fetch pred_alpha.

Returns data element

Return type *PixelData*

property gt_fg: mmedit.structures.pixel_data.PixelData

This is the function to fetch gt_fg.

Returns data element

Return type *PixelData*

property pred_fg: mmedit.structures.pixel_data.PixelData

This is the function to fetch pred_fg.

Returns _description_

Return type *PixelData*

property gt_bg: mmedit.structures.pixel_data.PixelData

This is the function to fetch gt_bg.

Returns data element

Return type *PixelData*

property pred_bg: mmedit.structures.pixel_data.PixelData

This is the function to fetch pred_bg in PixelData.

Returns data element

Return type *PixelData*

property gt_merged: mmedit.structures.pixel_data.PixelData

This is the function to fetch gt_merged in PixelData.

Returns _description_

Return type *PixelData*

property sample_model: str

This is the function to fetch sample model.

Returns Mode of Sample model.

Return type str

property ema: *EditDataSample*

This is the function to fetch ema results.

Returns Results of the ema model.

Return type *EditDataSample*

property orig: *EditDataSample*

This is the function to fetch original results.

Returns Results of the ema model.

Return type *EditDataSample*

property gt_label

This the function to fetch gt label.

Returns gt label.

Return type LabelData

gt_img()

This is the function to fetch gt_img.

gt_samples()

This is the function to delete gt_samples.

noise()

This is the functionto delete noise.

pred_img()

This is the function to fetch pred_img.

fake_img()

This is the function to delete fake_img.

img_lq()

This is the function to delete img_lq.

ref_img()

This is the function to fetch ref_img.

ref_lq()

This is the function to delete ref_lq.

gt_unsharp()

This is the function to delete gt_unsharp.

mask()

This is the function to delete mask.

gt_heatmap()

This is the function to delete gt_heatmap.

pred_heatmap()

This is the function to fetch pred_heatmap.

trimap()

This is the function to delete trimap.

gt_alpha()

This is the function to delete gt_alpha.

pred_alpha()

This is the function to delete pred_alpha.

gt_fg()

This is the function to delete gt_fg.

pred_fg()

This is the function to delete pred_fg.

gt_bg()

This is the function to delete gt_bg.

pred_bg()

This is the function to fetch pred_bg.

gt_merged()

This is the function to fetch gt_merged.

sample_model()

This is the function to delete sample model.

ema()

This is the function to delete ema results.

orig()

This is the function to delete ema results.

set_gt_label(*value: Union[numpy.ndarray, torch.Tensor, Sequence[numbers.Number], numbers.Number]*) → *EditDataSample*

Set label of `gt_label`.

gt_label()

This is the function to delete gt label.

class mmedit.structures.**PixelData**(*, *metainfo: Optional[dict] = None*, ***kwargs*)

Bases: `mmengine.structures.PixelData`

Data structure for pixel-level annnotations or predictions.

Different from parent class: Support value.ndim == 4 for frames.

All data items in `data_fields` of `PixelData` meet the following requirements:

- They all have 3 dimensions in orders of channel, height, and width.
- They should have the same height and width.

Examples

```
>>> metainfo = dict(
...     img_id=random.randint(0, 100),
...     img_shape=(random.randint(400, 600), random.randint(400, 600)))
>>> image = np.random.randint(0, 255, (4, 20, 40))
>>> featmap = torch.randint(0, 255, (10, 20, 40))
>>> pixel_data = PixelData(metainfo=metainfo,
...                        image=image,
...                        featmap=featmap)
>>> print(pixel_data)
>>> (20, 40)
```

```
>>> # slice
>>> slice_data = pixel_data[10:20, 20:40]
>>> assert slice_data.shape == (10, 10)
>>> slice_data = pixel_data[10, 20]
>>> assert slice_data.shape == (1, 1)
```

__setattr__(*name: str*, *value: Union[torch.Tensor, numpy.ndarray]*)

Set attributes of `PixelData`.

If the dimension of value is 2 and its shape meet the demand, it will automatically expend its channel-dimension.

Parameters

- **name** (*str*) – The key to access the value, stored in *PixelData*.

- **value** (*Union[torch.Tensor, np.ndarray]*) – The value to store in. The type of value must be *torch.Tensor* or *np.ndarray*, and its shape must meet the requirements of *PixelData*.

1.40 mmedit.datasets

1.40.1 Package Contents

Classes

BasicConditionalDataset	Custom dataset for conditional GAN. This class is based on the
BasicFramesDataset	BasicFramesDataset for open source projects in OpenMMLab/MMEditing.
BasicImageDataset	BasicImageDataset for open source projects in OpenMMLab/MMEditing.
CIFAR10	CIFAR10 Dataset.
AdobeComp1kDataset	Adobe composition-1k dataset.
GrowScaleImgDataset	Grow Scale Unconditional Image Dataset.
ImageNet	ImageNet Dataset.
MSCoCoDataset	MSCoCo 2014 dataset.
PairedImageDataset	General paired image folder dataset for image generation.
SinGANDataset	SinGAN Dataset.
UnpairedImageDataset	General unpaired image folder dataset for image generation.

class mmedit.datasets.**BasicConditionalDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: Union[str, dict] = ''*, *extensions: Sequence[str] = ('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif')*, *lazy_init: bool = False*, *classes: Union[str, Sequence[str], None] = None*, ***kwargs*)

Bases: mmengine.dataset.BaseDataset

Custom dataset for conditional GAN. This class is based on the combination of *BaseDataset* (https://github.com/open- mmlab/mmclassification/blob/1.x/mmcls/datasets/base_dataset.py) # noqa and *CustomDataset* (https://github.com/open- mmlab/mmclassification/blob/1.x/mmcls/datasets/custom.py). # noqa.

The dataset supports two kinds of annotation format.

1. A annotation file read by line (e.g., txt) is provided, and each line indicates a sample:

   The sample files:

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

(continues on next page)

(continued from previous page)

```
    ├── nsdf3.png
    └── ...
```

The annotation file (the first column is the image path and the second column is the index of category):

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

Please specify the name of categories by the argument `classes` or `metainfo`.

2. A dict-based annotation file (e.g., json) is provided, key and value indicate the path and label of the sample:

   The sample files:

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

   The annotation file (the key is the image path and the value column is the label):

```
{
    "folder_1/xxx.png": [1, 2, 3, 4],
    "folder_1/xxy.png": [2, 4, 1, 0],
    "folder_2/123.png": [0, 9, 8, 1],
    "folder_2/nsdf3.png", [1, 0, 0, 2],
    ...
}
```

   In this kind of annotation, labels can be any type and not restricted to an index.

3. The samples are arranged in the specific way:

```
data_prefix/
├── class_x
│   ├── xxx.png
│   ├── xxy.png
│   └── ...
│       └── xxz.png
└── class_y
    ├── 123.png
    ├── nsdf3.png
    ├── ...
    └── asd932_.png
```

If the `ann_file` is specified, the dataset will be generated by the first two ways, otherwise, try the third way.

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str*) – The root directory for `data_prefix` and `ann_file`. Defaults to ‘’.
- **data_prefix** (*str | dict*) – Prefix for the data. Defaults to ‘’.
- **extensions** (*Sequence[str]*) – A sequence of allowed extensions. Defaults to (‘.jpg’, ‘.jpeg’, ‘.png’, ‘.ppm’, ‘.bmp’, ‘.pgm’, ‘.tif’).
- **lazy_init** (*bool*) – Whether to load annotation during instantiation. In some cases, such as visualization, only the meta information of the dataset is needed, which is not necessary to load annotation file. `Basedataset` can skip load annotations to save time by set `lazy_init=False`. Defaults to False.
- ****kwargs** – Other keyword arguments in `BaseDataset`.

property img_prefix

The prefix of images.

property CLASSES

Return all categories names.

property class_to_idx

Map mapping class name to class index.

Returns mapping from class name to class index.

Return type dict

_find_samples(*file_backend*)

find samples from `data_prefix`.

load_data_list()

Load image paths and gt_labels.

is_valid_file(*filename: str*) → bool

Check if a file is a valid sample.

get_gt_labels()

Get all ground-truth labels (categories).

Returns categories for all images.

Return type np.ndarray

get_cat_ids(*idx: int*) → List[int]

Get category id by index.

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

Returns Image category of specified index.

Return type cat_ids (List[int])

_compat_classes(*metainfo*, *classes*)

Merge the old style `classes` arguments to `metainfo`.

full_init()

Load annotation file and set `BaseDataset._fully_initialized` to True.

__repr__()

Print the basic information of the dataset.

Returns Formatted string.

Return type str

extra_repr() → List[str]

The extra repr information of the dataset.

class mmedit.datasets.**BasicFramesDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: Optional[str] = None*, *data_prefix: dict = dict(img='')*, *pipeline: List[Union[dict, Callable]] = []*, *test_mode: bool = False*, *filename_tmpl: dict = dict()*, *search_key: Optional[str] = None*, *backend_args: Optional[dict] = None*, *depth: int = 1*, *num_input_frames: Optional[int] = None*, *num_output_frames: Optional[int] = None*, *fixed_seq_len: Optional[int] = None*, *load_frames_list: dict = dict()*, ***kwargs*)

Bases: mmengine.dataset.BaseDataset

BasicFramesDataset for open source projects in OpenMMLab/MMEditing.

This dataset is designed for low-level vision tasks with frames, such as video super-resolution and video frame interpolation.

The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
Case 1 (Vid4):

    calendar 41
    city 34
    foliage 49
    walk 47

Case 2 (REDS):

    000/00000000.png (720, 1280, 3)
    000/00000001.png (720, 1280, 3)

Case 3 (Vimeo90k):

    00001/0266 (256, 448, 3)
    00001/0268 (256, 448, 3)
```

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str, optional*) – The root directory for data_prefix and ann_file. Defaults to None.
- **data_prefix** (*dict, optional*) – Prefix for training data. Defaults to dict(img=”, gt=”).

- **pipeline** (*list, optional*) – Processing pipeline. Defaults to [].
- **test_mode** (*bool, optional*) – `test_mode=True` means in test phase. Defaults to False.
- **filename_tmpl** (*str*) – Template for each filename. Note that the template excludes the file extension. Default: '{}'.
- **search_key** (*str*) – The key used for searching the folder to get data_list. Default: 'gt'.
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.
- **depth** (*int*) – The depth of path. Default: 1
- **num_input_frames** (*None | int*) – Number of input frames. Default: None.
- **num_output_frames** (*None | int*) – Number of output frames. Default: None.
- **fixed_seq_len** (*None | int*) – The fixed sequence length. If None, BasicFramesDataset will obtain the length of each sequence. Default: None.
- **load_frames_list** (*dict*) – Load frames list for each key. Default: dict().

Examples

Assume the file structure as the following:

mmediting (root) ├── mmedit ├── tools ├── configs ├── data │ ├── Vid4 │ │ ├── BIx4 │ │ │ ├── city │ │ │ │ ├── img1.png │ │ ├── GT │ │ │ ├── city │ │ │ │ ├── img1.png │ │ ├── meta_info_Vid4_GT.txt │ ├── places │ │ ├── sequences │ │ ├── 00001 │ │ │ │ ├── 0389 │ │ │ │ │ ├── img1.png │ │ │ │ │ ├── img2.png │ │ │ │ │ ├── img3.png │ │ ├── tri_trainlist.txt

Case 1: Loading Vid4 dataset for training a VSR model.

```
dataset = BasicFramesDataset(
    ann_file='meta_info_Vid4_GT.txt',
    metainfo=dict(dataset_type='vid4', task_name='vsr'),
    data_root='data/Vid4',
    data_prefix=dict(img='BIx4', gt='GT'),
    pipeline=[],
    depth=2,
    num_input_frames=5)
```

Case 2: Loading Vimeo90k dataset for training a VFI model.

```
dataset = BasicFramesDataset(
    ann_file='tri_trainlist.txt',
    metainfo=dict(dataset_type='vimeo90k', task_name='vfi'),
    data_root='data/vimeo-triplet',
    data_prefix=dict(img='sequences', gt='sequences'),
    pipeline=[],
    depth=2,
    load_frames_list=dict(
        img=['img1.png', 'img3.png'], gt=['img2.png']))
```

See more details in unittest

tests/test_datasets/test_base_frames_dataset.py TestFramesDatasets().test_version_1_method()

METAINFO

load_data_list() → List[dict]

Load data list from folder or annotation file.

Returns A list of annotation.

Return type list[dict]

_get_path_list()

Get list of paths from annotation file or folder of dataset.

Returns A list of paths.

Return type list[str]

_get_path_list_from_ann()

Get list of paths from annotation file.

Returns A list of paths.

Return type list[str]

_get_path_list_from_folder(*sub_folder=None*, *need_ext=True*, *depth=1*)

Get list of paths from folder.

Parameters

- **sub_folder** (*None | str*) – The path of sub_folder. Default: None.
- **need_ext** (*bool*) – Whether need ext. Default: True.
- **depth** (*int*) – Residual depth of path, recursively called to `depth == 1`. Default: 1

Returns A list of paths.

Return type list[str]

_set_seq_lens()

Get sequence lengths.

_get_frames_list(*key*, *folder*)

Obtain list of frames.

Parameters

- **key** (*str*) – The key of frames list, e.g. `img`, `gt`.
- **folder** (*str*) – Folder of frames.

Returns The paths list of frames.

Return type list[str]

class mmedit.datasets.**BasicImageDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: Optional[str] = None*, *data_prefix: dict = dict(img='')*, *pipeline: List[Union[dict, Callable]] = []*, *test_mode: bool = False*, *filename_tmpl: dict = dict()*, *search_key: Optional[str] = None*, *backend_args: Optional[dict] = None*, *img_suffix: Optional[Union[str, Tuple[str]]] = IMG_EXTENSIONS*, *recursive: bool = False*, ***kwards*)

Bases: mmengine.dataset.BaseDataset

BasicImageDataset for open source projects in OpenMMLab/MMEditing.

This dataset is designed for low-level vision tasks with image, such as super-resolution and inpainting.

The annotation file is optional.

If use annotation file, the annotation format can be shown as follows.

```
Case 1 (CelebA-HQ):

    000001.png
    000002.png

Case 2 (DIV2K):

    0001_s001.png (480,480,3)
    0001_s002.png (480,480,3)
    0001_s003.png (480,480,3)
    0002_s001.png (480,480,3)
    0002_s002.png (480,480,3)

Case 3 (Vimeo90k):

    00001/0266 (256, 448, 3)
    00001/0268 (256, 448, 3)
```

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str, optional*) – The root directory for data_prefix and ann_file. Defaults to None.
- **data_prefix** (*dict, optional*) – Prefix for training data. Defaults to dict(img=None, ann=None).
- **pipeline** (*list, optional*) – Processing pipeline. Defaults to [].
- **test_mode** (*bool, optional*) – test_mode=True means in test phase. Defaults to False.
- **filename_tmpl** (*dict*) – Template for each filename. Note that the template excludes the file extension. Default: dict().
- **search_key** (*str*) – The key used for searching the folder to get data_list. Default: ‘gt’.
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.
- **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.

Note: Assume the file structure as the following:

```
mmediting (root)
├── mmedit
```

(continues on next page)

(continued from previous page)

```
├── tools
├── configs
├── data
    ├── DIV2K
        ├── DIV2K_train_HR
            ├── image.png
        ├── DIV2K_train_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
                ├── image_x4.png
        ├── DIV2K_valid_HR
        ├── DIV2K_valid_LR_bicubic
            ├── X2
            ├── X3
            ├── X4
    ├── places
        ├── test_set
        ├── train_set
|   |   ├── meta
|   |   |   ├── Places365_train.txt
|   |   |   ├── Places365_val.txt
```

Examples

Case 1: Loading DIV2K dataset for training a SISR model.

```
dataset = BasicImageDataset(
    ann_file='',
    metainfo=dict(
        dataset_type='div2k',
        task_name='sisr'),
    data_root='data/DIV2K',
    data_prefix=dict(
        gt='DIV2K_train_HR', img='DIV2K_train_LR_bicubic/X4'),
    filename_tmpl=dict(img='{}_x4', gt='{}'),
    pipeline=[])
```

Case 2: Loading places dataset for training an inpainting model.

```
dataset = BasicImageDataset(
    ann_file='meta/Places365_train.txt',
    metainfo=dict(
        dataset_type='places365',
        task_name='inpainting'),
    data_root='data/places',
    data_prefix=dict(gt='train_set'),
    pipeline=[])
```

METAINFO

load_data_list() → List[dict]

Load data list from folder or annotation file.

Returns A list of annotation.

Return type list[dict]

_get_path_list()

Get list of paths from annotation file or folder of dataset.

Returns A list of paths.

Return type list[dict]

_get_path_list_from_ann()

Get list of paths from annotation file.

Returns List of paths.

Return type List

_get_path_list_from_folder()

Get list of paths from folder.

Returns List of paths.

Return type List

class mmedit.datasets.CIFAR10(*data_prefix: str*, *test_mode: bool*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *download: bool = True*, ***kwargs*)

Bases: mmedit.datasets.basic_conditional_dataset.BasicConditionalDataset

CIFAR10 Dataset.

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

Parameters

- **data_prefix** (*str*) – Prefix for data.
- **test_mode** (*bool*) – test_mode=True means in test phase. It determines to use the training set or test set.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as categories information. Defaults to None.
- **data_root** (*str*) – The root directory for data_prefix. Defaults to ''.
- **download** (*bool*) – Whether to download the dataset if not exists. Defaults to True.
- ****kwargs** – Other keyword arguments in BaseDataset.

base_folder = cifar-10-batches-py

url = https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

filename = cifar-10-python.tar.gz

tgz_md5 = c58f30108f718f92721af3b95e74349a

train_list = [['data_batch_1', 'c99cafc152244af753f735de768cd75f'],
['data_batch_2',...

test_list = [['test_batch', '40351d587109b95175f43aff81a1287e']]

meta

METAINFO

load_data_list()

Load images and ground truth labels.

_load_meta()

Load categories information from metafile.

_check_integrity()

Check the integrity of data files.

extra_repr() → List[str]

The extra repr information of the dataset.

class mmedit.datasets.**AdobeComp1kDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: dict = dict(img_path='')*, *filter_cfg: Optional[dict] = None*, *indices: Optional[Union[int, Sequence[int]]] = None*, *serialize_data: bool = True*, *pipeline: List[Union[dict, Callable]] = []*, *test_mode: bool = False*, *lazy_init: bool = False*, *max_refetch: int = 1000*)

Bases: mmengine.dataset.BaseDataset

Adobe composition-1k dataset.

The dataset loads (alpha, fg, bg) data and apply specified transforms to the data. You could specify whether composite merged image online or load composited merged image in pipeline.

Example for online comp-1k dataset:

```
[
    {
        "alpha": 'alpha/000.png',
        "fg": 'fg/000.png',
        "bg": 'bg/000.png'
    },
    {
        "alpha": 'alpha/001.png',
        "fg": 'fg/001.png',
        "bg": 'bg/001.png'
    },
]
```

Example for offline comp-1k dataset:

```
[
    {
        "alpha": 'alpha/000.png',
        "merged": 'merged/000.png',
        "fg": 'fg/000.png',
        "bg": 'bg/000.png'
    },
    {
        "alpha": 'alpha/001.png',
```

(continues on next page)

(continued from previous page)

```
        "merged": 'merged/001.png',
        "fg": 'fg/001.png',
        "bg": 'bg/001.png'
    },
]
```

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ‘’.
- **data_root** (*str, optional*) – The root directory for `data_prefix` and `ann_file`. Defaults to None.
- **pipeline** (*list, optional*) – Processing pipeline. Defaults to [].
- **test_mode** (*bool, optional*) – `test_mode=True` means in test phase. Defaults to False.
- ****kwargs** – Other arguments passed to `mmengine.dataset.BaseDataset`.

Examples

See unit-tests TODO: Move some codes in unittest here

METAINFO

load_data_list() → List[dict]

Load annotations from an annotation file named as `self.ann_file`

In order to be compoatible to both new and old annotation format, we copy implementations from mmengine and do some modificatoins.

Returns A list of annotation.

Return type list[dict]

parse_data_info(*raw_data_info: dict*) → Union[dict, List[dict]]

Join data_root to each path in data_info.

class mmedit.datasets.**GrowScaleImgDataset**(*data_roots: dict*, *pipeline*, *len_per_stage=int(1000000.0)*, *gpu_samples_per_scale=None*, *gpu_samples_base=32*, *io_backend: Optional[str] = None*, *file_lists: Optional[Union[str, dict]] = None*, *test_mode=False*)

Bases: `mmengine.dataset.BaseDataset`

Grow Scale Unconditional Image Dataset.

This dataset is similar with `UnconditionalImageDataset`, but offer more dynamic functionalities for the supporting complex algorithms, like PGGAN.

Highlight functionalities:

1. Support growing scale dataset. The motivation is to decrease data pre-processing load in CPU. In this dataset, you can provide `imgs_roots` like:

```
{'64': 'path_to_64x64_imgs',
 '512': 'path_to_512x512_imgs'}
```

Then, in training scales lower than 64x64, this dataset will set `self.imgs_root` as ‘path_to_64x64_imgs’;

2. Offer samples_per_gpu according to different scales. In this dataset, self.samples_per_gpu will help runner to know the updated batch size.

Basically, This dataset contains raw images for training unconditional GANs. Given a root dir, we will recursively find all images in this root. The transformation on data is defined by the pipeline.

Parameters

- **imgs_root** (*str*) – Root path for unconditional images.
- **pipeline** (*list[dict | callable]*) – A sequence of data transforms.
- **len_per_stage** (*int, optional*) – The length of dataset for each scale. This args change the length dataset by concatenating or extracting subset. If given a value less than 0., the original length will be kept. Defaults to 1e6.
- **gpu_samples_per_scale** (*dict | None, optional*) – Dict contains samples_per_gpu for each scale. For example, {'32': 4} will set the scale of 32 with samples_per_gpu=4, despite other scale with samples_per_gpu=self.gpu_samples_base.
- **gpu_samples_base** (*int, optional*) – Set default samples_per_gpu for each scale. Defaults to 32.
- **io_backend** (*str, optional*) – The storage backend type. Options are "disk", "ceph", "memcached", "lmdb", "http" and "petrel". Default: None.
- **test_mode** (*bool, optional*) – If True, the dataset will work in test mode. Otherwise, in train mode. Default to False.

_VALID_IMG_SUFFIX = ('.jpg', '.png', '.jpeg', '.JPEG')

load_data_list()

Load annotations.

update_annotations(*curr_scale*)

Update annotations.

Parameters **curr_scale** (*int*) – Current image scale.

Returns Whether to update.

Return type bool

concat_imgs_list_to(*num*)

Concat image list to specified length.

Parameters **num** (*int*) – The length of the concatenated image list.

prepare_train_data(*idx*)

Prepare training data.

Parameters **idx** (*int*) – Index of current batch.

Returns Prepared training data batch.

Return type dict

prepare_test_data(*idx*)

Prepare testing data.

Parameters **idx** (*int*) – Index of current batch.

Returns Prepared training data batch.

Return type dict

__getitem__(*idx*)

Get the idx-th image and data information of dataset after `self.pipeline`, and `full_init` will be called if the dataset has not been fully initialized.

During training phase, if `self.pipeline` get `None`, `self._rand_another` will be called until a valid image is fetched or

the maximum limit of refetech is reached.

Parameters **idx** (*int*) – The index of self.data_list.

Returns The idx-th image and data information of dataset after `self.pipeline`.

Return type dict

__repr__()

Return repr(self).

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

Bases: `mmedit.datasets.basic_conditional_dataset.BasicConditionalDataset`

ImageNet Dataset.

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

Parameters

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

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

METAINFO

class mmedit.datasets.**MSCoCoDataset**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *drop_caption_rate=0.0*, *phase='train'*, *year=2014*, *data_prefix: Union[str, dict] = ''*, *extensions: Sequence[str] = ('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif')*, *lazy_init: bool = False*, *classes: Union[str, Sequence[str], None] = None*, ***kwargs*)

Bases: `mmedit.datasets.basic_conditional_dataset.BasicConditionalDataset`

MSCoCo 2014 dataset.

Parameters

- **ann_file** (*str*) – Annotation file path. Defaults to ''.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str*) – The root directory for `data_prefix` and `ann_file`. Defaults to ''.

- **drop_caption_rate** (*float, optional*) – Rate of dropping caption, used for training. Defaults to 0.0.
- **phase** (*str, optional*) – Subdataset used for certain phase, can be set to *train*, *test* and *val*. Defaults to 'train'.
- **year** (*int, optional*) – Version of CoCo dataset, can be set to 2014 and 2017. Defaults to 2014.
- **data_prefix** (*str | dict*) – Prefix for the data. Defaults to ''.
- **extensions** (*Sequence[str]*) – A sequence of allowed extensions. Defaults to ('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif').
- **lazy_init** (*bool*) – Whether to load annotation during instantiation. In some cases, such as visualization, only the meta information of the dataset is needed, which is not necessary to load annotation file. `Basedataset` can skip load annotations to save time by set `lazy_init=False`. Defaults to False.
- ****kwargs** – Other keyword arguments in `BaseDataset`.

METAINFO

load_data_list()

Load image paths and gt_labels.

class mmedit.datasets.**PairedImageDataset**(*data_root*, *pipeline*, *io_backend: Optional[str] = None*, *test_mode=False*, *test_dir='test'*)

Bases: `mmengine.dataset.BaseDataset`

General paired image folder dataset for image generation.

It assumes that the training directory is '/path/to/data/train'. During test time, the directory is '/path/to/data/test'. '/path/to/data' can be initialized by args 'dataroot'. Each sample contains a pair of images concatenated in the w dimension (A|B).

Parameters

- **dataroot** (str | `Path`) – Path to the folder root of paired images.
- **pipeline** (*List[dict | callable]*) – A sequence of data transformations.
- **test_mode** (*bool*) – Store *True* when building test dataset. Default: *False*.
- **test_dir** (*str*) – Subfolder of dataroot which contain test images. Default: 'test'.

load_data_list()

Load paired image paths.

Returns List that contains paired image paths.

Return type list[dict]

scan_folder(*path*)

Obtain image path list (including sub-folders) from a given folder.

Parameters **path** (str | `Path`) – Folder path.

Returns Image list obtained from the given folder.

Return type list[str]

class mmedit.datasets.**SinGANDataset**(*data_root*, *min_size*, *max_size*, *scale_factor_init*, *pipeline*, *num_samples=- 1*)

Bases: mmengine.dataset.BaseDataset

SinGAN Dataset.

In this dataset, we create an image pyramid and save it in the cache.

Parameters

- **img_path** (*str*) – Path to the single image file.
- **min_size** (*int*) – Min size of the image pyramid. Here, the number will be set to the min(H, W).
- **max_size** (*int*) – Max size of the image pyramid. Here, the number will be set to the max(H, W).
- **scale_factor_init** (*float*) – Rescale factor. Note that the actual factor we use may be a little bit different from this value.
- **num_samples** (*int, optional*) – The number of samples (length) in this dataset. Defaults to -1.

full_init()

Skip the full init process for SinGANDataset.

load_data_list(*min_size*, *max_size*, *scale_factor_init*)

Load annatations for SinGAN Dataset.

Parameters

- **min_size** (*int*) – The minimum size for the image pyramid.
- **max_size** (*int*) – The maximum size for the image pyramid.
- **scale_factor_init** (*float*) – The initial scale factor.

__getitem__(*index*)

Get *:attr:self.data_dict*. For SinGAN, we use single image with different resolution to train the model.

Parameters **idx** (*int*) – This will be ignored in *SinGANDataset*.

Returns Dict contains input image in different resolution. self.pipeline.

Return type dict

__len__()

Get the length of filtered dataset and automatically call full_init if the dataset has not been fully init.

Returns The length of filtered dataset.

Return type int

class mmedit.datasets.**UnpairedImageDataset**(*data_root*, *pipeline*, *io_backend: Optional[str] = None*, *test_mode=False*, *domain_a='A'*, *domain_b='B'*)

Bases: mmengine.dataset.BaseDataset

General unpaired image folder dataset for image generation.

It assumes that the training directory of images from domain A is '/path/to/data/trainA', and that from domain B is '/path/to/data/trainB', respectively. '/path/to/data' can be initialized by args 'dataroot'. During test time, the directory is '/path/to/data/testA' and '/path/to/data/testB', respectively.

Parameters

- **dataroot** (str | Path) – Path to the folder root of unpaired images.
- **pipeline** (*List[dict | callable]*) – A sequence of data transformations.
- **io_backend** (*str, optional*) – The storage backend type. Options are "disk", "ceph", "memcached", "lmdb", "http" and "petrel". Default: None.
- **test_mode** (*bool*) – Store *True* when building test dataset. Default: *False*.
- **domain_a** (*str, optional*) – Domain of images in trainA / testA. Defaults to 'A'.
- **domain_b** (*str, optional*) – Domain of images in trainB / testB. Defaults to 'B'.

load_data_list()

Load the data list.

Returns The data info list of source and target domain.

Return type list

_load_domain_data_list(*dataroot*)

Load unpaired image paths of one domain.

Parameters **dataroot** (*str*) – Path to the folder root for unpaired images of one domain.

Returns List that contains unpaired image paths of one domain.

Return type list[dict]

get_data_info(*idx*) → dict

Get annotation by index and automatically call `full_init` if the dataset has not been fully initialized.

Parameters **idx** (*int*) – The index of data.

Returns The idx-th annotation of the dataset.

Return type dict

__len__()

The length of the dataset.

scan_folder(*path*)

Obtain image path list (including sub-folders) from a given folder.

Parameters **path** (str | Path) – Folder path.

Returns Image list obtained from the given folder.

Return type list[str]

1.41 mmedit.datasets.transforms

1.41.1 Package Contents

Classes

GenerateSeg	Generate segmentation mask from alpha matte.
GenerateSoftSeg	Generate soft segmentation mask from input segmentation mask.

continues on next page

Table 1 – continued from previous page

MirrorSequence	Extend short sequences (e.g. Vimeo-90K) by mirroring the sequences.
TemporalReverse	Reverse frame lists for temporal augmentation.
BinarizeImage	Binarize image.
Clip	Clip the pixels.
ColorJitter	An interface for torch color jitter so that it can be invoked in
RandomAffine	Apply random affine to input images.
RandomMaskDilation	Randomly dilate binary masks.
UnsharpMasking	Apply unsharp masking to an image or a sequence of images.
Flip	Flip the input data with a probability.
NumpyPad	Numpy Padding.
RandomRotation	Rotate the image by a randomly-chosen angle, measured in degree.
RandomTransposeHW	Randomly transpose images in H and W dimensions with a probability.
Resize	Resize data to a specific size for training or resize the images to fit
CenterCropLongEdge	Center crop the given image by the long edge.
Crop	Crop data to specific size for training.
CropAroundCenter	Randomly crop the images around unknown area in the center 1/4 images.
CropAroundFg	Crop around the whole foreground in the segmentation mask.
CropAroundUnknown	Crop around unknown area with a randomly selected scale.
CropLike	Crop/pad the image in the target_key according to the size of image in
FixedCrop	Crop paired data (at a specific position) to specific size for training.
InstanceCrop	Use maskrcnn to detect instances on image.
ModCrop	Mod crop images, used during testing.
PairedRandomCrop	Paried random crop.
RandomCropLongEdge	Random crop the given image by the long edge.
RandomResizedCrop	Crop data to random size and aspect ratio.
CompositeFg	Composite foreground with a random foreground.
MergeFgAndBg	Composite foreground image and background image with alpha.
PerturbBg	Randomly add gaussian noise or gamma change to background image.
RandomJitter	Randomly jitter the foreground in hsv space.
RandomLoadResizeBg	Randomly load a background image and resize it.
PackEditInputs	Pack the inputs data for SR, VFI, matting and inpainting.
ToTensor	Convert some values in results dict to *torch.Tensor* type in data
GenerateCoordinateAndCell	Generate coordinate and cell. Generate coordinate from the desired size
GenerateFacialHeatmap	Generate heatmap from keypoint.
GenerateFrameIndices	Generate frame index for REDS datasets. It also performs temporal

continues on next page

Table 1 – continued from previous page

GenerateFrameIndiceswithPadding	Generate frame index with padding for REDS dataset and Vid4 dataset
GenerateSegmentIndices	Generate frame indices for a segment. It also performs temporal
GetMaskedImage	Get masked image.
GetSpatialDiscountMask	Get spatial discounting mask constant.
LoadImageFromFile	Load a single image or image frames from corresponding paths. Required
LoadMask	Load Mask for multiple types.
LoadPairedImageFromFile	Load a pair of images from file.
MATLABLikeResize	Resize the input image using MATLAB-like downsampling.
Normalize	Normalize images with the given mean and std value.
RescaleToZeroOne	Transform the images into a range between 0 and 1.
DegradationsWithShuffle	Apply random degradations to input, with degradations being shuffled.
RandomBlur	Apply random blur to the input.
RandomJPEGCompression	Apply random JPEG compression to the input.
RandomNoise	Apply random noise to the input.
RandomResize	Randomly resize the input.
RandomVideoCompression	Apply random video compression to the input.
RandomDownSampling	Generate LQ image from GT (and crop), which will randomly pick a scale.
FormatTrimap	Convert trimap (tensor) to one-hot representation.
GenerateTrimap	Using random erode/dilate to generate trimap from alpha matte.
GenerateTrimapWithDistTransform	Generate trimap with distance transform function.
TransformTrimap	Transform trimap into two-channel and six-channel.
CopyValues	Copy the value of source keys to destination keys.
SetValues	Set value to destination keys.

class mmedit.datasets.transforms.**GenerateSeg**(*kernel_size=5*, *erode_iter_range=(10, 20)*, *dilate_iter_range=(15, 30)*, *num_holes_range=(0, 3)*, *hole_sizes=[(15, 15), (25, 25), (35, 35), (45, 45)]*, *blur_ksizes=[(21, 21), (31, 31), (41, 41)]*)

Bases: mmcv.transforms.BaseTransform

Generate segmentation mask from alpha matte.

Parameters

- **kernel_size** (*int, optional*) – Kernel size for both erosion and dilation. The kernel will have the same height and width. Defaults to 5.
- **erode_iter_range** (*tuple, optional*) – Iteration of erosion. Defaults to (10, 20).
- **dilate_iter_range** (*tuple, optional*) – Iteration of dilation. Defaults to (15, 30).
- **num_holes_range** (*tuple, optional*) – Range of number of holes to randomly select from. Defaults to (0, 3).
- **hole_sizes** (*list, optional*) – List of (h, w) to be selected as the size of the rectangle hole. Defaults to [(15, 15), (25, 25), (35, 35), (45, 45)].
- **blur_ksizes** (*list, optional*) – List of (h, w) to be selected as the kernel_size of the gaussian blur. Defaults to [(21, 21), (31, 31), (41, 41)].

static _crop_hole(*img*, *start_point*, *hole_size*)

Create a all-zero rectangle hole in the image.

Parameters

- **img** (*np.ndarray*) – Source image.
- **start_point** (*tuple[int]*) – The top-left point of the rectangle.
- **hole_size** (*tuple[int]*) – The height and width of the rectangle hole.

Returns The cropped image.

Return type np.ndarray

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**GenerateSoftSeg**(*fg_thr=0.2*, *border_width=25*, *erode_ksize=3*, *dilate_ksize=5*, *erode_iter_range=(10, 20)*, *dilate_iter_range=(3, 7)*, *blur_ksizes=[(21, 21), (31, 31), (41, 41)]*)

Bases: mmcv.transforms.BaseTransform

Generate soft segmentation mask from input segmentation mask.

Required key is "seg", added key is "soft_seg".

Parameters

- **fg_thr** (*float, optional*) – Threshold of the foreground in the normalized input segmentation mask. Defaults to 0.2.
- **border_width** (*int, optional*) – Width of border to be padded to the bottom of the mask. Defaults to 25.
- **erode_ksize** (*int, optional*) – Fixed kernel size of the erosion. Defaults to 5.
- **dilate_ksize** (*int, optional*) – Fixed kernel size of the dilation. Defaults to 5.
- **erode_iter_range** (*tuple, optional*) – Iteration of erosion. Defaults to (10, 20).
- **dilate_iter_range** (*tuple, optional*) – Iteration of dilation. Defaults to (3, 7).
- **blur_ksizes** (*list, optional*) – List of (h, w) to be selected as the kernel_size of the gaussian blur. Defaults to [(21, 21), (31, 31), (41, 41)].

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()
Return repr(self).

class mmedit.datasets.transforms.MirrorSequence(*keys*)
Bases: mmcv.transforms.BaseTransform

Extend short sequences (e.g. Vimeo-90K) by mirroring the sequences.

Given a sequence with N frames (x1, …, xN), extend the sequence to (x1, …, xN, xN, …, x1).

Required Keys:

- [KEYS]

Modified Keys:

- [KEYS]

Parameters **keys** (*list[str]*) – The frame lists to be extended.

transform(*results*)
transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()
Return repr(self).

class mmedit.datasets.transforms.TemporalReverse(*keys*, *reverse_ratio=0.5*)
Bases: mmcv.transforms.BaseTransform

Reverse frame lists for temporal augmentation.

Required keys are the keys in attributes “lq” and “gt”, added or modified keys are “lq”, “gt” and “reverse”.

Parameters

- **keys** (*list[str]*) – The frame lists to be reversed.
- **reverse_ratio** (*float*) – The probability to reverse the frame lists. Default: 0.5.

transform(*results*)
transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()
Return repr(self).

class mmedit.datasets.transforms.BinarizeImage(*keys*, *binary_thr*, *a_min=0*, *a_max=1*, *dtype=np.uint8*)
Bases: mmcv.transforms.BaseTransform

Binarize image.

Parameters

- **keys** (*Sequence[str]*) – The images to be binarized.
- **binary_thr** (*float*) – Threshold for binarization.
- **a_min** (*int*) – Lower limits of pixel value.
- **a_max** (*int*) – Upper limits of pixel value.
- **dtype** (*np.dtype*) – Set the data type of the output. Default: np.uint8

_binarize(*img*)

Binarize image.

Parameters **img** (*np.ndarray*) – Input image.

Returns Output image.

Return type img (np.ndarray)

transform(*results*)

The transform function of BinarizeImage.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**Clip**(*keys*, *a_min=0*, *a_max=255*)

Bases: mmcv.transforms.BaseTransform

Clip the pixels.

Modified keys are the attributes specified in "keys".

Parameters

- **keys** (*list[str]*) – The keys whose values are clipped.
- **a_min** (*int*) – Lower limits of pixel value.
- **a_max** (*int*) – Upper limits of pixel value.

_clip(*input_*)

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns

A dict with the values of the specified keys are rounded and clipped.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**ColorJitter**(*keys*, *channel_order='rgb'*, ***kwargs*)

Bases: mmcv.transforms.BaseTransform

An interface for torch color jitter so that it can be invoked in mmediting pipeline.

Randomly change the brightness, contrast and saturation of an image. Modified keys are the attributes specified in "keys".

Required Keys:

- [KEYS]

Modified Keys:

- [KEYS]

Parameters

- **keys** (*list[str]*) – The images to be resized.
- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'rgb'.

Notes

**kwards follows the args list of torchvision.transforms.ColorJitter.

brightness (float or tuple of float (min, max)): How much to jitter brightness. brightness_factor is chosen uniformly from [max(0, 1 - brightness), 1 + brightness] or the given [min, max]. Should be non negative numbers.

contrast (float or tuple of float (min, max)): How much to jitter contrast. contrast_factor is chosen uniformly from [max(0, 1 - contrast), 1 + contrast] or the given [min, max]. Should be non negative numbers.

saturation (float or tuple of float (min, max)): How much to jitter saturation. saturation_factor is chosen uniformly from [max(0, 1 - saturation), 1 + saturation] or the given [min, max]. Should be non negative numbers.

hue (float or tuple of float (min, max)): How much to jitter hue. hue_factor is chosen uniformly from [-hue, hue] or the given [min, max]. Should have 0<= hue <= 0.5 or -0.5 <= min <= max <= 0.5.

_color_jitter(*image*, *this_seed*)

Color Jitter Function.

Parameters

- **image** (*np.ndarray*) – Image.
- **this_seed** (*int*) – Seed of torch.

Returns The output image.

Return type image (np.ndarray)

transform(*results: Dict*) → Dict

The transform function of ColorJitter.

Parameters **results** (*dict*) – The result dict.

Returns The result dict.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomAffine**(*keys*, *degrees*, *translate=None*, *scale=None*, *shear=None*, *flip_ratio=None*)

Bases: mmcv.transforms.BaseTransform

Apply random affine to input images.

This class is adopted from https://github.com/pytorch/vision/blob/v0.5.0/torchvision/transforms/ transforms.py#L1015 It should be noted that in https://github.com/Yaoyi-Li/GCA-Matting/blob/master/dataloader/ data_generator.py#L70 random flip is added. See explanation of *flip_ratio* below. Required keys are the keys in attribute "keys", modified keys are keys in attribute "keys".

Parameters

- **keys** (*Sequence[str]*) – The images to be affined.
- **degrees** (*float | tuple[float]*) – Range of degrees to select from. If it is a float instead of a tuple like (min, max), the range of degrees will be (-degrees, +degrees). Set to 0 to deactivate rotations.
- **translate** (*tuple, optional*) – Tuple of maximum absolute fraction for horizontal and vertical translations. For example translate=(a, b), then horizontal shift is randomly sampled in the range -img_width * a < dx < img_width * a and vertical shift is randomly sampled in the range -img_height * b < dy < img_height * b. Default: None.
- **scale** (*tuple, optional*) – Scaling factor interval, e.g (a, b), then scale is randomly sampled from the range a <= scale <= b. Default: None.
- **shear** (*float | tuple[float], optional*) – Range of shear degrees to select from. If shear is a float, a shear parallel to the x axis and a shear parallel to the y axis in the range (-shear, +shear) will be applied. Else if shear is a tuple of 2 values, a x-axis shear and a y-axis shear in (shear[0], shear[1]) will be applied. Default: None.
- **flip_ratio** (*float, optional*) – Probability of the image being flipped. The flips in horizontal direction and vertical direction are independent. The image may be flipped in both directions. Default: None.

static _get_params(*degrees*, *translate*, *scale_ranges*, *shears*, *flip_ratio*, *img_size*)

Get parameters for affine transformation.

Returns Params to be passed to the affine transformation.

Return type paras (tuple)

static _get_inverse_affine_matrix(*center*, *angle*, *translate*, *scale*, *shear*, *flip*)

Helper method to compute inverse matrix for affine transformation.

As it is explained in PIL.Image.rotate, we need compute INVERSE of affine transformation matrix: M = T * C * RSS * C^-1 where T is translation matrix:

[1, 0, tx | 0, 1, ty | 0, 0, 1];

C is translation matrix to keep center: [1, 0, cx | 0, 1, cy | 0, 0, 1];

RSS is rotation with scale and shear matrix.

It is different from the original function in torchvision. 1. The order are changed to flip -> scale -> rotation -> shear. 2. x and y have different scale factors. RSS(shear, a, scale, f) =

[cos(a + shear)*scale_x*f -sin(a + shear)*scale_y 0] [sin(a)*scale_x*f cos(a)*scale_y 0] [0 0 1]

Thus, the inverse is M^-1 = C * RSS^-1 * C^-1 * T^-1.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomMaskDilation**(*keys*, *binary_thr=0.0*, *kernel_min=9*, *kernel_max=49*)

Bases: mmcv.transforms.BaseTransform

Randomly dilate binary masks.

Parameters

- **keys** (*Sequence[str]*) – The images to be resized.
- **binary_thr** (*float*) – Threshold for obtaining binary mask. Default: 0.
- **kernel_min** (*int*) – Min size of dilation kernel. Default: 9.
- **kernel_max** (*int*) – Max size of dilation kernel. Default: 49.

_random_dilate(*img*)

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**UnsharpMasking**(*kernel_size*, *sigma*, *weight*, *threshold*, *keys*)

Bases: mmcv.transforms.BaseTransform

Apply unsharp masking to an image or a sequence of images.

Parameters

- **kernel_size** (*int*) – The kernel_size of the Gaussian kernel.
- **sigma** (*float*) – The standard deviation of the Gaussian.
- **weight** (*float*) – The weight of the "details" in the final output.
- **threshold** (*float*) – Pixel differences larger than this value are regarded as "details".
- **keys** (*list[str]*) – The keys whose values are processed.

Added keys are "xxx_unsharp", where "xxx" are the attributes specified in "keys".

_unsharp_masking(*imgs*)

Unsharp masking function.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**Flip**(*keys*, *flip_ratio=0.5*, *direction='horizontal'*)

Bases: mmcv.transforms.BaseTransform

Flip the input data with a probability.

Reverse the order of elements in the given data with a specific direction. The shape of the data is preserved, but the elements are reordered. Required keys are the keys in attributes "keys", added or modified keys are "flip", "flip_direction" and the keys in attributes "keys". It also supports flipping a list of images with the same flip.

Required Keys:

- [KEYS]

Modified Keys:

- [KEYS]

Parameters

- **keys** (*Union[str, List[str]]*) – The images to be flipped.
- **flip_ratio** (*float*) – The probability to flip the images. Default: 0.5.
- **direction** (*str*) – Flip images horizontally or vertically. Options are "horizontal" | "vertical". Default: "horizontal".

_directions = ['horizontal', 'vertical']

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**NumpyPad**(*keys*, *padding*, ***kwargs*)

Bases: mmcv.transforms.BaseTransform

Numpy Padding.

In this augmentation, numpy padding is adopted to customize padding augmentation. Please carefully read the numpy manual in: https://numpy.org/doc/stable/reference/generated/numpy.pad.html

If you just hope a single dimension to be padded, you must set `padding` like this:

```
padding = ((2, 2), (0, 0), (0, 0))
```

In this case, if you adopt an input with three dimension, only the first dimension will be padded.

Parameters

- **keys** (*Union[str, List[str]]*) – The images to be padded.
- **padding** (*int | tuple(int)*) – Please refer to the args `pad_width` in `numpy.pad`.

transform(*results*)

Call function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__() → str

Return repr(self).

class mmedit.datasets.transforms.**RandomRotation**(*keys*, *degrees*)

Bases: mmcv.transforms.BaseTransform

Rotate the image by a randomly-chosen angle, measured in degree.

Parameters

- **keys** (*list[str]*) – The images to be rotated.
- **degrees** (*tuple[float] | tuple[int] | float | int*) – If it is a tuple, it represents a range (min, max). If it is a float or int, the range is constructed as (-degrees, degrees).

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomTransposeHW**(*keys*, *transpose_ratio=0.5*)

Bases: mmcv.transforms.BaseTransform

Randomly transpose images in H and W dimensions with a probability.

(TransposeHW = horizontal flip + anti-clockwise rotation by 90 degrees) When used with horizontal/vertical flips, it serves as a way of rotation augmentation. It also supports randomly transposing a list of images.

Required keys are the keys in attributes “keys”, added or modified keys are “transpose” and the keys in attributes “keys”.

Parameters

- **keys** (*list[str]*) – The images to be transposed.

- **transpose_ratio** (*float*) – The probability to transpose the images. Default: 0.5.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**Resize**(*keys: Union[str, List[str]] = 'img'*, *scale=None*, *keep_ratio=False*, *size_factor=None*, *max_size=None*, *interpolation='bilinear'*, *backend=None*, *output_keys=None*)

Bases: mmcv.transforms.BaseTransform

Resize data to a specific size for training or resize the images to fit the network input regulation for testing.

When used for resizing images to fit network input regulation, the case is that a network may have several downsample and then upsample operation, then the input height and width should be divisible by the downsample factor of the network. For example, the network would downsample the input for 5 times with stride 2, then the downsample factor is 2^5 = 32 and the height and width should be divisible by 32.

Required keys are the keys in attribute "keys", added or modified keys are "keep_ratio", "scale_factor", "interpolation" and the keys in attribute "keys".

Required Keys:

- Required keys are the keys in attribute "keys"

Modified Keys:

- Modified the keys in attribute "keys" or save as new key ([OUT_KEY])

Added Keys:

- [OUT_KEY]_shape
- keep_ratio
- scale_factor
- interpolation

All keys in "keys" should have the same shape. "test_trans" is used to record the test transformation to align the input's shape.

Parameters

- **keys** (*str | list[str]*) – The image(s) to be resized.
- **scale** (*float | tuple[int]*) – If scale is tuple[int], target spatial size (h, w). Otherwise, target spatial size is scaled by input size. Note that when it is used, *size_factor* and *max_size* are useless. Default: None
- **keep_ratio** (*bool*) – If set to True, images will be resized without changing the aspect ratio. Otherwise, it will resize images to a given size. Default: False. Note that it is used together with *scale*.
- **size_factor** (*int*) – Let the output shape be a multiple of size_factor. Default:None. Note that when it is used, *scale* should be set to None and *keep_ratio* should be set to False.

- **max_size** (*int*) – The maximum size of the longest side of the output. Default:None. Note that it is used together with *size_factor*.
- **interpolation** (*str*) – Algorithm used for interpolation: "nearest" | "bilinear" | "bicubic" | "area" | "lanczos". Default: "bilinear".
- **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.
- **output_keys** (*list[str] | None*) – The resized images. Default: None Note that if it is not *None*, its length should be equal to keys.

_resize(*img*)

Resize function.

Parameters **img** (*np.ndarray*) – Image.

Returns Resized image.

Return type img (np.ndarray)

transform(*results: Dict*) → Dict

Transform function to resize images.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**CenterCropLongEdge**(*keys='img'*)

Bases: mmcv.transforms.BaseTransform

Center crop the given image by the long edge.

Parameters **keys** (*list[str]*) – The images to be cropped.

transform(*results*)

Call function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**Crop**(*keys*, *crop_size*, *random_crop=True*, *is_pad_zeros=False*)

Bases: mmcv.transforms.BaseTransform

Crop data to specific size for training.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped.
- **crop_size** (*Tuple[int]*) – Target spatial size (h, w).

- **random_crop** (*bool*) – If set to True, it will random crop image. Otherwise, it will work as center crop. Default: True.
- **is_pad_zeros** (*bool, optional*) – Whether to pad the image with 0 if crop_size is greater than image size. Default: False.

_crop(*data*)

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**CropAroundCenter**(*crop_size*)

Bases: mmcv.transforms.BaseTransform

Randomly crop the images around unknown area in the center 1/4 images.

This cropping strategy is adopted in GCA matting. The *unknown area* is the same as *semi-transparent area*. https://arxiv.org/pdf/2001.04069.pdf

It retains the center 1/4 images and resizes the images to 'crop_size'. Required keys are "fg", "bg", "trimap" and "alpha", added or modified keys are "crop_bbox", "fg", "bg", "trimap" and "alpha".

Parameters **crop_size** (*int | tuple*) – Desired output size. If int, square crop is applied.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**CropAroundFg**(*keys*, *bd_ratio_range=(0.1, 0.4)*, *test_mode=False*)

Bases: mmcv.transforms.BaseTransform

Crop around the whole foreground in the segmentation mask.

Required keys are "seg" and the keys in argument *keys*. Meanwhile, "seg" must be in argument *keys*. Added or modified keys are "crop_bbox" and the keys in argument *keys*.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped. It must contain 'seg'.
- **bd_ratio_range** (*tuple, optional*) – The range of the boundary (bd) ratio to select from. The boundary ratio is the ratio of the boundary to the minimal bbox that contains the whole foreground given by segmentation. Default to (0.1, 0.4).

- **test_mode** (*bool*) – Whether use test mode. In test mode, the tight crop area of foreground will be extended to the a square. Default to False.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

class mmedit.datasets.transforms.**CropAroundUnknown**(*keys*, *crop_sizes*, *unknown_source='alpha'*, *interpolations='bilinear'*)

Bases: mmcv.transforms.BaseTransform

Crop around unknown area with a randomly selected scale.

Randomly select the w and h from a list of (w, h). Required keys are the keys in argument *keys*, added or modified keys are "crop_bbox" and the keys in argument *keys*. This class assumes value of "alpha" ranges from 0 to 255.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped. It must contain 'alpha'. If unknown_source is set to 'trimap', then it must also contain 'trimap'.
- **crop_sizes** (*list[int | tuple[int]]*) – List of (w, h) to be selected.
- **unknown_source** (*str, optional*) – Unknown area to select from. It must be 'alpha' or 'trimap'. Default to 'alpha'.
- **interpolations** (*str | list[str], optional*) – Interpolation method of mmcv.imresize. The interpolation operation will be applied when image size is smaller than the crop_size. If given as a list of str, it should have the same length as *keys*. Or if given as a str all the keys will be resized with the same method. Default to 'bilinear'.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**CropLike**(*target_key*, *reference_key=None*)

Bases: mmcv.transforms.BaseTransform

Crop/pad the image in the target_key according to the size of image in the reference_key .

Parameters

- **target_key** (*str*) – The key needs to be cropped.
- **reference_key** (*str | None*) – The reference key, need its size. Default: None.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation. Require self.target_key and self.reference_key.

Returns

A dict containing the processed data and information. Modify self.target_key.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**FixedCrop**(*keys*, *crop_size*, *crop_pos=None*)

Bases: mmcv.transforms.BaseTransform

Crop paired data (at a specific position) to specific size for training.

Parameters

- **keys** (*Sequence[str]*) – The images to be cropped.
- **crop_size** (*Tuple[int]*) – Target spatial size (h, w).
- **crop_pos** (*Tuple[int]*) – Specific position (x, y). If set to None, random initialize the position to crop paired data batch. Default: None.

_crop(*data*, *x_offset*, *y_offset*, *crop_w*, *crop_h*)

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**InstanceCrop**(*config_file*, *key='img'*, *box_num_upbound=- 1*, *finesize=256*)

Bases: mmcv.transforms.BaseTransform

Use maskrcnn to detect instances on image.

Mask R-CNN is used to detect the instance on the image pred_bbox is used to segment the instance on the image

Parameters

- **config_file** (*str*) – config file name relative to detectron2's "configs/"
- **key** (*str*) – Unused
- **box_num_upbound** (*int*) – The upper limit on the number of instances in the figure

transform(*results: dict*) → dict

The transform function of InstanceCrop.

Parameters **results** (*dict*) – A dict containing the necessary information and data for Conversion

Returns

A dict containing the processed data and information.

Return type results (dict)

predict_bbox(*image*)

class mmedit.datasets.transforms.**ModCrop**(*key='gt'*)

Bases: mmcv.transforms.BaseTransform

Mod crop images, used during testing.

Required keys are "scale" and "KEY", added or modified keys are "KEY".

Parameters **key** (*str*) – The key of image. Default: 'gt'

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**PairedRandomCrop**(*gt_patch_size*, *lq_key='img'*, *gt_key='gt'*)

Bases: mmcv.transforms.BaseTransform

Paried random crop.

It crops a pair of img and gt images with corresponding locations. It also supports accepting img list and gt list. Required keys are "scale", "lq_key", and "gt_key", added or modified keys are "lq_key" and "gt_key".

Parameters

- **gt_patch_size** (*int*) – cropped gt patch size.
- **lq_key** (*str*) – Key of LQ img. Default: 'img'.
- **gt_key** (*str*) – Key of GT img. Default: 'gt'.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomCropLongEdge**(*keys='img'*)

Bases: mmcv.transforms.BaseTransform

Random crop the given image by the long edge.

Parameters **keys** (*list[str]*) – The images to be cropped.

transform(*results*)

Call function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomResizedCrop**(*keys*, *crop_size*, *scale=(0.08, 1.0)*, *ratio=(3.0 / 4.0, 4.0 / 3.0)*, *interpolation='bilinear'*)

Bases: mmcv.transforms.BaseTransform

Crop data to random size and aspect ratio.

A crop of a random proportion of the original image and a random aspect ratio of the original aspect ratio is made. The cropped image is finally resized to a given size specified by 'crop_size'. Modified keys are the attributes specified in "keys".

This code is partially adopted from torchvision.transforms.RandomResizedCrop: [https://pytorch.org/vision/stable/_modules/torchvision/transforms/ transforms.html#RandomResizedCrop].

Parameters

- **keys** (*list[str]*) – The images to be resized and random-cropped.
- **crop_size** (*int | tuple[int]*) – Target spatial size (h, w).
- **scale** (*tuple[float], optional*) – Range of the proportion of the original image to be cropped. Default: (0.08, 1.0).
- **ratio** (*tuple[float], optional*) – Range of aspect ratio of the crop. Default: (3. / 4., 4. / 3.).
- **interpolation** (*str, optional*) – Algorithm used for interpolation. It can be only either one of the following: "nearest" | "bilinear" | "bicubic" | "area" | "lanczos". Default: "bilinear".

get_params(*data*)

Get parameters for a random sized crop.

Parameters **data** (*np.ndarray*) – Image of type numpy array to be cropped.

Returns A tuple containing the coordinates of the top left corner and the chosen crop size.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**CompositeFg**(*fg_dirs*, *alpha_dirs*, *interpolation='nearest'*)

Bases: mmcv.transforms.BaseTransform

Composite foreground with a random foreground.

This class composites the current training sample with additional data randomly (could be from the same dataset). With probability 0.5, the sample will be composited with a random sample from the specified directory. The composition is performed as:

$$fg_{new} = \alpha_1 * fg_1 + (1 - \alpha_1) * fg_2$$
$$\alpha_{new} = 1 - (1 - \alpha_1) * (1 - \alpha_2)$$

where (fg_1, α_1) is from the current sample and (fg_2, α_2) is the randomly loaded sample. With the above composition, α_{new} is still in *[0, 1]*.

Required keys are "alpha" and "fg". Modified keys are "alpha" and "fg".

Parameters

- **fg_dirs** (*str | list[str]*) – Path of directories to load foreground images from.
- **alpha_dirs** (*str | list[str]*) – Path of directories to load alpha mattes from.
- **interpolation** (*str*) – Interpolation method of *mmcv.imresize* to resize the randomly loaded images. Default: 'nearest'.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

_get_file_list(*fg_dirs*, *alpha_dirs*)

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**MergeFgAndBg**

Bases: mmcv.transforms.BaseTransform

Composite foreground image and background image with alpha.

Required keys are "alpha", "fg" and "bg", added key is "merged".

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__() → str

Return repr(self).

class mmedit.datasets.transforms.**PerturbBg**(*gamma_ratio=0.6*)

Bases: mmcv.transforms.BaseTransform

Randomly add gaussian noise or gamma change to background image.

Required key is "bg", added key is "noisy_bg".

Parameters **gamma_ratio** (*float, optional*) – The probability to use gamma correction instead of gaussian noise. Defaults to 0.6.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomJitter**(*hue_range=40*)

Bases: mmcv.transforms.BaseTransform

Randomly jitter the foreground in hsv space.

The jitter range of hue is adjustable while the jitter ranges of saturation and value are adaptive to the images. Side effect: the "fg" image will be converted to *np.float32*. Required keys are "fg" and "alpha", modified key is "fg".

Parameters **hue_range** (*float | tuple[float]*) – Range of hue jittering. If it is a float instead of a tuple like (min, max), the range of hue jittering will be (-hue_range, +hue_range). Default: 40.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomLoadResizeBg**(*bg_dir*, *flag='color'*, *channel_order='bgr'*)

Bases: mmcv.transforms.BaseTransform

Randomly load a background image and resize it.

Required key is "fg", added key is "bg".

Parameters

- **bg_dir** (*str*) – Path of directory to load background images from.
- **flag** (*str*) – Loading flag for images. Default: 'color'.
- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'bgr'.
- **kwargs** (*dict*) – Args for file client.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**PackEditInputs**(*keys: Tuple[List[str], str, None] = None*, *pack_all: bool = False*)

Bases: mmcv.transforms.base.BaseTransform

Pack the inputs data for SR, VFI, matting and inpainting.

Keys for images include img, gt, ref, mask, gt_heatmap, trimap, gt_alpha, gt_fg, gt_bg. All of them will be packed into data field of EditDataSample.

pack_all (bool): Whether pack all variables in *results* to *inputs* dict. This is useful when keys of the input dict is not fixed. Please be careful when using this function, because we do not Defaults to False.

Others will be packed into metainfo field of EditDataSample.

transform(*results: dict*) → dict

Method to pack the input data.

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

Returns

- 'inputs' (obj:*torch.Tensor*): The forward data of models.
- **'data_samples' (obj:*EditDataSample*): The annotation info of the** sample.

Return type dict

__repr__() → str

Return repr(self).

class mmedit.datasets.transforms.**ToTensor**(*keys*, *to_float32=True*)

Bases: mmcv.transforms.base.BaseTransform

Convert some values in results dict to *torch.Tensor* type in data loader pipeline.

Parameters

- **keys** (*Sequence[str]*) – Required keys to be converted.
- **to_float32** (*bool*) – Whether convert tensors of images to float32. Default: True.

_data_to_tensor(*value*)

Convert the value to tensor.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

`__repr__`()

Return repr(self).

class mmedit.datasets.transforms.**GenerateCoordinateAndCell**(*sample_quantity=None*, *scale=None*, *target_size=None*, *reshape_gt=True*)

Bases: `mmcv.transforms.base.BaseTransform`

Generate coordinate and cell. Generate coordinate from the desired size of SR image.

Train or val:

1. Generate coordinate from GT.

#. Reshape GT image to (HgWg, 3) and transpose to (3, HgWg). where *Hg* and *Wg* represent the height and width of GT.

Test:

1. Generate coordinate from LQ and scale or target_size.
2. Then generate cell from coordinate.

Parameters

- **sample_quantity** (*int | None*) – The quantity of samples in coordinates. To ensure that the GT tensors in a batch have the same dimensions. Default: None.
- **scale** (*float*) – Scale of upsampling. Default: None.
- **target_size** (*tuple[int]*) – Size of target image. Default: None.
- **reshape_gt** (*bool*) – Whether reshape gt to (-1, 3). Default: True If sample_quantity is not None, reshape_gt = True.

The priority of getting 'size of target image' is:

1. results['gt'].shape[-2:]
2. results['lq'].shape[-2:] * scale
3. target_size

`transform`(*results*)

Call function.

Parameters

- **results** (*Require either in*) – A dict containing the necessary information
- **augmentation.** (*and data for*) –
- **results** –
- **'lq'** (*1.*) –
- **'gt'** (*2.*) –
- **None** (*3.*) –
- **and** (*the premise is self.target_size*) –
- **len** (*self.target_size*) –

Returns A dict containing the processed data and information. Reshape 'gt' to (-1, 3) and transpose to (3, -1) if 'gt' in results. Add 'coord' and 'cell'.

Return type dict

`__repr__`()

Return repr(self).

class mmedit.datasets.transforms.**GenerateFacialHeatmap**(*image_key*, *ori_size*, *target_size*, *sigma=1.0*, *use_cache=True*)

Bases: mmcv.transforms.base.BaseTransform

Generate heatmap from keypoint.

Parameters

- **image_key** (*str*) – Key of facial image in dict.
- **ori_size** (*int | Tuple[int]*) – Original image size of keypoint.
- **target_size** (*int | Tuple[int]*) – Target size of heatmap.
- **sigma** (*float*) – Sigma parameter of heatmap. Default: 1.0
- **use_cache** (*bool*) – If True, load all heatmap at once. Default: True.

`transform`(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation. Require keypoint.

Returns

A dict containing the processed data and information. Add 'heatmap'.

Return type dict

`generate_heatmap_from_img`(*image*)

Generate heatmap from img.

Parameters **image** (*np.ndarray*) – Face image.

results: heatmap (np.ndarray): Heatmap the face image.

`_face_alignment_detector`(*image*)

Generate face landmark by face_alignment.

Parameters **image** (*np.ndarray*) – Face image.

Returns Location of landmark.

Return type landmark (Tuple[float])

`_generate_one_heatmap`(*keypoint*)

Generate One Heatmap.

Parameters **keypoint** (*Tuple[float]*) – Location of a landmark.

results: heatmap (np.ndarray): A heatmap of landmark.

`__repr__`()

Return repr(self).

class mmedit.datasets.transforms.**GenerateFrameIndices**(*interval_list*, *frames_per_clip=99*)

Bases: mmcv.transforms.BaseTransform

Generate frame index for REDS datasets. It also performs temporal augmention with random interval.

Required Keys:

- img_path
- gt_path
- key
- num_input_frames

Modified Keys:

- img_path
- gt_path

Added Keys:

- interval
- reverse

Parameters

- **interval_list** (*list[int]*) – Interval list for temporal augmentation. It will randomly pick an interval from interval_list and sample frame index with the interval.
- **frames_per_clip** (*int*) – Number of frames per clips. Default: 99 for REDS dataset.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**GenerateFrameIndiceswithPadding**(*padding*, *filename_tmpl='{:08d}'*)

Bases: mmcv.transforms.BaseTransform

Generate frame index with padding for REDS dataset and Vid4 dataset during testing.

Required Keys:

- img_path
- gt_path
- key
- num_input_frames
- sequence_length

Modified Keys:

- img_path
- gt_path

Parameters **padding** – padding mode, one of 'replicate' | 'reflection' | 'reflection_circle' | 'circle'.

Examples: current_idx = 0, num_input_frames = 5 The generated frame indices under different padding mode:

replicate: [0, 0, 0, 1, 2] reflection: [2, 1, 0, 1, 2] reflection_circle: [4, 3, 0, 1, 2] circle: [3, 4, 0, 1, 2]

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**GenerateSegmentIndices**(*interval_list*, *start_idx=0*, *filename_tmpl='{:08d}.png'*)

Bases: mmcv.transforms.BaseTransform

Generate frame indices for a segment. It also performs temporal augmention with random interval.

Required Keys:

- img_path
- gt_path
- key
- num_input_frames
- sequence_length

Modified Keys:

- img_path
- gt_path

Added Keys:

- interval
- reverse

Parameters

- **interval_list** (*list[int]*) – Interval list for temporal augmentation. It will randomly pick an interval from interval_list and sample frame index with the interval.
- **start_idx** (*int*) – The index corresponds to the first frame in the sequence. Default: 0.
- **filename_tmpl** (*str*) – Template for file name. Default: '{:08d}.png'.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**GetMaskedImage**(*img_key='gt'*, *mask_key='mask'*, *out_key='img'*, *zero_value=127.5*)

Bases: mmcv.transforms.base.BaseTransform

Get masked image.

Parameters

- **img_key** (*str*) – Key for clean image. Default: 'gt'.
- **mask_key** (*str*) – Key for mask image. The mask shape should be (h, w, 1) while '1' indicate holes and '0' indicate valid regions. Default: 'mask'.
- **img_key** – Key for output image. Default: 'img'.
- **zero_value** (*float*) – Pixel value of masked area.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**GetSpatialDiscountMask**(*gamma=0.99*, *beta=1.5*)

Bases: mmcv.transforms.BaseTransform

Get spatial discounting mask constant.

Spatial discounting mask is first introduced in: Generative Image Inpainting with Contextual Attention.

Parameters

- **gamma** (*float, optional*) – Gamma for computing spatial discounting. Defaults to 0.99.
- **beta** (*float, optional*) – Beta for computing spatial discounting. Defaults to 1.5.

spatial_discount_mask(*mask_width*, *mask_height*)

Generate spatial discounting mask constant.

Parameters

- **mask_width** (*int*) – The width of bbox hole.
- **mask_height** (*int*) – The height of bbox height.

Returns Spatial discounting mask.

Return type np.ndarray

`transform`(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

`__repr__`()

Return repr(self).

class mmedit.datasets.transforms.**LoadImageFromFile**(*key: str*, *color_type: str = 'color'*, *channel_order: str = 'bgr'*, *imdecode_backend: Optional[str] = None*, *use_cache: bool = False*, *to_float32: bool = False*, *to_y_channel: bool = False*, *save_original_img: bool = False*, *backend_args: Optional[dict] = None*)

Bases: mmcv.transforms.BaseTransform

Load a single image or image frames from corresponding paths. Required Keys: - [Key]_path

New Keys: - [KEY] - ori_[KEY]_shape - ori_[KEY]

Parameters

- **key** (*str*) – Keys in results to find corresponding path.
- **color_type** (*str*) – The flag argument for :func:mmcv.imfrombytes. Defaults to 'color'.
- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'bgr'.
- **imdecode_backend** (*str*) – The image decoding backend type. The backend argument for :func:mmcv.imfrombytes. See :func:mmcv.imfrombytes for details. candidates are 'cv2', 'turbojpeg', 'pillow', and 'tifffile'. Defaults to None.
- **use_cache** (*bool*) – If True, load all images at once. Default: False.
- **to_float32** (*bool*) – Whether to convert the loaded image to a float32 numpy array. If set to False, the loaded image is an uint8 array. Defaults to False.
- **to_y_channel** (*bool*) – Whether to convert the loaded image to y channel. Only support 'rgb2ycbcr' and 'rgb2ycbcr' Defaults to False.
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.

`transform`(*results: dict*) → dict

Functions to load image or frames.

Parameters **results** (*dict*) – Result dict from :obj:mmcv.BaseDataset.

Returns The dict contains loaded image and meta information.

Return type dict

`_load_image`(*filename*)

Load an image from file.

Parameters **filename** (*str*) – Path of image file.

Returns Image.

Return type np.ndarray

_convert(*img: numpy.ndarray*)

Convert an image to the require format.

Parameters **img** (*np.ndarray*) – The original image.

Returns The converted image.

Return type np.ndarray

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**LoadMask**(*mask_mode='bbox'*, *mask_config=None*)

Bases: mmcv.transforms.BaseTransform

Load Mask for multiple types.

For different types of mask, users need to provide the corresponding config dict.

Example config for bbox:

```
config = dict(img_shape=(256, 256), max_bbox_shape=128)
```

Example config for irregular:

```
config = dict(
    img_shape=(256, 256),
    num_vertices=(4, 12),
    max_angle=4.,
    length_range=(10, 100),
    brush_width=(10, 40),
    area_ratio_range=(0.15, 0.5))
```

Example config for ff:

```
config = dict(
    img_shape=(256, 256),
    num_vertices=(4, 12),
    mean_angle=1.2,
    angle_range=0.4,
    brush_width=(12, 40))
```

Example config for set:

```
config = dict(
    mask_list_file='xxx/xxx/ooxx.txt',
    prefix='/xxx/xxx/ooxx/',
    io_backend='local',
    color_type='unchanged',
    file_client_kwargs=dict()
)

The mask_list_file contains the list of mask file name like this:
    test1.jpeg
    test2.jpeg
    ...
```

(continues on next page)

(continued from previous page)

```
    ...

The prefix gives the data path.
```

Parameters

- **mask_mode** (*str*) – Mask mode in ['bbox', 'irregular', 'ff', 'set', 'file']. Default: 'bbox'. * bbox: square bounding box masks. * irregular: irregular holes. * ff: free-form holes from DeepFillv2. * set: randomly get a mask from a mask set. * file: get mask from 'mask_path' in results.
- **mask_config** (*dict*) – Params for creating masks. Each type of mask needs different configs. Default: None.

_init_info()

_get_random_mask_from_set()

_get_mask_from_file(*path*)

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**LoadPairedImageFromFile**(*key: str*, *domain_a: str = 'A'*, *domain_b: str = 'B'*, *color_type: str = 'color'*, *channel_order: str = 'bgr'*, *imdecode_backend: Optional[str] = None*, *use_cache: bool = False*, *to_float32: bool = False*, *to_y_channel: bool = False*, *save_original_img: bool = False*, *backend_args: Optional[dict] = None*)

Bases: *LoadImageFromFile*

Load a pair of images from file.

Each sample contains a pair of images, which are concatenated in the w dimension (a|b). This is a special loading class for generation paired dataset. It loads a pair of images as the common loader does and crops it into two images with the same shape in different domains.

Required key is "pair_path". Added or modified keys are "pair", "pair_ori_shape", "ori_pair", "img_{domain_a}", "img_{domain_b}", "img_{domain_a}_path", "img_{domain_b}_path", "img_{domain_a}_ori_shape", "img_{domain_b}_ori_shape", "ori_img_{domain_a}" and "ori_img_{domain_b}".

Parameters

- **key** (*str*) – Keys in results to find corresponding path.

- **domain_a** (*str, Optional*) – One of the paired image domain. Defaults to 'A'.
- **domain_b** (*str, Optional*) – The other of the paired image domain. Defaults to 'B'.
- **color_type** (*str*) – The flag argument for :func:mmcv.imfrombytes. Defaults to 'color'.
- **channel_order** (*str*) – Order of channel, candidates are 'bgr' and 'rgb'. Default: 'bgr'.
- **imdecode_backend** (*str*) – The image decoding backend type. The backend argument for :func:mmcv.imfrombytes. See :func:mmcv.imfrombytes for details. candidates are 'cv2', 'turbojpeg', 'pillow', and 'tiffile'. Defaults to None.
- **use_cache** (*bool*) – If True, load all images at once. Default: False.
- **to_float32** (*bool*) – Whether to convert the loaded image to a float32 numpy array. If set to False, the loaded image is an uint8 array. Defaults to False.
- **to_y_channel** (*bool*) – Whether to convert the loaded image to y channel. Only support 'rgb2ycbcr' and 'rgb2ycbcr' Defaults to False.
- **backend_args** (*dict, optional*) – Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.
- **io_backend** (*str, optional*) – io backend where images are store. Defaults to None.

transform(*results: dict*) → dict

Functions to load paired images.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

class mmedit.datasets.transforms.**MATLABLikeResize**(*keys*, *scale=None*, *output_shape=None*, *kernel='bicubic'*, *kernel_width=4.0*)

Bases: mmcv.transforms.BaseTransform

Resize the input image using MATLAB-like downsampling.

Currently support bicubic interpolation only. Note that the output of this function is slightly different from the official MATLAB function.

Required keys are the keys in attribute "keys". Added or modified keys are "scale" and "output_shape", and the keys in attribute "keys".

Parameters

- **keys** (*list[str]*) – A list of keys whose values are modified.
- **scale** (*float | None, optional*) – The scale factor of the resize operation. If None, it will be determined by output_shape. Default: None.
- **output_shape** (*tuple(int) | None, optional*) – The size of the output image. If None, it will be determined by scale. Note that if scale is provided, output_shape will not be used. Default: None.
- **kernel** (*str, optional*) – The kernel for the resize operation. Currently support 'bicubic' only. Default: 'bicubic'.
- **kernel_width** (*float*) – The kernel width. Currently support 4.0 only. Default: 4.0.

_resize(*img*)

resize an image to the require size.

Parameters **img** (*np.ndarray*) – The original image.

Returns The resized image.

Return type output (np.ndarray)

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**Normalize**(*keys*, *mean*, *std*, *to_rgb=False*, *save_original=False*)

Bases: mmcv.transforms.BaseTransform

Normalize images with the given mean and std value.

Required keys are the keys in attribute "keys", added or modified keys are the keys in attribute "keys" and these keys with postfix '_norm_cfg'. It also supports normalizing a list of images.

Parameters

- **keys** (*Sequence[str]*) – The images to be normalized.
- **mean** (*np.ndarray*) – Mean values of different channels.
- **std** (*np.ndarray*) – Std values of different channels.
- **to_rgb** (*bool*) – Whether to convert channels from BGR to RGB. Default: False.
- **save_original** (*bool*) – Whether to save original images. Default: False.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RescaleToZeroOne**(*keys*)

Bases: mmcv.transforms.BaseTransform

Transform the images into a range between 0 and 1.

Required keys are the keys in attribute "keys", added or modified keys are the keys in attribute "keys". It also supports rescaling a list of images.

Parameters **keys** (*Sequence[str]*) – The images to be transformed.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**DegradationsWithShuffle**(*degradations*, *keys*, *shuffle_idx=None*)

Apply random degradations to input, with degradations being shuffled.

Degradation groups are supported. The order of degradations within the same group is preserved. For example, if we have degradations = [a, b, [c, d]] and shuffle_idx = None, then the possible orders are

```
[a, b, [c, d]]
[a, [c, d], b]
[b, a, [c, d]]
[b, [c, d], a]
[[c, d], a, b]
[[c, d], b, a]
```

Modified keys are the attributed specified in "keys".

Parameters

- **degradations** (*list[dict]*) – The list of degradations.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.
- **shuffle_idx** (*list | None, optional*) – The degradations corresponding to these indices are shuffled. If None, all degradations are shuffled. Default: None.

_build_degradations(*degradations*)

__call__(*results*)

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomBlur**(*params*, *keys*)

Apply random blur to the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

get_kernel(*num_kernels: int*)

This is the function to create kernel.

Parameters **num_kernels** (*int*) – the number of kernels

Returns _description_

Return type _type_

_apply_random_blur(*imgs*)

This is the function to apply blur operation on images.

Parameters **imgs** (*Tensor*) – images

Returns Images applied blur

Return type Tensor

__call__(*results*)

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomJPEGCompression**(*params*, *keys*, *color_type='color'*, *bgr2rgb=False*)

Apply random JPEG compression to the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.
- **bgr2rgb** (*str*) – Whether change channel order. Default: False.

_apply_random_compression(*imgs*)

__call__(*results*)

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomNoise**(*params*, *keys*)

Apply random noise to the input.

Currently support Gaussian noise and Poisson noise.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

_apply_gaussian_noise(*imgs*)

This is the function used to apply gaussian noise on images.

Parameters **imgs** (*Tensor*) – images

Returns images applied gaussian noise

Return type Tensor

_apply_poisson_noise(*imgs*)

_apply_random_noise(*imgs*)

This is the function used to apply random noise on images.

Parameters **imgs** (*Tensor*) – training images

Returns _description_

Return type _type_

__call__(*results*)

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomResize**(*params*, *keys*)

Randomly resize the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

_random_resize(*imgs*)

This is the function used to randomly resize images for training augmentation.

Parameters **imgs** (*Tensor*) – training images.

Returns images after radomly resized

Return type Tensor

__call__(*results*)

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomVideoCompression**(*params*, *keys*)

Apply random video compression to the input.

Modified keys are the attributed specified in "keys".

Parameters

- **params** (*dict*) – A dictionary specifying the degradation settings.
- **keys** (*list[str]*) – A list specifying the keys whose values are modified.

_apply_random_compression(*imgs*)

This is the function to apply random compression on images.

Parameters **imgs** (*Tensor*) – training images

Returns images after randomly compressed

Return type Tensor

__call__(*results*)

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**RandomDownSampling**(*scale_min=1.0*, *scale_max=4.0*, *patch_size=None*, *interpolation='bicubic'*, *backend='pillow'*)

Bases: mmcv.transforms.BaseTransform

Generate LQ image from GT (and crop), which will randomly pick a scale.

Parameters

- **scale_min** (*float*) – The minimum of upsampling scale, inclusive. Default: 1.0.
- **scale_max** (*float*) – The maximum of upsampling scale, exclusive. Default: 4.0.
- **patch_size** (*int*) – The cropped lr patch size. Default: None, means no crop.
- **interpolation** (*str*) – Interpolation method, accepted values are "nearest", "bilinear", "bicubic", "area", "lanczos" for 'cv2' backend, "nearest", "bilinear", "bicubic", "box", "lanczos", "hamming" for 'pillow' backend. Default: "bicubic".
- **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: "pillow".
- **[scale_min** (*Scale will be picked in the range of*) –
- **scale_max).** –

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation. 'gt' is required.

Returns

A dict containing the processed data and information. modified 'gt', supplement 'lq' and 'scale' to keys.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**FormatTrimap**(*to_onehot=False*)

Bases: `mmcv.transforms.BaseTransform`

Convert trimap (tensor) to one-hot representation.

It transforms the trimap label from (0, 128, 255) to (0, 1, 2). If `to_onehot` is set to True, the trimap will convert to one-hot tensor of shape (3, H, W). Required key is "trimap", added or modified key are "trimap" and "format_trimap_to_onehot".

Parameters **to_onehot** (*bool*) – whether convert trimap to one-hot tensor. Default: `False`.

transform(*results*)

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**GenerateTrimap**(*kernel_size*, *iterations=1*, *random=True*)

Bases: `mmcv.transforms.BaseTransform`

Using random erode/dilate to generate trimap from alpha matte.

Required key is "alpha", added key is "trimap".

Parameters

- **kernel_size** (*int | tuple[int]*) – The range of random kernel_size of erode/dilate; int indicates a fixed kernel_size. If *random* is set to False and kernel_size is a tuple of length 2, then it will be interpreted as (erode kernel_size, dilate kernel_size). It should be noted that the kernel of the erosion and dilation has the same height and width.
- **iterations** (*int | tuple[int], optional*) – The range of random iterations of erode/dilate; int indicates a fixed iterations. If *random* is set to False and iterations is a tuple of length 2, then it will be interpreted as (erode iterations, dilate iterations). Default to 1.
- **random** (*bool, optional*) – Whether use random kernel_size and iterations when generating trimap. See *kernel_size* and *iterations* for more information. Default to True.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**GenerateTrimapWithDistTransform**(*dist_thr=20*, *random=True*)

Bases: mmcv.transforms.BaseTransform

Generate trimap with distance transform function.

Parameters

- **dist_thr** (*int, optional*) – Distance threshold. Area with alpha value between (0, 255) will be considered as initial unknown area. Then area with distance to unknown area smaller than the distance threshold will also be consider as unknown area. Defaults to 20.
- **random** (*bool, optional*) – If True, use random distance threshold from [1, dist_thr). If False, use *dist_thr* as the distance threshold directly. Defaults to True.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**TransformTrimap**

Bases: mmcv.transforms.BaseTransform

Transform trimap into two-channel and six-channel.

This class will generate a two-channel trimap composed of definite foreground and background masks and encode it into a six-channel trimap using Gaussian blurs of the generated two-channel trimap at three different scales. The transformed trimap has 6 channels.

Required key is "trimap", added key is "transformed_trimap" and "two_channel_trimap".

Adopted from the following repository: https://github.com/MarcoForte/FBA_Matting/blob/master/networks/transforms.py.

transform(*results: dict*) → dict

Transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict containing the processed data and information.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**CopyValues**(*src_keys*, *dst_keys*)

Bases: mmcv.transforms.BaseTransform

Copy the value of source keys to destination keys.

TODO Change to dict(dst=src)

It does the following: results[dst_key] = results[src_key] for (src_key, dst_key) in zip(src_keys, dst_keys).

Added keys are the keys in the attribute "dst_keys".

Required Keys:

- [SRC_KEYS]

Added Keys:

- [DST_KEYS]

Parameters

- **src_keys** (*list[str]*) – The source keys.
- **dst_keys** (*list[str]*) – The destination keys.

transform(*results*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict with a key added/modified.

Return type dict

__repr__()

Return repr(self).

class mmedit.datasets.transforms.**SetValues**(*dictionary*)

Bases: mmcv.transforms.BaseTransform

Set value to destination keys.

It does the following: results[key] = value

Added keys are the keys in the dictionary.

Required Keys:

- None

Added or Modifyed Keys:

- keys in the dictionary

  Parameters **dictionary** (*dict*) – The dictionary to update.

transform(*results: Dict*)

transform function.

Parameters **results** (*dict*) – A dict containing the necessary information and data for augmentation.

Returns A dict with a key added/modified.

Return type dict

__repr__()

Return repr(self).

1.42 mmedit.evaluation

1.42.1 Package Contents

Classes

GenEvaluator	Evaluator for generative models. Unlike high-level vision tasks, metrics
MAE	Mean Absolute Error metric for image.
MSE	Mean Squared Error metric for image.
NIQE	Calculate NIQE (Natural Image Quality Evaluator) metric.
PSNR	Peak Signal-to-Noise Ratio.
SAD	Sum of Absolute Differences metric for image matting.
SNR	Signal-to-Noise Ratio.
SSIM	Calculate SSIM (structural similarity).
ConnectivityError	Connectivity error for evaluating alpha matte prediction.
Equivariance	Metric for generative metrics. Except for the preparation phase
FrechetInceptionDistance	FID metric. In this metric, we calculate the distance between real
GradientError	Gradient error for evaluating alpha matte prediction.
InceptionScore	IS (Inception Score) metric. The images are split into groups, and the
MattingMSE	Mean Squared Error metric for image matting.
MultiScaleStructureSimilarity	MS-SSIM (Multi-Scale Structure Similarity) metric.
PerceptualPathLength	Perceptual path length.
PrecisionAndRecall	Improved Precision and recall metric.
SlicedWassersteinDistance	SWD (Sliced Wasserstein distance) metric. We calculate the SWD of two
TransFID	FID metric. In this metric, we calculate the distance between real
TransIS	IS (Inception Score) metric. The images are split into groups, and the

Functions

gauss_gradient(img, sigma)	Gaussian gradient.

class mmedit.evaluation.**GenEvaluator**(*metrics: Union[dict, mmengine.evaluator.BaseMetric, Sequence]*)

Bases: mmengine.evaluator.Evaluator

Evaluator for generative models. Unlike high-level vision tasks, metrics for generative models have various input types. For example, Inception Score (IS, *InceptionScore*) only needs to take fake images as input. However, Frechet Inception Distance (FID, *FrechetInceptionDistance*) needs to take both real images and fake images as input, and the numbers of real images and fake images can be set arbitrarily. For Perceptual path length (PPL, *PerceptualPathLength*), generator need to sample images along a latent path.

In order to be compatible with different metrics, we designed two critical functions, *prepare_metrics()* and *prepare_samplers()* to support those requirements.

- *prepare_metrics()* set the image images' color order and pass the dataloader to all metrics. Therefore metrics need pre-processing to prepare the corresponding feature.
- *prepare_samplers()* pass the dataloader and model to the metrics, and get the corresponding sampler of each kind of metrics. Metrics with same sample mode can share the sampler.

The whole evaluation process can be found in *mmedit.engine.runner.GenValLoop.run()* and *mmedit.engine.runner.GenTestLoop.run()*.

Parameters **metrics** (*dict or BaseMetric or Sequence*) – The config of metrics.

prepare_metrics(*module: mmengine.model.BaseModel*, *dataloader: torch.utils.data.dataloader.DataLoader*)

Prepare for metrics before evaluation starts. Some metrics use pretrained model to extract feature. Some metrics use pretrained model to extract feature and input channel order may vary among those models. Therefore, we first parse the output color order from data preprocessor and set the color order for each metric. Then we pass the dataloader to each metrics to prepare pre-calculated items. (e.g. inception feature of the real images). If metric has no pre-calculated items, `metric.prepare()` will be ignored. Once the function has been called, `self.is_ready` will be set as *True*. If `self.is_ready` is *True*, this function will directly return to avoid duplicate computation.

Parameters

- **module** (*BaseModel*) – Model to evaluate.
- **dataloader** (*DataLoader*) – The dataloader for real images.

static _cal_metric_hash(*metric: mmedit.evaluation.metrics.base_gen_metric.GenMetric*)

Calculate a unique hash value based on the *SAMPLER_MODE* and *sample_model*.

prepare_samplers(*module: mmengine.model.BaseModel*, *dataloader: torch.utils.data.dataloader.DataLoader*) → List[Tuple[List[mmengine.evaluator.BaseMetric], Iterator]]

Prepare for the sampler for metrics whose sampling mode are different. For generative models, different metric need image generated with different inputs. For example, FID, KID and IS need images generated with random noise, and PPL need paired images on the specific noise interpolation path. Therefore, we first group metrics with respect to their sampler's mode (refers to :attr:~`GenMetrics.SAMPLER_MODE`), and build a shared sampler for each metric group. To be noted that, the length of the shared sampler depends on the metric of the most images required in each group.

Parameters

- **module** (*BaseModel*) – Model to evaluate. Some metrics (e.g. PPL) require *module* in their sampler.
- **dataloader** (*DataLoader*) – The dataloader for real image.

Returns

A list of "metrics-shared sampler" pair.

Return type List[Tuple[List[BaseMetric], Iterator]]

process(*data_samples: Sequence[mmedit.structures.EditDataSample]*, *data_batch: Optional[Any]*, *metrics: Sequence[mmengine.evaluator.BaseMetric]*) → None

Pass *data_batch* from dataloader and *predictions* (generated results) to corresponding *metrics*.

Parameters

- **data_samples** (*Sequence[EditDataSample]*) – A batch of generated results from model.
- **data_batch** (*Optional[Any]*) – A batch of data from the metrics specific sampler or the dataloader.
- **metrics** (*Optional[Sequence[BaseMetric]]*) – Metrics to evaluate.

evaluate() → dict

Invoke evaluate method of each metric and collect the metrics dictionary. Different from *Evaluator.evaluate*, this function does not take *size* as input, and elements in *self.metrics* will call their own *evaluate* method to calculate the metric.

Returns

Evaluation results of all metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type dict

mmedit.evaluation.**gauss_gradient**(*img*, *sigma*)

Gaussian gradient.

From https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/ submissions/8060/versions/2/previews/gaussgradient/gaussgradient.m/ index.html

Parameters

- **img** (*np.ndarray*) – Input image.
- **sigma** (*float*) – Standard deviation of the gaussian kernel.

Returns Gaussian gradient of input *img*.

Return type np.ndarray

class mmedit.evaluation.**MAE**(*gt_key: str = 'gt_img'*, *pred_key: str = 'pred_img'*, *mask_key: Optional[str] = None*, *scaling=1*, *device='cpu'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*)

Bases: mmedit.evaluation.metrics.base_sample_wise_metric.BaseSampleWiseMetric

Mean Absolute Error metric for image.

mean(abs(a-b))

Parameters

- **gt_key** (*str*) – Key of ground-truth. Default: 'gt_img'
- **pred_key** (*str*) – Key of prediction. Default: 'pred_img'
- **mask_key** (*str, optional*) – Key of mask, if mask_key is None, calculate all regions. Default: None
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Default: None

Metrics:

- MAE (float): Mean of Absolute Error

metric = MAE

process_image(*gt*, *pred*, *mask*)

Process an image.

Parameters

- **gt** (*Tensor | np.ndarray*) – GT image.
- **pred** (*Tensor | np.ndarray*) – Pred image.
- **mask** (*Tensor | np.ndarray*) – Mask of evaluation.

Returns MAE result.

Return type result (np.ndarray)

class mmedit.evaluation.**MSE**(*gt_key: str = 'gt_img'*, *pred_key: str = 'pred_img'*, *mask_key: Optional[str] = None*, *scaling=1*, *device='cpu'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*)

Bases: mmedit.evaluation.metrics.base_sample_wise_metric.BaseSampleWiseMetric

Mean Squared Error metric for image.

mean((a-b)^2)

Parameters

- **gt_key** (*str*) – Key of ground-truth. Default: 'gt_img'
- **pred_key** (*str*) – Key of prediction. Default: 'pred_img'
- **mask_key** (*str, optional*) – Key of mask, if mask_key is None, calculate all regions. Default: None
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Default: None

Metrics:

- MSE (float): Mean of Squared Error

metric = MSE

process_image(*gt*, *pred*, *mask*)

Process an image.

Parameters

- **gt** (*Torch | np.ndarray*) – GT image.
- **pred** (*Torch | np.ndarray*) – Pred image.
- **mask** (*Torch | np.ndarray*) – Mask of evaluation.

Returns MSE result.

Return type result (np.ndarray)

class mmedit.evaluation.**NIQE**(*key: str = 'pred_img'*, *is_predicted: bool = True*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *crop_border=0*, *input_order='HWC'*, *convert_to='gray'*)

Bases: mmedit.evaluation.metrics.base_sample_wise_metric.BaseSampleWiseMetric

Calculate NIQE (Natural Image Quality Evaluator) metric.

Ref: Making a "Completely Blind" Image Quality Analyzer. This implementation could produce almost the same results as the official MATLAB codes: http://live.ece.utexas.edu/research/quality/niqe_release.zip

We use the official params estimated from the pristine dataset. We use the recommended block size (96, 96) without overlaps.

Parameters

- **key** (*str*) – Key of image. Default: 'pred_img'
- **is_predicted** (*bool*) – If the image is predicted, it will be picked from predictions; otherwise, it will be picked from data_batch. Default: True
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Default: None
- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the PSNR calculation. Default: 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Default: 'HWC'.
- **convert_to** (*str*) – Whether to convert the images to other color models. If None, the images are not altered. When computing for 'Y', the images are assumed to be in BGR order. Options are 'Y' and None. Default: 'gray'.

Metrics:

- NIQE (float): Natural Image Quality Evaluator

metric = NIQE

process_image(*gt*, *pred*, *mask*) → None

Process an image.

Parameters

- **gt** (*np.ndarray*) – GT image.
- **pred** (*np.ndarray*) – Pred image.
- **mask** (*np.ndarray*) – Mask of evaluation.

Returns NIQE result.

Return type result (np.ndarray)

class mmedit.evaluation.**PSNR**(*gt_key: str = 'gt_img'*, *pred_key: str = 'pred_img'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *crop_border=0*, *input_order='CHW'*, *convert_to=None*)

Bases: mmedit.evaluation.metrics.base_sample_wise_metric.BaseSampleWiseMetric

Peak Signal-to-Noise Ratio.

Ref: https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

Parameters

- **gt_key** (*str*) – Key of ground-truth. Default: 'gt_img'
- **pred_key** (*str*) – Key of prediction. Default: 'pred_img'
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.

- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Default: None
- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the PSNR calculation. Default: 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Default: 'CHW'.
- **convert_to** (*str*) – Whether to convert the images to other color models. If None, the images are not altered. When computing for 'Y', the images are assumed to be in BGR order. Options are 'Y' and None. Default: None.

Metrics:

- PSNR (float): Peak Signal-to-Noise Ratio

metric = PSNR

process_image(*gt*, *pred*, *mask*)

Process an image.

Parameters

- **gt** (*Torch | np.ndarray*) – GT image.
- **pred** (*Torch | np.ndarray*) – Pred image.
- **mask** (*Torch | np.ndarray*) – Mask of evaluation.

Returns PSNR result.

Return type np.ndarray

class mmedit.evaluation.**SAD**(*norm_const=1000*, ***kwargs*)

Bases: mmengine.evaluator.BaseMetric

Sum of Absolute Differences metric for image matting.

This metric compute per-pixel absolute difference and sum across all pixels. i.e. sum(abs(a-b)) / norm_const

---

Note: Current implementation assume image / alpha / trimap array in numpy format and with pixel value ranging from 0 to 255.

Note: pred_alpha should be masked by trimap before passing into this metric

---

Default prefix: ''

Parameters **norm_const** (*int*) – Divide the result to reduce its magnitude. Default to 1000.

Metrics:

- SAD (float): Sum of Absolute Differences

default_prefix =

process(*data_batch: Sequence[dict]*, *data_samples: Sequence[dict]*) → None

Process one batch of data and predictions.

Parameters

- **data_batch** (*Sequence[Tuple[Any, dict]]*) – A batch of data from the dataloader.
- **predictions** (*Sequence[dict]*) – A batch of outputs from the model.

compute_metrics(*results: List*)

Compute the metrics from processed results.

Parameters **results** (*dict*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict

class mmedit.evaluation.**SNR**(*gt_key: str = 'gt_img'*, *pred_key: str = 'pred_img'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *crop_border=0*, *input_order='CHW'*, *convert_to=None*)

Bases: mmedit.evaluation.metrics.base_sample_wise_metric.BaseSampleWiseMetric

Signal-to-Noise Ratio.

Ref: https://en.wikipedia.org/wiki/Signal-to-noise_ratio

Parameters

- **gt_key** (*str*) – Key of ground-truth. Default: 'gt_img'
- **pred_key** (*str*) – Key of prediction. Default: 'pred_img'
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Default: None
- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the SNR calculation. Default: 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Default: 'CHW'.
- **convert_to** (*str*) – Whether to convert the images to other color models. If None, the images are not altered. When computing for 'Y', the images are assumed to be in BGR order. Options are 'Y' and None. Default: None.

Metrics:

- SNR (float): Signal-to-Noise Ratio

metric = SNR

process_image(*gt*, *pred*, *mask*)

Process an image.

Parameters

- **gt** (*Torch | np.ndarray*) – GT image.
- **pred** (*Torch | np.ndarray*) – Pred image.

- **mask** (*Torch | np.ndarray*) – Mask of evaluation.

Returns SNR result.

Return type np.ndarray

class mmedit.evaluation.**SSIM**(*gt_key: str = 'gt_img'*, *pred_key: str = 'pred_img'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *crop_border=0*, *input_order='CHW'*, *convert_to=None*)

Bases: mmedit.evaluation.metrics.base_sample_wise_metric.BaseSampleWiseMetric

Calculate SSIM (structural similarity).

Ref: Image quality assessment: From error visibility to structural similarity

The results are the same as that of the official released MATLAB code in https://ece.uwaterloo.ca/~z70wang/research/ssim/.

For three-channel images, SSIM is calculated for each channel and then averaged.

Parameters

- **gt_key** (*str*) – Key of ground-truth. Default: 'gt_img'
- **pred_key** (*str*) – Key of prediction. Default: 'pred_img'
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Default: None
- **crop_border** (*int*) – Cropped pixels in each edges of an image. These pixels are not involved in the PSNR calculation. Default: 0.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. Default: 'HWC'.
- **convert_to** (*str*) – Whether to convert the images to other color models. If None, the images are not altered. When computing for 'Y', the images are assumed to be in BGR order. Options are 'Y' and None. Default: None.

Metrics:

- SSIM (float): Structural similarity

metric = SSIM

process_image(*gt*, *pred*, *mask*)

Process an image.

Parameters

- **gt** (*Torch | np.ndarray*) – GT image.
- **pred** (*Torch | np.ndarray*) – Pred image.
- **mask** (*Torch | np.ndarray*) – Mask of evaluation.

Returns SSIM result.

Return type np.ndarray

class mmedit.evaluation.**ConnectivityError**(*step=0.1*, *norm_constant=1000*, ***kwargs*)

Bases: mmengine.evaluator.BaseMetric

Connectivity error for evaluating alpha matte prediction.

Note: Current implementation assume image / alpha / trimap array in numpy format and with pixel value ranging from 0 to 255.

Note: pred_alpha should be masked by trimap before passing into this metric

---

Parameters

- **step** (*float*) – Step of threshold when computing intersection between *alpha* and *pred_alpha*. Default to 0.1 .
- **norm_const** (*int*) – Divide the result to reduce its magnitude. Default to 1000.

Default prefix: ''

Metrics:

- ConnectivityError (float): Connectivity Error

process(*data_batch: Sequence[dict]*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*Sequence[dict]*) – A batch of data from the dataloader.
- **predictions** (*Sequence[dict]*) – A batch of outputs from the model.

compute_metrics(*results: List*)

Compute the metrics from processed results.

Parameters **results** (*dict*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict

class mmedit.evaluation.**Equivariance**(*fake_nums: int*, *real_nums: int = 0*, *fake_key: Optional[str] = None*, *real_key: Optional[str] = 'img'*, *need_cond_input: bool = False*, *sample_mode: str = 'ema'*, *sample_kwargs: dict = dict()*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *eq_cfg=dict()*)

Bases: mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric

Metric for generative metrics. Except for the preparation phase (`prepare()`), generative metrics do not need extra real images.

Parameters

- **fake_nums** (*int*) – Numbers of the generated image need for the metric.
- **real_nums** (*int*) – Numbers of the real image need for the metric. If *-1* is passed means all images from the dataset is need. Defaults to 0.

- **fake_key** (*Optional[str]*) – Key for get fake images of the output dict. Defaults to None.
- **real_key** (*Optional[str]*) – Key for get real images from the input dict. Defaults to 'img'.
- **need_cond_input** (*bool*) – If true, the sampler will return the conditional input randomly sampled from the original dataset. This require the dataset implement *get_data_info* and field *gt_label* must be contained in the return value of *get_data_info*. Noted that, for unconditional models, set *need_cond_input* as True may influence the result of evaluation results since the conditional inputs are sampled from the dataset distribution; otherwise will be sampled from the uniform distribution. Defaults to False.
- **sample_model** (*str*) – Sampling mode for the generative model. Support 'orig' and 'ema'. Defaults to 'ema'.
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Defaults to None.
- **sample_kwargs** (*dict*) – Sampling arguments for model test.

name = Equivariance

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

get_metric_sampler(*model: torch.nn.Module*, *dataloader: torch.utils.data.dataloader.DataLoader*, *metrics: List[mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric]*)

Get sampler for generative metrics. Returns a dummy iterator, whose return value of each iteration is a dict containing batch size and sample mode to generate images.

Parameters

- **model** (*nn.Module*) – Model to evaluate.
- **dataloader** (*DataLoader*) – Dataloader for real images. Used to get batch size during generate fake images.
- **metrics** (*List['GenerativeMetric']*) – Metrics with the same sampler mode.

Returns Sampler for generative metrics.

Return type dummy_iterator

compute_metrics(*results*) → dict

Compute the metrics from processed results.

Parameters **results** (*list*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type dict

_collect_target_results(*target: str*) → Optional[list]

Collect function for Eq metric. This function support collect results typing as Dict[List[Tensor]]`.

Parameters **target** (*str*) – Target results to collect.

Returns The collected results.

Return type Optional[list]

class mmedit.evaluation.**FrechetInceptionDistance**(*fake_nums: int*, *real_nums: int = - 1*, *inception_style='StyleGAN'*, *inception_path: Optional[str] = None*, *inception_pkl: Optional[str] = None*, *fake_key: Optional[str] = None*, *real_key: Optional[str] = 'img'*, *need_cond_input: bool = False*, *sample_model: str = 'orig'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *sample_kwargs: dict = dict()*)

Bases: mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric

FID metric. In this metric, we calculate the distance between real distributions and fake distributions. The distributions are modeled by the real samples and fake samples, respectively. *Inception_v3* is adopted as the feature extractor, which is widely used in StyleGAN and BigGAN.

Parameters

- **fake_nums** (*int*) – Numbers of the generated image need for the metric.
- **real_nums** (*int*) – Numbers of the real images need for the metric. If -1 is passed, means all real images in the dataset will be used. Defaults to -1.
- **inception_style** (*str*) – The target inception style want to load. If the given style cannot be loaded successful, will attempt to load a valid one. Defaults to 'StyleGAN'.
- **inception_path** (*str, optional*) – Path the the pretrain Inception network. Defaults to None.
- **inception_pkl** (*str, optional*) – Path to reference inception pickle file. If *None*, the statistical value of real distribution will be calculated at running time. Defaults to None.
- **fake_key** (*Optional[str]*) – Key for get fake images of the output dict. Defaults to None.
- **real_key** (*Optional[str]*) – Key for get real images from the input dict. Defaults to 'img'.
- **need_cond_input** (*bool*) – If true, the sampler will return the conditional input randomly sampled from the original dataset. This require the dataset implement *get_data_info* and field *gt_label* must be contained in the return value of *get_data_info*. Noted that, for unconditional models, set *need_cond_input* as True may influence the result of evaluation results since the conditional inputs are sampled from the dataset distribution; otherwise will be sampled from the uniform distribution. Defaults to False.
- **sample_model** (*str*) – Sampling mode for the generative model. Support 'orig' and 'ema'. Defaults to 'orig'.
- **collect_device** (*str, optional*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Defaults to None.

name = FID

prepare(*module: torch.nn.Module*, *dataloader: torch.utils.data.dataloader.DataLoader*) → None

Preparing inception feature for the real images.

Parameters

- **module** (*nn.Module*) – The model to evaluate.
- **dataloader** (*DataLoader*) – The dataloader for real images.

_load_inception(*inception_style: str*, *inception_path: Optional[str]*) → Tuple[torch.nn.Module, str]

Load inception and return the successful loaded style.

Parameters

- **inception_style** (*str*) – Target style of Inception network want to load.
- **inception_path** (*Optional[str]*) – The path to the inception.

Returns

The actually loaded inception network and corresponding style.

Return type Tuple[nn.Module, str]

forward_inception(*image: torch.Tensor*) → torch.Tensor

Feed image to inception network and get the output feature.

Parameters **image** (*Tensor*) – Image tensor fed to the Inception network.

Returns Image feature extracted from inception.

Return type Tensor

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

static _calc_fid(*sample_mean: numpy.ndarray*, *sample_cov: numpy.ndarray*, *real_mean: numpy.ndarray*, *real_cov: numpy.ndarray*, *eps: float = 1e-06*) → Tuple[float]

Refer to the implementation from:

https://github.com/rosinality/stylegan2-pytorch/blob/master/fid.py#L34

compute_metrics(*fake_results: list*) → dict

Compulate the result of FID metric.

Parameters **fake_results** (*list*) – List of image feature of fake images.

Returns

A dict of the computed FID metric and its mean and covariance.

Return type dict

class mmedit.evaluation.**GradientError**(*sigma=1.4*, *norm_constant=1000*, ***kwargs*)

Bases: mmengine.evaluator.BaseMetric

Gradient error for evaluating alpha matte prediction.

Note: Current implementation assume image / alpha / trimap array in numpy format and with pixel value ranging from 0 to 255.

Note: pred_alpha should be masked by trimap before passing into this metric

---

Parameters

- **sigma** (*float*) – Standard deviation of the gaussian kernel. Defaults to 1.4 .
- **norm_const** (*int*) – Divide the result to reduce its magnitude. Defaults to 1000 .

Default prefix: ''

Metrics:

- GradientError (float): Gradient Error

process(*data_batch: Sequence[dict]*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*Sequence[dict]*) – A batch of data from the dataloader.
- **predictions** (*Sequence[dict]*) – A batch of outputs from the model.

compute_metrics(*results: List*)

Compute the metrics from processed results.

Parameters **results** (*dict*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict

class mmedit.evaluation.**InceptionScore**(*fake_nums: int = 50000.0*, *resize: bool = True*, *splits: int = 10*, *inception_style: str = 'StyleGAN'*, *inception_path: Optional[str] = None*, *resize_method='bicubic'*, *use_pillow_resize: bool = True*, *fake_key: Optional[str] = None*, *need_cond_input: bool = False*, *sample_model='orig'*, *collect_device: str = 'cpu'*, *prefix: str = None*)

Bases: mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric

IS (Inception Score) metric. The images are split into groups, and the inception score is calculated on each group of images, then the mean and standard deviation of the score is reported. The calculation of the inception score on a group of images involves first using the inception v3 model to calculate the conditional probability for each image (p(y|x)). The marginal probability is then calculated as the average of the conditional probabilities for the images in the group (p(y)). The KL divergence is then calculated for each image as the conditional probability multiplied by the log of the conditional probability minus the log of the marginal probability. The KL divergence

is then summed over all images and averaged over all classes and the exponent of the result is calculated to give the final score.

Ref: https://github.com/sbarratt/inception-score-pytorch/blob/master/inception_score.py # noqa

Note that we highly recommend that users should download the Inception V3 script module from the following address. Then, the *inception_pkl* can be set with user's local path. If not given, we will use the Inception V3 from pytorch model zoo. However, this may bring significant different in the final results.

Tero's Inception V3: https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-pytorch/pretrained/metrics/inception-2015-12-05.pt # noqa

Parameters

- **fake_nums** (*int*) – Numbers of the generated image need for the metric.
- **resize** (*bool, optional*) – Whether resize image to 299x299. Defaults to True.
- **splits** (*int, optional*) – The number of groups. Defaults to 10.
- **inception_style** (*str*) – The target inception style want to load. If the given style cannot be loaded successful, will attempt to load a valid one. Defaults to 'StyleGAN'.
- **inception_path** (*str, optional*) – Path the the pretrain Inception network. Defaults to None.
- **resize_method** (*str*) – Resize method. If *resize* is False, this will be ignored. Defaults to 'bicubic'.
- **use_pil_resize** (*bool*) – Whether use Bicubic interpolation with Pillow's backend. If set as True, the evaluation process may be a little bit slow, but achieve a more accurate IS result. Defaults to False.
- **fake_key** (*Optional[str]*) – Key for get fake images of the output dict. Defaults to None.
- **real_key** (*Optional[str]*) – Key for get real images from the input dict. Defaults to 'img'.
- **need_cond_input** (*bool*) – If true, the sampler will return the conditional input randomly sampled from the original dataset. This require the dataset implement *get_data_info* and field *gt_label* must be contained in the return value of *get_data_info*. Noted that, for unconditional models, set *need_cond_input* as True may influence the result of evaluation results since the conditional inputs are sampled from the dataset distribution; otherwise will be sampled from the uniform distribution. Defaults to False.
- **sample_model** (*str*) – Sampling mode for the generative model. Support 'orig' and 'ema'. Defaults to 'orig'.
- **collect_device** (*str, optional*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Defaults to None.

name = IS

pil_resize_method_mapping

prepare(*module: torch.nn.Module*, *dataloader: torch.utils.data.dataloader.DataLoader*) → None

Prepare for the pre-calculating items of the metric. Defaults to do nothing.

Parameters

- **module** (*nn.Module*) – Model to evaluate.
- **dataloader** (*DataLoader*) – Dataloader for the real images.

_load_inception(*inception_style: str*, *inception_path: Optional[str]*) → Tuple[torch.nn.Module, str]

Load pretrain model of inception network. :param inception_style: Target style of Inception network want to

load.

Parameters **inception_path** (*Optional[str]*) – The path to the inception.

Returns

The actually loaded inception network and corresponding style.

Return type Tuple[nn.Module, str]

_preprocess(*image: torch.Tensor*) → torch.Tensor

Preprocess image before pass to the Inception. Preprocess operations contain channel conversion and resize.

Parameters **image** (*Tensor*) – Image tensor before preprocess.

Returns

Image tensor after resize and channel conversion (if need.)

Return type Tensor

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

compute_metrics(*fake_results: list*) → dict

Compute the results of Inception Score metric.

Parameters **fake_results** (*list*) – List of image feature of fake images.

Returns A dict of the computed IS metric and its standard error

Return type dict

class mmedit.evaluation.**MattingMSE**(*norm_const=1000*, ***kwargs*)

Bases: mmengine.evaluator.BaseMetric

Mean Squared Error metric for image matting.

This metric compute per-pixel squared error average across all pixels. i.e. mean((a-b)^2) / norm_const

---

Note: Current implementation assume image / alpha / trimap array in numpy format and with pixel value ranging from 0 to 255.

Note: pred_alpha should be masked by trimap before passing into this metric

---

Default prefix: ‘’

Parameters **norm_const** (*int*) – Divide the result to reduce its magnitude. Default to 1000.

Metrics:

- MattingMSE (float): Mean of Squared Error

default_prefix =

process(*data_batch: Sequence[dict]*, *data_samples: Sequence[dict]*) → None

Process one batch of data and predictions.

Parameters

- **data_batch** (*Sequence[dict]*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

compute_metrics(*results: List*)

Compute the metrics from processed results.

Parameters **results** (*dict*) – The processed results of each batch.

Returns The computed metrics. The keys are the names of the metrics, and the values are corresponding results.

Return type Dict

class mmedit.evaluation.**MultiScaleStructureSimilarity**(*fake_nums: int*, *fake_key: Optional[str] = None*, *need_cond_input: bool = False*, *sample_model: str = 'ema'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*)

Bases: mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric

MS-SSIM (Multi-Scale Structure Similarity) metric.

Ref: https://github.com/tkarras/progressive_growing_of_gans/blob/master/metrics/ms_ssim.py # noqa

Parameters

- **fake_nums** (*int*) – Numbers of the generated image need for the metric.
- **fake_key** (*Optional[str]*) – Key for get fake images of the output dict. Defaults to None.
- **real_key** (*Optional[str]*) – Key for get real images from the input dict. Defaults to 'img'.
- **need_cond_input** (*bool*) – If true, the sampler will return the conditional input randomly sampled from the original dataset. This require the dataset implement *get_data_info* and field *gt_label* must be contained in the return value of *get_data_info*. Noted that, for unconditional models, set *need_cond_input* as True may influence the result of evaluation results since the conditional inputs are sampled from the dataset distribution; otherwise will be sampled from the uniform distribution. Defaults to False.
- **sample_model** (*str*) – Sampling mode for the generative model. Support 'orig' and 'ema'. Defaults to 'ema'.
- **collect_device** (*str, optional*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Defaults to None.

name = MS-SSIM

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Feed data to the metric.

Parameters

- **data_batch** (*dict*) – Real images from dataloader. Do not be used in this metric.
- **data_samples** (*Sequence[dict]*) – Generated images.

_collect_target_results(*target: str*) → Optional[list]

Collected results for MS-SSIM metric. Size of *self.fake_results* in MS-SSIM does not relay on *self.fake_nums* but *self.num_pairs*.

Parameters **target** (*str*) – Target results to collect.

Returns The collected results.

Return type Optional[list]

compute_metrics(*results_fake: List*)

Computed the result of MS-SSIM.

Returns Calculated MS-SSIM result.

Return type dict

class mmedit.evaluation.**PerceptualPathLength**(*fake_nums: int*, *real_nums: int = 0*, *fake_key: Optional[str] = None*, *real_key: Optional[str] = 'img'*, *need_cond_input: bool = False*, *sample_model: str = 'ema'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *crop=True*, *epsilon=0.0001*, *space='W'*, *sampling='end'*, *latent_dim=512*)

Bases: mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric

Perceptual path length.

Measure the difference between consecutive images (their VGG16 embeddings) when interpolating between two random inputs. Drastic changes mean that multiple features have changed together and that they might be entangled.

Ref: https://github.com/rosinality/stylegan2-pytorch/blob/master/ppl.py # noqa

Parameters

- **num_images** (*int*) – The number of evaluated generated samples.
- **image_shape** (*tuple, optional*) – Image shape in order "CHW". Defaults to None.
- **crop** (*bool, optional*) – Whether crop images. Defaults to True.
- **epsilon** (*float, optional*) – Epsilon parameter for path sampling. Defaults to 1e-4.
- **space** (*str, optional*) – Latent space. Defaults to 'W'.
- **sampling** (*str, optional*) – Sampling mode, whether sampling in full path or endpoints. Defaults to 'end'.
- **latent_dim** (*int, optional*) – Latent dimension of input noise. Defaults to 512.
- **need_cond_input** (*bool*) – If true, the sampler will return the conditional input randomly sampled from the original dataset. This require the dataset implement *get_data_info* and field *gt_label* must be contained in the return value of *get_data_info*. Noted that, for unconditional models, set *need_cond_input* as True may influence the result of evaluation results since the

conditional inputs are sampled from the dataset distribution; otherwise will be sampled from the uniform distribution. Defaults to False.

SAMPLER_MODE = path

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

_compute_distance(*images*)

Feed data to the metric.

Parameters **images** (*Tensor*) – Input tensor.

compute_metrics(*fake_results: list*) → dict

Summarize the results.

Returns Summarized results.

Return type dict | list

get_metric_sampler(*model: torch.nn.Module*, *dataloader: torch.utils.data.dataloader.DataLoader*, *metrics: list*)

Get sampler for generative metrics. Returns a dummy iterator, whose return value of each iteration is a dict containing batch size and sample mode to generate images.

Parameters

- **model** (*nn.Module*) – Model to evaluate.
- **dataloader** (*DataLoader*) – Dataloader for real images. Used to get batch size during generate fake images.
- **metrics** (*list*) – Metrics with the same sampler mode.

Returns Sampler for generative metrics.

Return type dummy_iterator

class mmedit.evaluation.**PrecisionAndRecall**(*fake_nums*, *real_nums=- 1*, *k=3*, *fake_key: Optional[str] = None*, *real_key: Optional[str] = 'img'*, *need_cond_input: bool = False*, *sample_model: str = 'ema'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*, *vgg16_script='work_dirs/cache/vgg16.pt'*, *vgg16_pkl=None*, *row_batch_size=10000*, *col_batch_size=10000*, *auto_save=True*)

Bases: mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric

Improved Precision and recall metric.

In this metric, we draw real and generated samples respectively, and embed them into a high-dimensional feature space using a pre-trained classifier network. We use these features to estimate the corresponding manifold. We obtain the estimation by calculating pairwise Euclidean distances between all feature vectors in the set and, for each feature vector, construct a hypersphere with radius equal to the distance to its kth nearest neighbor. Together, these hyperspheres define a volume in the feature space that serves as an estimate of the true manifold. Precision is quantified by querying for each generated image whether the image is within the estimated manifold of real

images. Symmetrically, recall is calculated by querying for each real image whether the image is within estimated manifold of generated image.

Ref: https://github.com/NVlabs/stylegan2-ada-pytorch/blob/main/metrics/precision_recall.py # noqa

Note that we highly recommend that users should download the vgg16 script module from the following address. Then, the *vgg16_script* can be set with user's local path. If not given, we will use the vgg16 from pytorch model zoo. However, this may bring significant different in the final results.

Tero's vgg16: https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-pytorch/pretrained/metrics/vgg16.pt

Parameters

- **num_images** (*int*) – The number of evaluated generated samples.
- **image_shape** (*tuple*) – Image shape in order "CHW". Defaults to None.
- **num_real_need** (*int | None, optional*) – The number of real images. Defaults to None.
- **full_dataset** (*bool, optional*) – Whether to use full dataset for evaluation. Defaults to False.
- **k** (*int, optional*) – Kth nearest parameter. Defaults to 3.
- **bgr2rgb** (*bool, optional*) – Whether to change the order of image channel. Defaults to True.
- **vgg16_script** (*str, optional*) – Path for the Tero's vgg16 module. Defaults to 'work_dirs/cache/vgg16.pt'.
- **row_batch_size** (*int, optional*) – The batch size of row data. Defaults to 10000.
- **col_batch_size** (*int, optional*) – The batch size of col data. Defaults to 10000.
- **auto_save** (*bool, optional*) – Whether save vgg feature automatically.
- **need_cond_input** (*bool*) – If true, the sampler will return the conditional input randomly sampled from the original dataset. This require the dataset implement *get_data_info* and field *gt_label* must be contained in the return value of *get_data_info*. Noted that, for unconditional models, set *need_cond_input* as True may influence the result of evaluation results since the conditional inputs are sampled from the dataset distribution; otherwise will be sampled from the uniform distribution. Defaults to False.

name = PR

_load_vgg(*vgg16_script: Optional[str]*) → Tuple[torch.nn.Module, bool]

Load VGG network from the given path.

Parameters **vgg16_script** – The path of script model of VGG network. If None, will load the pytorch version.

Returns

The actually loaded VGG network and corresponding style.

Return type Tuple[nn.Module, str]

extract_features(*images*)

Extracting image features.

Parameters **images** (*torch.Tensor*) – Images tensor.

Returns Vgg16 features of input images.

Return type torch.Tensor

compute_metrics(*results_fake*) → dict

compute_metrics.

Returns Summarized results.

Return type dict

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

prepare(*module: torch.nn.Module*, *dataloader: torch.utils.data.dataloader.DataLoader*) → None

Prepare for the pre-calculating items of the metric. Defaults to do nothing.

Parameters

- **module** (*nn.Module*) – Model to evaluate.
- **dataloader** (*DataLoader*) – Dataloader for the real images.

class mmedit.evaluation.**SlicedWassersteinDistance**(*fake_nums: int*, *image_shape: tuple*, *fake_key: Optional[str] = None*, *real_key: Optional[str] = 'img'*, *sample_model: str = 'ema'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*)

Bases: mmedit.evaluation.metrics.base_gen_metric.GenMetric

SWD (Sliced Wasserstein distance) metric. We calculate the SWD of two sets of images in the following way. In every 'feed', we obtain the Laplacian pyramids of every images and extract patches from the Laplacian pyramids as descriptors. In 'summary', we normalize these descriptors along channel, and reshape them so that we can use these descriptors to represent the distribution of real/fake images. And we can calculate the sliced Wasserstein distance of the real and fake descriptors as the SWD of the real and fake images.

Ref: https://github.com/tkarras/progressive_growing_of_gans/blob/master/metrics/sliced_wasserstein.py # noqa

Parameters

- **fake_nums** (*int*) – Numbers of the generated image need for the metric.
- **image_shape** (*tuple*) – Image shape in order "CHW".
- **fake_key** (*Optional[str]*) – Key for get fake images of the output dict. Defaults to None.
- **real_key** (*Optional[str]*) – Key for get real images from the input dict. Defaults to 'img'.
- **sample_model** (*str*) – Sampling mode for the generative model. Support 'orig' and 'ema'. Defaults to 'ema'.
- **collect_device** (*str*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Defaults to None.

name = SWD

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results` and `self.real_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

_collect_target_results(*target: str*) → Optional[list]

Collect function for SWD metric. This function support collect results typing as *List[List[Tensor]]*.

Parameters **target** (*str*) – Target results to collect.

Returns The collected results.

Return type Optional[list]

compute_metrics(*results_fake*, *results_real*) → dict

Compulate the result of SWD metric.

Parameters

- **fake_results** (*list*) – List of image feature of fake images.
- **real_results** (*list*) – List of image feature of real images.

Returns A dict of the computed SWD metric.

Return type dict

class mmedit.evaluation.**TransFID**(*fake_nums: int*, *real_nums: int = - 1*, *inception_style='StyleGAN'*, *inception_path: Optional[str] = None*, *inception_pkl: Optional[str] = None*, *fake_key: Optional[str] = None*, *real_key: Optional[str] = 'img'*, *sample_model: str = 'ema'*, *collect_device: str = 'cpu'*, *prefix: Optional[str] = None*)

Bases: *FrechetInceptionDistance*

FID metric. In this metric, we calculate the distance between real distributions and fake distributions. The distributions are modeled by the real samples and fake samples, respectively. *Inception_v3* is adopted as the feature extractor, which is widely used in StyleGAN and BigGAN.

Parameters

- **fake_nums** (*int*) – Numbers of the generated image need for the metric.
- **real_nums** (*int*) – Numbers of the real images need for the metric. If -1 is passed, means all real images in the dataset will be used. Defaults to -1.
- **inception_style** (*str*) – The target inception style want to load. If the given style cannot be loaded successful, will attempt to load a valid one. Defaults to 'StyleGAN'.
- **inception_path** (*str, optional*) – Path the the pretrain Inception network. Defaults to None.
- **inception_pkl** (*str, optional*) – Path to reference inception pickle file. If *None*, the statistical value of real distribution will be calculated at running time. Defaults to None.
- **fake_key** (*Optional[str]*) – Key for get fake images of the output dict. Defaults to None.
- **real_key** (*Optional[str]*) – Key for get real images from the input dict. Defaults to 'img'.

- **need_cond_input** (*bool*) – If true, the sampler will return the conditional input randomly sampled from the original dataset. This require the dataset implement *get_data_info* and field *gt_label* must be contained in the return value of *get_data_info*. Noted that, for unconditional models, set *need_cond_input* as True may influence the result of evaluation results since the conditional inputs are sampled from the dataset distribution; otherwise will be sampled from the uniform distribution. Defaults to False.
- **sample_model** (*str*) – Sampling mode for the generative model. Support 'orig' and 'ema'. Defaults to 'orig'.
- **collect_device** (*str, optional*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Defaults to None.

get_metric_sampler(*model: torch.nn.Module*, *dataloader: torch.utils.data.dataloader.DataLoader*, *metrics: List[mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric]*) → torch.utils.data.dataloader.DataLoader

Get sampler for normal metrics. Directly returns the dataloader.

Parameters

- **model** (*nn.Module*) – Model to evaluate.
- **dataloader** (*DataLoader*) – Dataloader for real images.
- **metrics** (*List['GenMetric']*) – Metrics with the same sample mode.

Returns Default sampler for normal metrics.

Return type DataLoader

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **data_samples** (*Sequence[dict]*) – A batch of outputs from the model.

class mmedit.evaluation.**TransIS**(*fake_nums: int = 50000*, *resize: bool = True*, *splits: int = 10*, *inception_style: str = 'StyleGAN'*, *inception_path: Optional[str] = None*, *resize_method='bicubic'*, *use_pillow_resize: bool = True*, *fake_key: Optional[str] = None*, *sample_model='ema'*, *collect_device: str = 'cpu'*, *prefix: str = None*)

Bases: *InceptionScore*

IS (Inception Score) metric. The images are split into groups, and the inception score is calculated on each group of images, then the mean and standard deviation of the score is reported. The calculation of the inception score on a group of images involves first using the inception v3 model to calculate the conditional probability for each image (p(y|x)). The marginal probability is then calculated as the average of the conditional probabilities for the images in the group (p(y)). The KL divergence is then calculated for each image as the conditional probability multiplied by the log of the conditional probability minus the log of the marginal probability. The KL divergence is then summed over all images and averaged over all classes and the exponent of the result is calculated to give the final score.

Ref: https://github.com/sbarratt/inception-score-pytorch/blob/master/inception_score.py # noqa

Note that we highly recommend that users should download the Inception V3 script module from the following address. Then, the *inception_pkl* can be set with user's local path. If not given, we will use the Inception V3 from pytorch model zoo. However, this may bring significant different in the final results.

Tero's Inception V3: https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-pytorch/pretrained/metrics/inception-2015-12-05.pt # noqa

Parameters

- **fake_nums** (*int*) – Numbers of the generated image need for the metric.
- **resize** (*bool, optional*) – Whether resize image to 299x299. Defaults to True.
- **splits** (*int, optional*) – The number of groups. Defaults to 10.
- **inception_style** (*str*) – The target inception style want to load. If the given style cannot be loaded successful, will attempt to load a valid one. Defaults to 'StyleGAN'.
- **inception_path** (*str, optional*) – Path the the pretrain Inception network. Defaults to None.
- **resize_method** (*str*) – Resize method. If *resize* is False, this will be ignored. Defaults to 'bicubic'.
- **use_pil_resize** (*bool*) – Whether use Bicubic interpolation with Pillow's backend. If set as True, the evaluation process may be a little bit slow, but achieve a more accurate IS result. Defaults to False.
- **fake_key** (*Optional[str]*) – Key for get fake images of the output dict. Defaults to None.
- **real_key** (*Optional[str]*) – Key for get real images from the input dict. Defaults to 'img'.
- **sample_model** (*str*) – Sampling mode for the generative model. Support 'orig' and 'ema'. Defaults to 'ema'.
- **collect_device** (*str, optional*) – Device name used for collecting results from different ranks during distributed training. Must be 'cpu' or 'gpu'. Defaults to 'cpu'.
- **prefix** (*str, optional*) – The prefix that will be added in the metric names to disambiguate homonymous metrics of different evaluators. If prefix is not provided in the argument, self.default_prefix will be used instead. Defaults to None.

process(*data_batch: dict*, *data_samples: Sequence[dict]*) → None

Process one batch of data samples and predictions. The processed results should be stored in `self.fake_results`, which will be used to compute the metrics when all batches have been processed.

Parameters

- **data_batch** (*dict*) – A batch of data from the dataloader.
- **predictions** (*Sequence[dict]*) – A batch of outputs from the model.

get_metric_sampler(*model: torch.nn.Module*, *dataloader: torch.utils.data.dataloader.DataLoader*, *metrics: List[mmedit.evaluation.metrics.base_gen_metric.GenerativeMetric]*) → torch.utils.data.dataloader.DataLoader

Get sampler for normal metrics. Directly returns the dataloader.

Parameters

- **model** (*nn.Module*) – Model to evaluate.
- **dataloader** (*DataLoader*) – Dataloader for real images.
- **metrics** (*List['GenMetric']*) – Metrics with the same sample mode.

Returns Default sampler for normal metrics.

Return type DataLoader

1.43 mmedit.visualization

1.43.1 Package Contents

Classes

ConcatImageVisualizer	Visualize multiple images by concatenation.
GenVisualizer	MMEditing Visualizer.
GenVisBackend	Generation visualization backend class. It can write image, config,
PaviGenVisBackend	Visualization backend for Pavi.
TensorboardGenVisBackend	Tensorboard visualization backend class.
WandbGenVisBackend	Wandb visualization backend for MMEditing.

class mmedit.visualization.**ConcatImageVisualizer**(*fn_key: str*, *img_keys: Sequence[str]*, *pixel_range={}*, *bgr2rgb=False*, *name: str = 'visualizer'*, **args*, ***kwargs*)

Bases: mmengine.visualization.Visualizer

Visualize multiple images by concatenation.

This visualizer will horizontally concatenate images belongs to different keys and vertically concatenate images belongs to different frames to visualize.

Image to be visualized can be:

- torch.Tensor or np.array
- Image sequences of shape (T, C, H, W)
- Multi-channel image of shape (1/3, H, W)
- Single-channel image of shape (C, H, W)

Parameters

- **fn_key** (*str*) – key used to determine file name for saving image. Usually it is the path of some input image. If the value is *dir/basename.ext*, the name used for saving will be basename.
- **img_keys** (*str*) – keys, values of which are images to visualize.
- **pixel_range** (*dict*) – min and max pixel value used to denormalize images, note that only float array or tensor will be denormalized, uint8 arrays are assumed to be unnormalized.
- **bgr2rgb** (*bool*) – whether to convert the image from BGR to RGB.
- **name** (*str*) – name of visualizer. Default: 'visualizer'.
- ****kwargs** (**args and*) – Other arguments are passed to *Visualizer*. # noqa

add_datasample(*data_sample:* mmedit.structures.EditDataSample, *step=0*) → None

Concatenate image and draw.

Parameters

- **input** (*torch.Tensor*) – Single input tensor from data_batch.
- **data_sample** (*EditDataSample*) – Single data_sample from data_batch.
- **output** (*EditDataSample*) – Single prediction output by model.
- **step** (*int*) – Global step value to record. Default: 0.

class mmedit.visualization.**GenVisualizer**(*name='visualizer'*, *vis_backends: Optional[List[Dict]] = None*, *save_dir: Optional[str] = None*)

Bases: mmengine.visualization.Visualizer

MMEditing Visualizer.

Parameters

- **name** (*str*) – Name of the instance. Defaults to 'visualizer'.
- **vis_backends** (*list, optional*) – Visual backend config list. Defaults to None.
- **save_dir** (*str, optional*) – Save file dir for all storage backends. If it is None, the backend storage will not save any data.

Examples:

```
>>> # Draw image
>>> vis = GenVisualizer()
>>> vis.add_datasample(
>>>     'random_noise',
>>>     gen_samples=torch.rand(2, 3, 10, 10),
>>>     gt_samples=dict(imgs=torch.randn(2, 3, 10, 10)),
>>>     gt_keys='imgs',
>>>     vis_mode='image',
>>>     n_rows=2,
>>>     step=10)
```

static _post_process_image(*image: torch.Tensor*, *color_order: str*, *mean: mean_std_type = None*, *std: mean_std_type = None*) → torch.Tensor

Post process images. First convert image to *rgb* order. And then de-norm image to *mean* and *std* if they are passed.

Parameters

- **image** (*Tensor*) – Image to pose process.
- **color_order** (*str*) – The color order of the passed image.
- **mean** (*Optional[Sequence[Union[float, int]]], optional*) – Target mean of the passed image. Defaults to None.
- **std** (*Optional[Sequence[Union[float, int]]], optional*) – Target std of the passed image. Defaults to None.

Returns Image in original value range and RGB color order.

Return type Tensor

static _get_n_row_and_padding(*samples: Tuple[dict, torch.Tensor]*, *n_row: Optional[int] = None*) → Tuple[int, Optional[torch.Tensor]]

Get number of sample in each row and tensor for padding the empty position.

Parameters

- **samples** (*Tuple[dict, Tensor]*) – Samples to visualize.
- **n_row** (*int, optional*) – Number of images displayed in each row of. If not passed, n_row will be set as `int(sqrt(batch_size))`.

Returns

Number of sample in each row and tensor for padding the empty position.

Return type Tuple[int, Optional[int]]

_vis_gif_sample(*gen_samples: mmedit.utils.typing.SampleList*, *target_keys: Union[str, List[str], None]*, *color_order: str*, *target_mean: mean_std_type*, *target_std: mean_std_type*, *n_row: int*) → numpy.ndarray

Visualize gif samples.

Parameters

- **gen_samples** (*SampleList*) – List of data samples to visualize
- **target_keys** (*Union[str, List[str], None]*) – Keys of the visualization target in data samples.
- **color_order** (*str*) – The color order of the passed images.
- **target_mean** (*Sequence[Union[float, int]]*) – The target mean of the visualization results.
- **target_std** (*Sequence[Union[float, int]]*) – The target std of the visualization resutts.
- **n_rows** (*int, optional*) – Number of images in one row.

Returns The visualization results.

Return type np.ndarray

_vis_image_sample(*gen_samples: mmedit.utils.typing.SampleList*, *target_keys: Union[str, List[str], None]*, *color_order: str*, *target_mean: mean_std_type*, *target_std: mean_std_type*, *n_row: int*) → numpy.ndarray

Visualize image samples.

Parameters

- **gen_samples** (*SampleList*) – List of data samples to visualize
- **target_keys** (*Union[str, List[str], None]*) – Keys of the visualization target in data samples.
- **color_order** (*str*) – The color order of the passed images.
- **target_mean** (*Sequence[Union[float, int]]*) – The target mean of the visualization results.
- **target_std** (*Sequence[Union[float, int]]*) – The target std of the visualization resutts.
- **n_rows** (*int, optional*) – Number of images in one row.

Returns The visualization results.

Return type np.ndarray

_get_pixel_data_by_key(*sample:* mmedit.structures.EditDataSample, *key: Union[str, List[str]]*) → torch.Tensor

Get tensor in `EditDataSample` by the given key.

Parameters

- **sample** (*EditDataSample*) – Input data sample.
- **key** (*Union[str, List[str]]*) – Name of the target tensor.

Returns Tensor from the data sample.

Return type Tensor

add_datasample(*name: str*, ***, *gen_samples: Sequence[*mmedit.structures.EditDataSample*]*, *target_keys: Optional[Tuple[str, List[str]]] = None*, *vis_mode: Optional[str] = None*, *n_row: Optional[int] = 1*, *color_order: str = 'bgr'*, *target_mean: Sequence[Union[float, int]] = 127.5*, *target_std: Sequence[Union[float, int]] = 127.5*, *show: bool = False*, *wait_time: int = 0*, *step: int = 0*, ***kwargs*) → None

Draw datasample and save to all backends.

If GT and prediction are plotted at the same time, they are displayed in a stitched image where the left image is the ground truth and the right image is the prediction.

If `show` is True, all storage backends are ignored, and the images will be displayed in a local window.

Parameters

- **name** (*str*) – The image identifier.
- **gen_samples** (*List[EditDataSample]*) – Data samples to visualize.
- **vis_mode** (*str, optional*) – Visualization mode. If not passed, will visualize results as image. Defaults to None.
- **n_rows** (*int, optional*) – Number of images in one row. Defaults to 1.
- **color_order** (*str*) – The color order of the passed images. Defaults to 'bgr'.
- **target_mean** (*Sequence[Union[float, int]]*) – The target mean of the visualization results. Defaults to 127.5.
- **target_std** (*Sequence[Union[float, int]]*) – The target std of the visualization resutts. Defaults to 127.5.
- **show** (*bool*) – Whether to display the drawn image. Default to False.
- **wait_time** (*float*) – The interval of show (s). Defaults to 0.
- **step** (*int*) – Global step value to record. Defaults to 0.

add_image(*name: str*, *image: numpy.ndarray*, *step: int = 0*, ***kwargs*) → None

Record the image. Support input kwargs.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray, optional*) – The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) – Global step value to record. Default to 0.

class mmedit.visualization.**GenVisBackend**(*save_dir: str*, *img_save_dir: str = 'vis_image'*, *config_save_file: str = 'config.py'*, *scalar_save_file: str = 'scalars.json'*, *ceph_path: Optional[str] = None*, *delete_local_image: bool = True*)

Bases: mmengine.visualization.BaseVisBackend

Generation visualization backend class. It can write image, config, scalars, etc. to the local hard disk and ceph path. You can get the drawing backend through the experiment property for custom drawing.

Examples

```
>>> from mmgen.visualization import GenVisBackend
>>> import numpy as np
>>> vis_backend = GenVisBackend(save_dir='temp_dir',
>>>                             ceph_path='s3://temp-bucket')
>>> img = np.random.randint(0, 256, size=(10, 10, 3))
>>> vis_backend.add_image('img', img)
>>> vis_backend.add_scalar('mAP', 0.6)
>>> vis_backend.add_scalars({'loss': [1, 2, 3], 'acc': 0.8})
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> vis_backend.add_config(cfg)
```

Parameters

- **save_dir** (*str*) – The root directory to save the files produced by the visualizer.
- **img_save_dir** (*str*) – The directory to save images. Default to 'vis_image'.
- **config_save_file** (*str*) – The file name to save config. Default to 'config.py'.
- **scalar_save_file** (*str*) – The file name to save scalar values. Default to 'scalars.json'.
- **ceph_path** (*Optional[str]*) – The remote path of Ceph cloud storage. Defaults to None.
- **delete_local** (*bool*) – Whether eelete local after uploading to ceph or not. If ceph_path is None, this will be ignored. Defaults to True.

property experiment: *GenVisBackend*

Return the experiment object associated with this visualization backend.

_init_env()

Setup env for VisBackend.

add_config(*config: mmengine.config.Config*, ***kwargs*) → None

Record the config to disk.

Parameters **config** (*Config*) – The Config object

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*) → None

Record the image to disk.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray*) – The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalar(*name: str*, *value: Union[int, float, torch.Tensor, numpy.ndarray]*, *step: int = 0*, ***kwargs*) → None

Record the scalar data to disk.

Parameters

- **name** (*str*) – The scalar identifier.
- **value** (*int, float, torch.Tensor, np.ndarray*) – Value to save.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalars(*scalar_dict: dict*, *step: int = 0*, *file_path: Optional[str] = None*, ***kwargs*) → None

Record the scalars to disk.

The scalar dict will be written to the default and specified files if `file_path` is specified.

Parameters

- **scalar_dict** (*dict*) – Key-value pair storing the tag and corresponding values. The value must be dumped into json format.
- **step** (*int*) – Global step value to record. Default to 0.
- **file_path** (*str, optional*) – The scalar's data will be saved to the `file_path` file at the same time if the `file_path` parameter is specified. Default to None.

_dump(*value_dict: dict*, *file_path: str*, *file_format: str*) → None

dump dict to file.

Parameters

- **value_dict** (*dict*) – The dict data to saved.
- **file_path** (*str*) – The file path to save data.
- **file_format** (*str*) – The file format to save data.

_upload(*path: str*, *delete_local=False*) → None

Upload file at path to remote.

Parameters **path** (*str*) – Path of file to upload.

class mmedit.visualization.**PaviGenVisBackend**(*save_dir: str*, *exp_name: Optional[str] = None*, *labels: Optional[str] = None*, *project: Optional[str] = None*, *model: Optional[str] = None*, *description: Optional[str] = None*)

Bases: mmengine.visualization.BaseVisBackend

Visualization backend for Pavi.

property experiment: *GenVisBackend*

Return the experiment object associated with this visualization backend.

_init_env()

Init save dir.

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*) → None

Record the image to Pavi.

Parameters

- **name** (*str*) – The image identifier.

- **image** (*np.ndarray*) – The image to be saved. The format should be RGB. Default to None.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalar(*name: str*, *value: Union[int, float, torch.Tensor, numpy.ndarray]*, *step: int = 0*, ***kwargs*) → None

Record the scalar data to Pavi.

Parameters

- **name** (*str*) – The scalar identifier.
- **value** (*int, float, torch.Tensor, np.ndarray*) – Value to save.
- **step** (*int*) – Global step value to record. Default to 0.

add_scalars(*scalar_dict: dict*, *step: int = 0*, *file_path: Optional[str] = None*, ***kwargs*) → None

Record the scalars to Pavi.

The scalar dict will be written to the default and specified files if `file_path` is specified.

Parameters

- **scalar_dict** (*dict*) – Key-value pair storing the tag and corresponding values. The value must be dumped into json format.
- **step** (*int*) – Global step value to record. Default to 0.
- **file_path** (*str, optional*) – The scalar's data will be saved to the `file_path` file at the same time if the `file_path` parameter is specified. Default to None.

class mmedit.visualization.**TensorboardGenVisBackend**(*save_dir: str*)

Bases: `mmengine.visualization.TensorboardVisBackend`

Tensorboard visualization backend class.

It can write images, config, scalars, etc. to a tensorboard file.

Examples

```
>>> from mmengine.visualization import TensorboardVisBackend
>>> import numpy as np
>>> vis_backend = TensorboardVisBackend(save_dir='temp_dir')
>>> img = np.random.randint(0, 256, size=(10, 10, 3))
>>> vis_backend.add_image('img', img)
>>> vis_backend.add_scaler('mAP', 0.6)
>>> vis_backend.add_scalars({'loss': 0.1,'acc':0.8})
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> vis_backend.add_config(cfg)
```

Parameters **save_dir** (*str*) – The root directory to save the files produced by the backend.

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*)

Record the image to Tensorboard. Additional support upload gif files.

Parameters

- **name** (*str*) – The image identifier.

- **image** (*np.ndarray*) – The image to be saved. The format should be RGB.
- **step** (*int*) – Useless parameter. Wandb does not need this parameter. Default to 0.

class mmedit.visualization.**WandbGenVisBackend**(*save_dir: str*, *init_kwargs: Optional[dict] = None*, *define_metric_cfg: Optional[dict] = None*, *commit: Optional[bool] = True*, *log_code_name: Optional[str] = None*, *watch_kwargs: Optional[dict] = None*)

Bases: mmengine.visualization.WandbVisBackend

Wandb visualization backend for MMEditing.

_init_env()

Setup env for wandb.

add_image(*name: str*, *image: numpy.array*, *step: int = 0*, ***kwargs*)

Record the image to wandb. Additional support upload gif files.

Parameters

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray*) – The image to be saved. The format should be RGB.
- **step** (*int*) – Useless parameter. Wandb does not need this parameter. Default to 0.

1.44 mmedit.engine.hooks

1.44.1 Package Contents

Classes

ExponentialMovingAverageHook	Exponential Moving Average Hook.
GenIterTimerHook	GenIterTimerHooks inherits from mmengine.hooks.IterTimerHook
PGGANFetchDataHook	PGGAN Fetch Data Hook.
PickleDataHook	Pickle Useful Data Hook.
ReduceLRSchedulerHook	A hook to update learning rate.
BasicVisualizationHook	Basic hook that invoke visualizers during validation and test.
GenVisualizationHook	Generation Visualization Hook. Used to visual output samples in

class mmedit.engine.hooks.**ExponentialMovingAverageHook**(*module_keys*, *interp_mode='lerp'*, *interp_cfg=None*, *interval=- 1*, *start_iter=0*)

Bases: mmengine.hooks.Hook

Exponential Moving Average Hook.

Exponential moving average is a trick that widely used in current GAN literature, e.g., PGGAN, StyleGAN, and BigGAN. This general idea of it is maintaining a model with the same architecture, but its parameters are updated as a moving average of the trained weights in the original model. In general, the model with moving averaged weights achieves better performance.

Parameters

- **module_keys** (*str | tuple[str]*) – The name of the ema model. Note that we require these keys are followed by '_ema' so that we can easily find the original model by discarding the last four characters.
- **interp_mode** (*str, optional*) – Mode of the interpolation method. Defaults to 'lerp'.
- **interp_cfg** (*dict | None, optional*) – Set arguments of the interpolation function. Defaults to None.
- **interval** (*int, optional*) – Evaluation interval (by iterations). Default: -1.
- **start_iter** (*int, optional*) – Start iteration for ema. If the start iteration is not reached, the weights of ema model will maintain the same as the original one. Otherwise, its parameters are updated as a moving average of the trained weights in the original model. Default: 0.

static lerp(*a*, *b*, *momentum=0.001*, *momentum_nontrainable=1.0*, *trainable=True*)

Does a linear interpolation of two parameters/ buffers.

Parameters

- **a** (*torch.Tensor*) – Interpolation start point, refer to orig state.
- **b** (*torch.Tensor*) – Interpolation end point, refer to ema state.
- **momentum** (*float, optional*) – The weight for the interpolation formula. Defaults to 0.001.
- **momentum_nontrainable** (*float, optional*) – The weight for the interpolation formula used for nontrainable parameters. Defaults to 1..
- **trainable** (*bool, optional*) – Whether input parameters is trainable. If set to False, momentum_nontrainable will be used. Defaults to True.

Returns Interpolation result.

Return type torch.Tensor

every_n_iters(*runner: mmengine.runner.Runner*, *n: int*)

This is the function to perform every n iterations.

Parameters

- **runner** (*Runner*) – runner used to drive the whole pipeline
- **n** (*int*) – the number of iterations

Returns the latest iterations

Return type int

after_train_iter(*runner: mmengine.runner.Runner*, *batch_idx: int*, *data_batch: DATA_BATCH = None*, *outputs: Optional[dict] = None*) → None

This is the function to perform after each training iteration.

Parameters

- **runner** (*Runner*) – runner to drive the pipeline
- **batch_idx** (*int*) – the id of batch
- **data_batch** (*DATA_BATCH, optional*) – data batch. Defaults to None.
- **outputs** (*Optional[dict], optional*) – output. Defaults to None.

before_run(*runner: mmengine.runner.Runner*)

This is the function perform before each run.

Parameters **runner** (*Runner*) – runner used to drive the whole pipeline

Raises **RuntimeError** – error message

class mmedit.engine.hooks.**GenIterTimerHook**

Bases: mmengine.hooks.IterTimerHook

GenIterTimerHooks inherits from mmengine.hooks.IterTimerHook and overwrites self._after_iter().

This hooks should be used along with *mmedit.engine.runner.GenValLoop* and *mmedit.engine.runner.GenTestLoop*.

_after_iter(*runner*, *batch_idx: int*, *data_batch: DATA_BATCH = None*, *outputs: Optional[Union[dict, Sequence[mmengine.structures.BaseDataElement]]] = None*, *mode: str = 'train'*) → None

Calculating time for an iteration and updating "time" HistoryBuffer of runner.message_hub. If *mode* is 'train', we take *runner.max_iters* as the total iterations and calculate the rest time. If *mode* in *val* or *test*, we use *runner.val_loop.total_length* or *runner.test_loop.total_length* as total number of iterations. If you want to know how *total_length* is calculated, please refers to *mmedit.engine.runner.GenValLoop.run()* and *mmedit.engine.runner.GenTestLoop.run()*.

Parameters

- **runner** (*Runner*) – The runner of the training validation and testing process.
- **batch_idx** (*int*) – The index of the current batch in the loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. Defaults to None.
- **outputs** (*dict or sequence, optional*) – Outputs from model. Defaults to None.
- **mode** (*str*) – Current mode of runner. Defaults to 'train'.

class mmedit.engine.hooks.**PGGANFetchDataHook**

Bases: mmengine.hooks.Hook

PGGAN Fetch Data Hook.

Parameters **interval** (*int, optional*) – The interval of calling this hook. If set to -1, the visualization hook will not be called. Defaults to 1.

before_train_iter(*runner*, *batch_idx: int*, *data_batch: DATA_BATCH = None*) → None

All subclasses should override this method, if they need any operations before each training iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*dict or tuple or list, optional*) – Data from dataloader.

update_dataloader(*dataloader: torch.utils.data.dataloader.DataLoader*, *curr_scale: int*) → Optional[torch.utils.data.dataloader.DataLoader]

Updata the data loader.

Parameters

- **dataloader** (*DataLoader*) – The dataloader to be updated.
- **curr_scale** (*int*) – The current scale of the generated image.

Returns

The updated dataloader. If the dataloader do not need to update, return None.

Return type Optional[DataLoader]

class mmedit.engine.hooks.**PickleDataHook**(*output_dir*, *data_name_list*, *interval=- 1*, *before_run=False*, *after_run=False*, *filename_tmpl='iter_{}.pkl'*)

Bases: mmengine.hooks.Hook

Pickle Useful Data Hook.

This hook will be used in SinGAN training for saving some important data that will be used in testing or inference.

Parameters

- **output_dir** (*str*) – The output path for saving pickled data.
- **data_name_list** (*list[str]*) – The list contains the name of results in outputs dict.
- **interval** (*int*) – The interval of calling this hook. If set to -1, the PickleDataHook will not be called during training. Default: -1.
- **before_run** (*bool, optional*) – Whether to save before running. Defaults to False.
- **after_run** (*bool, optional*) – Whether to save after running. Defaults to False.
- **filename_tmpl** (*str, optional*) – Format string used to save images. The output file name will be formatted as this args. Defaults to 'iter_{}.pkl'.

after_run(*runner*)

The behavior after each train iteration.

Parameters **runner** (*object*) – The runner.

before_run(*runner*)

The behavior after each train iteration.

Parameters **runner** (*object*) – The runner.

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

The behavior after each train iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. Defaults to None.

_pickle_data(*runner: mmengine.runner.Runner*)

Save target data to pickle file.

Parameters **runner** (*Runner*) – The runner of the training process.

_get_numpy_data(*data: Tuple[List[torch.Tensor], torch.Tensor, int]*) → Tuple[List[numpy.ndarray], numpy.ndarray, int]

Convert tensor or list of tensor to numpy or list of numpy.

Parameters **data** (*Tuple[List[Tensor], Tensor, int]*) – Data to be converted.

Returns Converted data.

Return type Tuple[List[np.ndarray], np.ndarray, int]

class mmedit.engine.hooks.**ReduceLRSchedulerHook**(*val_metric: str = None*, *by_epoch=True*, *interval=1*)

Bases: mmengine.hooks.ParamSchedulerHook

A hook to update learning rate.

Parameters

- **val_metric** (*str*) – The metric of validation. If val_metric is not None, we check val_metric to reduce learning. Default: None.
- **by_epoch** (*bool*) – Whether to update by epoch. Default: True.
- **interval** (*int*) – The interval of iterations to update. Default: 1.

_calculate_average_value()

after_train_epoch(*runner: mmengine.runner.Runner*)

Call step function for each scheduler after each train epoch.

Parameters **runner** (*Runner*) – The runner of the training process.

after_train_iter(*runner: mmengine.runner.Runner*, *batch_idx: int*, *data_batch: DATA_BATCH = None*, *outputs: Optional[dict] = None*) → None

Call step function for each scheduler after each iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. In order to keep this interface consistent with other hooks, we keep data_batch here. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. In order to keep this interface consistent with other hooks, we keep data_batch here. Defaults to None.

after_val_epoch(*runner*, *metrics: Optional[Dict[str, float]] = None*)

Call step function for each scheduler after each validation epoch.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **metrics** (*dict, optional*) – The metrics of validation. Default: None.

class mmedit.engine.hooks.**BasicVisualizationHook**(*interval: dict = {}*, *on_train=False*, *on_val=True*, *on_test=True*)

Bases: mmengine.hooks.Hook

Basic hook that invoke visualizers during validation and test.

Parameters

- **interval** (*int | dict*) – Visualization interval. Default: {}.
- **on_train** (*bool*) – Whether to call hook during train. Default to False.
- **on_val** (*bool*) – Whether to call hook during validation. Default to True.
- **on_test** (*bool*) – Whether to call hook during test. Default to True.

priority = NORMAL

_after_iter(*runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]]*, *outputs: Optional[Sequence[mmengine.structures.BaseDataElement]]*, *mode=None*) → None

Show or Write the predicted results.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the test loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. Defaults to None.
- **outputs** (*Sequence[BaseDataElement], optional*) – Outputs from model. Defaults to None.

class mmedit.engine.hooks.**GenVisualizationHook**(*interval: int = 1000*, *vis_kwargs_list: Tuple[List[dict], dict] = None*, *fixed_input: bool = True*, *n_samples: Optional[int] = 64*, *n_row: Optional[int] = 8*, *message_hub_vis_kwargs: Optional[Tuple[str, dict, List[str], List[Dict]]] = None*, *save_at_test: bool = True*, *max_save_at_test: int = 100*, *test_vis_keys: Optional[Union[str, List[str]]] = None*, *show: bool = False*, *wait_time: float = 0*)

Bases: mmengine.hooks.Hook

Generation Visualization Hook. Used to visual output samples in training, validation and testing. In this hook, we use a list called *sample_kwargs_list* to control how to generate samples and how to visualize them. Each element in *sample_kwargs_list*, called *sample_kwargs*, may contains the following keywords:

- **Required key words:**
  - **'type': Value must be string. Denotes what kind of sampler is used to** generate image. Refers to *get_sampler()*.
- **Optional key words (If not passed, will use the default value):**
  - 'n_rows': Value must be int. The number of images in one row.
  - 'num_samples': Value must be int. The number of samples to visualize.
  - **'vis_mode': Value must be string. How to visualize the generated** samples (e.g. image, gif).
  - **'fixed_input': Value must be bool. Whether use the fixed input** during the loop.
  - 'draw_gt': Value must be bool. Whether save the real images.
  - **'target_keys': Value must be string or list of string. The keys of** the target image to visualize.
  - **'name': Value must be string. If not passed, will use** *sample_kwargs['type']* as default.

For convenience, we also define a group of alias of samplers' type for models supported in MMEditing. Refers to *:attr:self.SAMPLER_TYPE_MAPPING*.

Example

```
>>> # for GAN models
>>> custom_hooks = [
>>>     dict(
>>>         type='GenVisualizationHook',
>>>         interval=1000,
>>>         fixed_input=True,
>>>         vis_kwargs_list=dict(type='GAN', name='fake_img'))]
>>> # for Translation models
>>> custom_hooks = [
>>>     dict(
>>>         type='GenVisualizationHook',
>>>         interval=10,
>>>         fixed_input=False,
>>>         vis_kwargs_list=[dict(type='Translation',
>>>                               name='translation_train',
>>>                               n_samples=6, draw_gt=True,
>>>                               n_rows=3),
>>>                          dict(type='TranslationVal',
>>>                               name='translation_val',
>>>                               n_samples=16, draw_gt=True,
>>>                               n_rows=4)])]
```

NOTE: user-defined vis_kwargs > vis_kwargs_mapping > hook init args

Parameters

- **interval** (*int*) – Visualization interval. Default: 1000.
- **sampler_kwargs_list** (*Tuple[List[dict], dict]*) – The list of sampling behavior to generate images.
- **fixed_input** (*bool*) – The default action of whether use fixed input to generate samples during the loop. Defaults to True.
- **n_samples** (*Optional[int]*) – The default value of number of samples to visualize. Defaults to 64.
- **n_row** (*Optional[int]*) – The default value of number of images in each row in the visualization results. Defaults to 8.
- **(Optional[Tuple[str** (*message_hub_vis_kwargs*) – List[Dict]]]): Key arguments visualize images in message hub. Defaults to None.
- **dict** – List[Dict]]]): Key arguments visualize images in message hub. Defaults to None.
- **List[str]** – List[Dict]]]): Key arguments visualize images in message hub. Defaults to None.

:param [List[Dict]]]): Key arguments visualize images in message hub.] Defaults to None.

Parameters

- **save_at_test** (*bool*) – Whether save images during test. Defaults to True.
- **max_save_at_test** (*int*) – Maximum number of samples saved at test time. If None is passed, all samples will be saved. Defaults to 100.
- **show** (*bool*) – Whether to display the drawn image. Default to False.

- **wait_time** (*float*) – The interval of show (s). Defaults to 0.

priority = NORMAL

VIS_KWARGS_MAPPING

after_val_iter(*runner: mmengine.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs*) → None

GenVisualizationHook do not support visualize during validation.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the test loop.
- **data_batch** (*Sequence[dict], optional*) – Data from dataloader. Defaults to None.
- **outputs** – outputs of the generation model

after_test_iter(*runner: mmengine.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs*)

Visualize samples after test iteraiton.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the test loop.
- **data_batch** (*dict, optional*) – Data from dataloader. Defaults to None.
- **outputs** – outputs of the generation model Defaults to None.

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

Visualize samples after train iteration.

Parameters

- **runner** (*Runner*) – The runner of the training process.
- **batch_idx** (*int*) – The index of the current batch in the train loop.
- **data_batch** (*dict*) – Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. Defaults to None.

vis_sample(*runner: mmengine.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs: Optional[dict] = None*) → None

Visualize samples.

Parameters

- **runner** (*Runner*) – The runner conatians model to visualize.
- **batch_idx** (*int*) – The index of the current batch in loop.
- **data_batch** (*dict*) – Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) – Outputs from model. Defaults to None.

vis_from_message_hub(*batch_idx: int*, *color_order: str*, *target_mean: Sequence[Union[float, int]]*, *target_std: Sequence[Union[float, int]]*)

Visualize samples from message hub.

Parameters

- **batch_idx** (*int*) – The index of the current batch in the test loop.
- **color_order** (*str*) – The color order of generated images.
- **target_mean** (*Sequence[Union[float, int]]*) – The original mean of the image tensor before preprocessing. Image will be re-shifted to `target_mean` before visualizing.
- **target_std** (*Sequence[Union[float, int]]*) – The original std of the image tensor before preprocessing. Image will be re-scaled to `target_std` before visualizing.

1.45 mmedit.engine.optimizers

1.45.1 Package Contents

Classes

MultiOptimWrapperConstructor	OptimizerConstructor for GAN models. This class construct optimizer for
PGGANOptimWrapperConstructor	OptimizerConstructor for PGGAN models. Set optimizers for each
SinGANOptimWrapperConstructor	OptimizerConstructor for SinGAN models. Set optimizers for each

class mmedit.engine.optimizers.**MultiOptimWrapperConstructor**(*optim_wrapper_cfg: dict*, *paramwise_cfg=None*)

OptimizerConstructor for GAN models. This class construct optimizer for the submodules of the model separately, and return a `mmengine.optim.OptimWrapperDict`.

Example

```
>>> # build GAN model
>>> model = dict(
>>>     type='GANModel',
>>>     num_classes=10,
>>>     generator=dict(type='Generator'),
>>>     discriminator=dict(type='Discriminator'))
>>> gan_model = MODELS.build(model)
>>> # build constructor
>>> optim_wrapper = dict(
>>>     constructor='MultiOptimWrapperConstructor',
>>>     generator=dict(
>>>         type='OptimWrapper',
>>>         accumulative_counts=1,
>>>         optimizer=dict(type='Adam', lr=0.0002,
>>>                        betas=(0.5, 0.999))),
>>>     discriminator=dict(
>>>         type='OptimWrapper',
>>>         accumulative_counts=1,
>>>         optimizer=dict(type='Adam', lr=0.0002,
>>>                            betas=(0.5, 0.999))))
```

(continues on next page)

(continued from previous page)

```
>>> optim_dict_builder = MultiOptimWrapperConstructor(optim_wrapper)
>>> # build optim wrapper dict
>>> optim_wrapper_dict = optim_dict_builder(gan_model)
```

Parameters

- **optim_wrapper_cfg_dict** (*dict*) – Config of the optimizer wrapper.
- **paramwise_cfg** (*dict*) – Config of parameter-wise settings. Default: None.

__call__(*module: torch.nn.Module*) → mmengine.optim.OptimWrapperDict

Build optimizer and return a optimizer_wrapper_dict.

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

OptimizerConstructor for PGGAN models. Set optimizers for each stage of PGGAN. All submodule must be contained in a `torch.nn.ModuleList` named 'blocks'. And we access each submodule by *MODEL.blocks[SCALE]*, where *MODLE* is generator or discriminator, and the scale is the index of the resolution scale.

More detail about the resolution scale and naming rule please refers to `PGGANGenerator` and `PGGANDiscriminator`.

Example

```
>>> # build PGGAN model
>>> model = dict(
>>>     type='ProgressiveGrowingGAN',
>>>     data_preprocessor=dict(type='GANDataPreprocessor'),
>>>     noise_size=512,
>>>     generator=dict(type='PGGANGenerator', out_scale=1024,
>>>                    noise_size=512),
>>>     discriminator=dict(type='PGGANDiscriminator', in_scale=1024),
>>>     nkimgs_per_scale={
>>>         '4': 600,
>>>         '8': 1200,
>>>         '16': 1200,
>>>         '32': 1200,
>>>         '64': 1200,
>>>         '128': 1200,
>>>         '256': 1200,
>>>         '512': 1200,
>>>         '1024': 12000,
>>>     },
>>>     transition_kimgs=600,
>>>     ema_config=dict(interval=1))
>>> pggan = MODELS.build(model)
>>> # build constructor
>>> optim_wrapper = dict(
>>>     generator=dict(optimizer=dict(type='Adam', lr=0.001,
>>>                                   betas=(0., 0.99))),
```

(continues on next page)

(continued from previous page)

```
>>>     discriminator=dict(
>>>         optimizer=dict(type='Adam', lr=0.001, betas=(0., 0.99))),
>>>     lr_schedule=dict(
>>>         generator={
>>>             '128': 0.0015,
>>>             '256': 0.002,
>>>             '512': 0.003,
>>>             '1024': 0.003
>>>         },
>>>         discriminator={
>>>             '128': 0.0015,
>>>             '256': 0.002,
>>>             '512': 0.003,
>>>             '1024': 0.003
>>>         }))
>>> optim_wrapper_dict_builder = PGGANOptimWrapperConstructor(
>>>     optim_wrapper)
>>> # build optim wrapper dict
>>> optim_wrapper_dict = optim_wrapper_dict_builder(pggan)
```

Parameters

- **optim_wrapper_cfg** (*dict*) – Config of the optimizer wrapper.
- **paramwise_cfg** (*Optional[dict]*) – Parameter-wise options.

__call__(*module: torch.nn.Module*) → mmengine.optim.OptimWrapperDict

Build optimizer and return a optimizerwrapperdict.

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

OptimizerConstructor for SinGAN models. Set optimizers for each submodule of SinGAN. All submodule must be contained in a `torch.nn.ModuleList` named 'blocks'. And we access each submodule by *MODEL.blocks[SCALE]*, where *MODLE* is generator or discriminator, and the scale is the index of the resolution scale.

More detail about the resolution scale and naming rule please refers to `SinGANMultiScaleGenerator` and `SinGANMultiScaleDiscriminator`.

Example

```
>>> # build SinGAN model
>>> model = dict(
>>>     type='SinGAN',
>>>     data_preprocessor=dict(
>>>         type='GANDataPreprocessor',
>>>         non_image_keys=['input_sample']),
>>>     generator=dict(
>>>         type='SinGANMultiScaleGenerator',
>>>         in_channels=3,
>>>         out_channels=3,
```

(continues on next page)

(continued from previous page)

```
>>>         num_scales=2),
>>>     discriminator=dict(
>>>         type='SinGANMultiScaleDiscriminator',
>>>         in_channels=3,
>>>         num_scales=3))
>>> singan = MODELS.build(model)
>>> # build constructor
>>> optim_wrapper = dict(
>>>     generator=dict(optimizer=dict(type='Adam', lr=0.0005,
>>>                                   betas=(0.5, 0.999))),
>>>     discriminator=dict(
>>>         optimizer=dict(type='Adam', lr=0.0005,
>>>                        betas=(0.5, 0.999))))
>>> optim_wrapper_dict_builder = SinGANOptimWrapperConstructor(
>>>     optim_wrapper)
>>> # build optim wrapper dict
>>> optim_wrapper_dict = optim_wrapper_dict_builder(singan)
```

Parameters

- **optim_wrapper_cfg** (*dict*) – Config of the optimizer wrapper.
- **paramwise_cfg** (*Optional[dict]*) – Parameter-wise options.

__call__(*module: torch.nn.Module*) → mmengine.optim.OptimWrapperDict

Build optimizer and return a optimizerwrapperdict.

1.46 mmedit.engine.runner

1.46.1 Package Contents

Classes

GenTestLoop	Validation loop for generative models. This class support evaluate
GenValLoop	Validation loop for generative models. This class support evaluate
GenLogProcessor	GenLogProcessor inherits from `mmengine.runner.LogProcessor` and
MultiTestLoop	Loop for validation multi-datasets.
MultiValLoop	Loop for validation multi-datasets.

class mmedit.engine.runner.**GenTestLoop**(*runner: mmengine.runner.Runner*, *dataloader: Union[torch.utils.data.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.Evaluator, Dict, List]*)

Bases: `mmengine.runner.TestLoop`

Validation loop for generative models. This class support evaluate metrics with different sample mode.

Parameters

- **runner** (*Runner*) – A reference of runner.
- **dataloader** (*Dataloader or dict*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) – Used for computing metrics.

run()

Launch validation. The evaluation process consists of four steps.

1. Prepare pre-calculated items for all metrics by calling `self.evaluator.prepare_metrics()`.
2. Get a list of metrics-sampler pair. Each pair contains a list of metrics with the same sampler mode and a shared sampler.
3. Generate images for the each metrics group. Loop for elements in each sampler and feed to the model as input by calling `self.run_iter()`.
4. Evaluate all metrics by calling `self.evaluator.evaluate()`.

run_iter(*idx*, *data_batch: dict*, *metrics: Sequence[mmengine.evaluator.BaseMetric]*)

Iterate one mini-batch and feed the output to corresponding *metrics*.

Parameters

- **idx** (*int*) – Current idx for the input data.
- **data_batch** (*dict*) – Batch of data from dataloader.
- **metrics** (*Sequence[BaseMetric]*) – Specific metrics to evaluate.

class mmedit.engine.runner.**GenValLoop**(*runner: mmengine.runner.Runner*, *dataloader: Union[torch.utils.data.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.Evaluator, Dict, List]*)

Bases: `mmengine.runner.ValLoop`

Validation loop for generative models. This class support evaluate metrics with different sample mode.

Parameters

- **runner** (*Runner*) – A reference of runner.
- **dataloader** (*Dataloader or dict*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) – Used for computing metrics.

run()

Launch validation. The evaluation process consists of four steps.

1. Prepare pre-calculated items for all metrics by calling `self.evaluator.prepare_metrics()`.
2. Get a list of metrics-sampler pair. Each pair contains a list of metrics with the same sampler mode and a shared sampler.
3. Generate images for the each metrics group. Loop for elements in each sampler and feed to the model as input by calling `self.run_iter()`.
4. Evaluate all metrics by calling `self.evaluator.evaluate()`.

run_iter(*idx*, *data_batch: dict*, *metrics: Sequence[mmengine.evaluator.BaseMetric]*)

Iterate one mini-batch and feed the output to corresponding *metrics*.

Parameters

- **idx** (*int*) – Current idx for the input data.
- **data_batch** (*dict*) – Batch of data from dataloader.

- **metrics** (*Sequence[BaseMetric]*) – Specific metrics to evaluate.

class mmedit.engine.runner.**GenLogProcessor**(*window_size=10*, *by_epoch=True*, *custom_cfg: Optional[List[dict]] = None*, *num_digits: int = 4*)

Bases: mmengine.runner.LogProcessor

GenLogProcessor inherits from mmengine.runner.LogProcessor and overwrites self.get_log_after_iter().

This log processor should be used along with *mmedit.engine.runner.GenValLoop* and *mmedit.engine.runner.GenTestLoop*.

get_log_after_iter(*runner*, *batch_idx: int*, *mode: str*) → Tuple[dict, str]

Format log string after training, validation or testing epoch.

If *mode* is in 'val' or 'test', we use *runner.val_loop.total_length* and *runner.test_loop.total_length* as the total number of iterations shown in log. If you want to know how *total_length* is calculated, please refers to *mmedit.engine.runner.GenValLoop.run()* and *mmedit.engine.runner.GenTestLoop.run()*.

Parameters

- **runner** (*Runner*) – The runner of training phase.
- **batch_idx** (*int*) – The index of the current batch in the current loop.
- **mode** (*str*) – Current mode of runner, train, test or val.

Returns

Formatted log dict/string which will be recorded by runner.message_hub and runner.visualizer.

Return type Tuple(dict, str)

get_log_after_epoch(*runner*, *batch_idx: int*, *mode: str*) → Tuple[dict, str]

Format log string after validation or testing epoch.

We use *runner.val_loop.total_length* and *runner.test_loop.total_length* as the total number of iterations shown in log. If you want to know how *total_length* is calculated, please refers to *mmedit.engine.runner.GenValLoop.run()* and *mmedit.engine.runner.GenTestLoop.run()*.

Parameters

- **runner** (*Runner*) – The runner of validation/testing phase.
- **batch_idx** (*int*) – The index of the current batch in the current loop.
- **mode** (*str*) – Current mode of runner.

Returns Formatted log dict/string which will be recorded by runner.message_hub and runner.visualizer.

Return type Tuple(dict, str)

class mmedit.engine.runner.**MultiTestLoop**(*runner*, *dataloader: Union[torch.utils.data.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.Evaluator, Dict, List]*, *fp16: bool = False*)

Bases: mmengine.runner.base_loop.BaseLoop

Loop for validation multi-datasets.

Parameters

- **runner** (*Runner*) – A reference of runner.

- **dataloader** (*Dataloader or dict*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*Evaluator or dict or list*) – Used for computing metrics.
- **fp16** (*bool*) – Whether to enable fp16 validation. Defaults to False.

run()

Launch test.

run_iter(*idx: int*, *data_batch: Sequence[dict]*)

Iterate one mini-batch.

Parameters

- **idx** (*int*) – The index of the current batch in the loop.
- **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

class mmedit.engine.runner.**MultiValLoop**(*runner*, *dataloader: Union[torch.utils.data.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.Evaluator, Dict, List]*, *fp16: bool = False*)

Bases: mmengine.runner.base_loop.BaseLoop

Loop for validation multi-datasets.

Parameters

- **runner** (*Runner*) – A reference of runner.
- **dataloader** (*list[Dataloader or dic]*) – A dataloader object or a dict to build a dataloader.
- **evaluator** (*list[]*) – Used for computing metrics.
- **fp16** (*bool*) – Whether to enable fp16 validation. Defaults to False.

run()

Launch validation.

run_iter(*idx: int*, *data_batch: Sequence[dict]*)

Iterate one mini-batch.

Parameters

- **idx** (*int*) – The index of the current batch in the loop.
- **data_batch** (*Sequence[dict]*) – Batch of data from dataloader.

1.47 mmedit.engine.schedulers

1.47.1 Package Contents

Classes

LinearLrInterval	Linear learning rate scheduler for image generation.
ReduceLR	Decays the learning rate of each parameter group by linearly changing

class mmedit.engine.schedulers.**LinearLrInterval**(**args*, *interval=1*, ***kwargs*)

Bases: mmengine.optim.LinearLR

Linear learning rate scheduler for image generation.

In the beginning, the learning rate is 'start_factor' defined in mmengine. We give a target learning rate 'end_factor' and a start point 'begin'. If :attr:self.by_epoch is True, 'begin' is calculated by epoch, otherwise, calculated by iteration." Before 'begin', we fix learning rate as 'start_factor'; After 'begin', we linearly update learning rate to 'end_factor'.

Parameters **interval** (*int*) – The interval to update the learning rate. Default: 1.

_get_value()

Compute value using chainable form of the scheduler.

class mmedit.engine.schedulers.**ReduceLR**(*optimizer*, *mode: str = 'min'*, *factor: float = 0.1*, *patience: int = 10*, *threshold: float = 0.0001*, *threshold_mode: str = 'rel'*, *cooldown: int = 0*, *min_lr: float = 0.0*, *eps: float = 1e-08*, ***kwargs*)

Bases: mmengine.optim._ParamScheduler

Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: end.

Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

Note:

The learning rate of each parameter group will be update at regular intervals.

Parameters

- **optimizer** (*Optimizer or OptimWrapper*) – Wrapped optimizer.
- **mode** (*str, optional*) – One of *min*, *max*. In *min* mode, lr will be reduced when the quantity monitored has stopped decreasing; in *max* mode it will be reduced when the quantity monitored has stopped increasing. Default: 'min'.
- **factor** (*float, optional*) – Factor by which the learning rate will be reduced. new_lr = lr * factor. Default: 0.1.
- **patience** (*int, optional*) – Number of epochs with no improvement after which learning rate will be reduced. For example, if *patience = 2*, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn't improved then. Default: 10.
- **threshold** (*float, optional*) – Threshold for measuring the new optimum, to only focus on significant changes. Default: 1e-4.
- **threshold_mode** (*str, optional*) – One of *rel*, *abs*. In *rel* mode, dynamic_threshold = best * (1 + threshold) in 'max' mode or best * (1 - threshold) in *min* mode. In *abs* mode, dynamic_threshold = best + threshold in *max* mode or best - threshold in *min* mode. Default: 'rel'.
- **cooldown** (*int, optional*) – Number of epochs to wait before resuming normal operation after lr has been reduced. Default: 0.
- **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. Default: 0.

- **eps** (*float, optional*) – Minimal decay applied to lr. If the difference between new and old lr is smaller than eps, the update is ignored. Default: 1e-8.
- **begin** (*int*) – Step at which to start updating the learning rate. Defaults to 0.
- **end** (*int*) – Step at which to stop updating the learning rate.
- **last_step** (*int*) – The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) – Whether the scheduled learning rate is updated by epochs. Defaults to True.

property in_cooldown

_get_value()

Compute value using chainable form of the scheduler.

_init_is_better(*mode*)

_reset()

is_better(*a*, *best*)

1.48 mmedit.models.base_archs

1.48.1 Package Contents

Classes

AllGatherLayer	All gather layer with backward propagation path.
ASPP	ASPP module from DeepLabV3.
SpatialTemporalEnsemble	Apply spatial and temporal ensemble and compute outputs.
SimpleGatedConvModule	Simple Gated Convolutional Module.
ImgNormalize	Normalize images with the given mean and std value.
LinearModule	A linear block that contains linear/norm/activation layers.
MultiLayerDiscriminator	Multilayer Discriminator.
PatchDiscriminator	A PatchGAN discriminator.
ResNet	General ResNet.
DepthwiseSeparableConvModule	Depthwise separable convolution module.
SimpleEncoderDecoder	Simple encoder-decoder model from matting.
SoftMaskPatchDiscriminator	A Soft Mask-Guided PatchGAN discriminator.
ResidualBlockNoBN	Residual block without BN.
PixelShufflePack	Pixel Shuffle upsample layer.
VGG16	Customized VGG16 Encoder.

Functions

conv2d(input, weight[, bias, stride, padding, ...])	
conv_transpose2d(input, weight[, bias, stride, ...])	
pixel_unshuffle(→ torch.Tensor)	Down-sample by pixel unshuffle.

class mmedit.models.base_archs.**AllGatherLayer**(**args*, ***kwargs*)

Bases: torch.autograd.Function

All gather layer with backward propagation path.

Indeed, this module is to make dist.all_gather() in the backward graph. Such kind of operation has been widely used in Moco and other contrastive learning algorithms.

static forward(*ctx*, *x*)

Forward function.

static backward(*ctx*, **grad_outputs*)

Backward function.

class mmedit.models.base_archs.**ASPP**(*in_channels: int*, *out_channels: int = 256*, *mid_channels: int = 256*, *dilations: Sequence[int] = (12, 24, 36)*, *conv_cfg: Optional[dict] = None*, *norm_cfg: Optional[dict] = dict(type='BN')*, *act_cfg: Optional[dict] = dict(type='ReLU')*, *separable_conv: bool = False*)

Bases: torch.nn.Module

ASPP module from DeepLabV3.

The code is adopted from https://github.com/pytorch/vision/blob/master/torchvision/models/ segmentation/deeplabv3.py

For more information about the module: "Rethinking Atrous Convolution for Semantic Image Segmentation".

Parameters

- **in_channels** (*int*) – Input channels of the module.
- **out_channels** (*int*) – Output channels of the module. Default: 256.
- **mid_channels** (*int*) – Output channels of the intermediate ASPP conv modules. Default: 256.
- **dilations** (*Sequence[int]*) – Dilation rate of three ASPP conv module. Default: [12, 24, 36].
- **conv_cfg** (*dict*) – Config dict for convolution layer. If "None", nn.Conv2d will be applied. Default: None.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: dict(type='BN').
- **act_cfg** (*dict*) – Config dict for activation layer. Default: dict(type='ReLU').
- **separable_conv** (*bool*) – Whether replace normal conv with depthwise separable conv which is faster. Default: False.

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

Forward function for ASPP module.

Parameters **x** (*Tensor*) – Input tensor with shape (N, C, H, W).

Returns Output tensor.

Return type Tensor

mmedit.models.base_archs.**conv2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*)

mmedit.models.base_archs.**conv_transpose2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *dilation=1*)

mmedit.models.base_archs.**pixel_unshuffle**(*x: torch.Tensor*, *scale: int*) → torch.Tensor

Down-sample by pixel unshuffle.

Parameters

- **x** (*Tensor*) – Input tensor.
- **scale** (*int*) – Scale factor.

Returns Output tensor.

Return type Tensor

class mmedit.models.base_archs.**SpatialTemporalEnsemble**(*is_temporal_ensemble: Optional[bool] = False*)

Bases: torch.nn.Module

Apply spatial and temporal ensemble and compute outputs.

Parameters **is_temporal_ensemble** (*bool, optional*) – Whether to apply ensemble temporally. If True, the sequence will also be flipped temporally. If the input is an image, this argument must be set to False. Default: False.

_transform(*imgs: torch.Tensor*, *mode: str*) → torch.Tensor

Apply spatial transform (flip, rotate) to the images.

Parameters

- **imgs** (*torch.Tensor*) – The images to be transformed/
- **mode** (*str*) – The mode of transform. Supported values are 'vertical', 'horizontal', and 'transpose', corresponding to vertical flip, horizontal flip, and rotation, respectively.

Returns Output of the model with spatial ensemble applied.

Return type torch.Tensor

spatial_ensemble(*imgs: torch.Tensor*, *model: torch.nn.Module*) → torch.Tensor

Apply spatial ensemble.

Parameters

- **imgs** (*torch.Tensor*) – The images to be processed by the model. Its size should be either (n, t, c, h, w) or (n, c, h, w).
- **model** (*nn.Module*) – The model to process the images.

Returns Output of the model with spatial ensemble applied.

Return type torch.Tensor

forward(*imgs: torch.Tensor*, *model: torch.nn.Module*) → torch.Tensor

Apply spatial and temporal ensemble.

Parameters

- **imgs** (*torch.Tensor*) – The images to be processed by the model. Its size should be either (n, t, c, h, w) or (n, c, h, w).
- **model** (*nn.Module*) – The model to process the images.

Returns Output of the model with spatial ensemble applied.

Return type torch.Tensor

class mmedit.models.base_archs.**SimpleGatedConvModule**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *feat_act_cfg: Optional[dict] = dict(type='ELU')*, *gate_act_cfg: Optional[dict] = dict(type='Sigmoid')*, ***kwargs*)

Bases: torch.nn.Module

Simple Gated Convolutional Module.

This module is a simple gated convolutional module. The detailed formula is:

$$y = \phi(conv1(x)) * \sigma(conv2(x)),$$

where *phi* is the feature activation function and *sigma* is the gate activation function. In default, the gate activation function is sigmoid.

Parameters

- **in_channels** (*int*) – Same as nn.Conv2d.
- **out_channels** (*int*) – The number of channels of the output feature. Note that *out_channels* in the conv module is doubled since this module contains two convolutions for feature and gate separately.
- **kernel_size** (*int or tuple[int]*) – Same as nn.Conv2d.
- **feat_act_cfg** (*dict*) – Config dict for feature activation layer. Default: dict(type='ELU').
- **gate_act_cfg** (*dict*) – Config dict for gate activation layer. Default: dict(type='Sigmoid').
- **kwargs** (*keyword arguments*) – Same as *ConvModule*.

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

Forward Function.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.base_archs.**ImgNormalize**(*pixel_range: float*, *img_mean: Tuple[float, float, float]*, *img_std: Tuple[float, float, float]*, *sign: int = - 1*)

Bases: torch.nn.Conv2d

Normalize images with the given mean and std value.

Based on Conv2d layer, can work in GPU.

Parameters

- **pixel_range** (*float*) – Pixel range of feature.
- **img_mean** (*Tuple[float]*) – Image mean of each channel.

- **img_std** (*Tuple[float]*) – Image std of each channel.
- **sign** (*int*) – Sign of bias. Default -1.

class mmedit.models.base_archs.**LinearModule**(*in_features: int*, *out_features: int*, *bias: bool = True*, *act_cfg: Optional[dict] = dict(type='ReLU')*, *inplace: bool = True*, *with_spectral_norm: bool = False*, *order: Tuple[str, str] = ('linear', 'act')*)

Bases: torch.nn.Module

A linear block that contains linear/norm/activation layers.

For low level vision, we add spectral norm and padding layer.

Parameters

- **in_features** (*int*) – Same as nn.Linear.
- **out_features** (*int*) – Same as nn.Linear.
- **bias** (*bool*) – Same as nn.Linear. Default: True.
- **act_cfg** (*dict*) – Config dict for activation layer, "relu" by default.
- **inplace** (*bool*) – Whether to use inplace mode for activation. Default: True.
- **with_spectral_norm** (*bool*) – Whether use spectral norm in linear module. Default: False.
- **order** (*tuple[str]*) – The order of linear/activation layers. It is a sequence of "linear", "norm" and "act". Examples are ("linear", "act") and ("act", "linear").

init_weights() → None

Init weights for the model.

forward(*x: torch.Tensor*, *activate: Optional[bool] = True*) → torch.Tensor

Forward Function.

Parameters

- **x** (*torch.Tensor*) – Input tensor with shape of $(n, *, c)$. Same as torch.nn.Linear.
- **activate** (*bool, optional*) – Whether to use activation layer. Defaults to True.

Returns Same as torch.nn.Linear.

Return type torch.Tensor

class mmedit.models.base_archs.**MultiLayerDiscriminator**(*in_channels: int*, *max_channels: int*, *num_convs: int = 5*, *fc_in_channels: Optional[int] = None*, *fc_out_channels: int = 1024*, *kernel_size: int = 5*, *conv_cfg: Optional[dict] = None*, *norm_cfg: Optional[dict] = None*, *act_cfg: Optional[dict] = dict(type='ReLU')*, *out_act_cfg: Optional[dict] = dict(type='ReLU')*, *with_input_norm: bool = True*, *with_out_convs: bool = False*, *with_spectral_norm: bool = False*, ***kwargs*)

Bases: torch.nn.Module

Multilayer Discriminator.

This is a commonly used structure with stacked multiply convolution layers.

Parameters

- **in_channels** (*int*) – Input channel of the first input convolution.
- **max_channels** (*int*) – The maximum channel number in this structure.
- **num_conv** (*int*) – Number of stacked intermediate convs (including input conv but excluding output conv). Default to 5.
- **fc_in_channels** (*int | None*) – Input dimension of the fully connected layer. If *fc_in_channels* is None, the fully connected layer will be removed. Default to None.
- **fc_out_channels** (*int*) – Output dimension of the fully connected layer. Default to 1024.
- **kernel_size** (*int*) – Kernel size of the conv modules. Default to 5.
- **conv_cfg** (*dict*) – Config dict to build conv layer.
- **norm_cfg** (*dict*) – Config dict to build norm layer.
- **act_cfg** (*dict*) – Config dict for activation layer, "relu" by default.
- **out_act_cfg** (*dict*) – Config dict for output activation, "relu" by default.
- **with_input_norm** (*bool*) – Whether add normalization after the input conv. Default to True.
- **with_out_convs** (*bool*) – Whether add output convs to the discriminator. The output convs contain two convs. The first out conv has the same setting as the intermediate convs but a stride of 1 instead of 2. The second out conv is a conv similar to the first out conv but reduces the number of channels to 1 and has no activation layer. Default to False.
- **with_spectral_norm** (*bool*) – Whether use spectral norm after the conv layers. Default to False.
- **kwargs** (*keyword arguments*) –

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

Forward Function.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w') or (n, c).

Return type torch.Tensor

init_weights(*pretrained: Optional[str] = None*) → None

Init weights for models.

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Defaults to None.

class mmedit.models.base_archs.**PatchDiscriminator**(*in_channels: int*, *base_channels: int = 64*, *num_conv: int = 3*, *norm_cfg: dict = dict(type='BN')*, *init_cfg: Optional[dict] = dict(type='normal', gain=0.02)*)

Bases: torch.nn.Module

A PatchGAN discriminator.

Parameters

- **in_channels** (*int*) – Number of channels in input images.

- **base_channels** (*int*) – Number of channels at the first conv layer. Default: 64.
- **num_conv** (*int*) – Number of stacked intermediate convs (excluding input and output conv). Default: 3.
- **norm_cfg** (*dict*) – Config dict to build norm layer. Default: *dict(type='BN')*.
- **init_cfg** (*dict*) – Config dict for initialization. *type*: The name of our initialization method. Default: 'normal'. *gain*: Scaling factor for normal, xavier and orthogonal. Default: 0.02.

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

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

init_weights(*pretrained: Optional[str] = None*) → None

Initialize weights for the model.

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Default: None.

class mmedit.models.base_archs.**ResNet**(*depth: int*, *in_channels: int = 3*, *stem_channels: int = 64*, *base_channels: int = 64*, *num_stages: int = 4*, *strides: Sequence[int] = (1, 2, 2, 2)*, *dilations: Sequence[int] = (1, 1, 2, 4)*, *deep_stem: bool = False*, *avg_down: bool = False*, *frozen_stages: int = - 1*, *act_cfg: dict = dict(type='ReLU')*, *conv_cfg: Optional[dict] = None*, *norm_cfg: dict = dict(type='BN')*, *with_cp: bool = False*, *multi_grid: Optional[Sequence[int]] = None*, *contract_dilation: bool = False*, *zero_init_residual: bool = True*)

Bases: torch.nn.Module

General ResNet.

This class is adopted from https://github.com/open-mmlab/mmsegmentation/blob/master/mmseg/models/backbones/resnet.py.

Parameters

- **depth** (*int*) – Depth of resnet, from {18, 34, 50, 101, 152}.
- **in_channels** (*int*) – Number of input image channels. Default" 3.
- **stem_channels** (*int*) – Number of stem channels. Default: 64.
- **base_channels** (*int*) – Number of base channels of res layer. Default: 64.
- **num_stages** (*int*) – Resnet stages, normally 4.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage. Default: (1, 2, 2, 2).
- **dilations** (*Sequence[int]*) – Dilation of each stage. Default: (1, 1, 2, 4).
- **deep_stem** (*bool*) – Replace 7x7 conv in input stem with 3 3x3 conv. Default: False.
- **avg_down** (*bool*) – Use AvgPool instead of stride conv when downsampling in the bottleneck. Default: False.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Default: -1.

- **act_cfg** (*dict*) – Dictionary to construct and config activation layer. Default: dict(type='ReLU').
- **conv_cfg** (*dict*) – Dictionary to construct and config convolution layer. Default: None.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Default: dict(type='BN').
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Default: False.
- **multi_grid** (*Sequence[int]|None*) – Multi grid dilation rates of last stage. Default: None.
- **contract_dilation** (*bool*) – Whether contract first dilation of each layer Default: False.
- **zero_init_residual** (*bool*) – Whether to use zero init for last norm layer in resblocks to let them behave as identity. Default: True.

property norm1: torch.nn.Module

normalization layer after the second convolution layer

Type nn.Module

arch_settings

_make_stem_layer(*in_channels: int*, *stem_channels: int*) → None

Make stem layer for ResNet.

_make_layer(*block: BasicBlock*, *planes: int*, *blocks: int*, *stride: int = 1*, *dilation: int = 1*) → torch.nn.Module

_nostride_dilate(*m: torch.nn.Module*, *dilate: int*) → None

init_weights(*pretrained: Optional[str] = None*) → None

Init weights for the model.

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Defaults to None.

_freeze_stages() → None

Freeze stages param and norm stats.

forward(*x: torch.Tensor*) → List[torch.Tensor]

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (N, C, H, W).

Returns Output tensor.

Return type Tensor

class mmedit.models.base_archs.**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: Optional[dict] = 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*)

Bases: torch.nn.Module

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*) – 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. Default: 1.
- **padding** (*int or tuple[int]*) – Same as nn.Conv2d. Default: 0.
- **dilation** (*int or tuple[int]*) – Same as nn.Conv2d. 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

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (N, C, H, W).

Returns Output tensor.

Return type Tensor

class mmedit.models.base_archs.**SimpleEncoderDecoder**(*encoder: dict*, *decoder: dict*, *init_cfg: Optional[dict] = None*)

Bases: mmengine.model.BaseModule

Simple encoder-decoder model from matting.

Parameters

- **encoder** (*dict*) – Config of the encoder.
- **decoder** (*dict*) – Config of the decoder.
- **init_cfg** (*dict, optional*) – Initialization config dict.

forward(**args*, ***kwargs*) → torch.Tensor

Forward function.

Returns The output tensor of the decoder.

Return type Tensor

class mmedit.models.base_archs.**SoftMaskPatchDiscriminator**(*in_channels: int*, *base_channels: Optional[int] = 64*, *num_conv: Optional[int] = 3*, *norm_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = dict(type='normal', gain=0.02)*, *with_spectral_norm: Optional[bool] = False*)

Bases: mmengine.model.BaseModule

A Soft Mask-Guided PatchGAN discriminator.

Parameters

- **in_channels** (*int*) – Number of channels in input images.
- **base_channels** (*int, optional*) – Number of channels at the first conv layer. Default: 64.
- **num_conv** (*int, optional*) – Number of stacked intermediate convs (excluding input and output conv). Default: 3.
- **norm_cfg** (*dict, optional*) – Config dict to build norm layer. Default: None.
- **init_cfg** (*dict, optional*) – Config dict for initialization. *type*: The name of our initialization method. Default: 'normal'. *gain*: Scaling factor for normal, xavier and orthogonal. Default: 0.02.
- **with_spectral_norm** (*bool, optional*) – Whether use spectral norm after the conv layers. Default: False.

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

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

init_weights() → None

Initialize weights for the model.

class mmedit.models.base_archs.**ResidualBlockNoBN**(*mid_channels: int = 64*, *res_scale: float = 1.0*)

Bases: torch.nn.Module

Residual block without BN.

It has a style of:

```
---Conv-ReLU-Conv-+-
 |________________|
```

Parameters

- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64.
- **res_scale** (*float*) – Used to scale the residual before addition. Default: 1.0.

init_weights() → None

Initialize weights for ResidualBlockNoBN.

Initialization methods like *kaiming_init* are for VGG-style modules. For modules with residual paths, using smaller std is better for stability and performance. We empirically use 0.1. See more details in "ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks"

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

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.base_archs.**PixelShufflePack**(*in_channels: int*, *out_channels: int*, *scale_factor: int*, *upsample_kernel: int*)

Bases: torch.nn.Module

Pixel Shuffle upsample layer.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.
- **scale_factor** (*int*) – Upsample ratio.
- **upsample_kernel** (*int*) – Kernel size of Conv layer to expand channels.

Returns Upsampled feature map.

init_weights() → None

Initialize weights for PixelShufflePack.

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

Forward function for PixelShufflePack.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.base_archs.**VGG16**(*in_channels: int*, *batch_norm: Optional[bool] = False*, *aspp: Optional[bool] = False*, *dilations: Optional[List[int]] = None*, *init_cfg: Optional[dict] = None*)

Bases: mmengine.model.BaseModule

Customized VGG16 Encoder.

A 1x1 conv is added after the original VGG16 conv layers. The indices of max pooling layers are returned for unpooling layers in decoders.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **batch_norm** (*bool, optional*) – Whether use nn.BatchNorm2d. Default to False.
- **aspp** (*bool, optional*) – Whether use ASPP module after the last conv layer. Default to False.
- **dilations** (*list[int], optional*) – Atrous rates of ASPP module. Default to None.
- **init_cfg** (*dict, optional*) – Initialization config dict.

_make_layer(*inplanes: int*, *planes: int*, *convs_layers: int*) → torch.nn.Module

init_weights() → None

Init weights for the model.

forward(*x: torch.Tensor*) → Dict[str, torch.Tensor]

Forward function for ASPP module.

Parameters **x** (*Tensor*) – Input tensor with shape (N, C, H, W).

Returns Dict containing output tensor and maxpooling indices.

Return type dict

1.49 mmedit.models.base_models

1.49.1 Package Contents

Classes

ExponentialMovingAverage	Implements the exponential moving average (EMA) of the model.
RampUpEMA	Implements the exponential moving average with ramping up momentum.
BaseConditionalGAN	Base class for Conditional GAM models.
BaseEditModel	Base model for image and video editing.
BaseGAN	Base class for GAN models.
BaseMattor	Base class for trimap-based matting models.
BaseTranslationModel	Base Translation Model.
BasicInterpolator	Basic model for video interpolation.
OneStageInpaintor	Standard one-stage inpaintor with commonly used losses.
TwoStageInpaintor	Standard two-stage inpaintor with commonly used losses. A two-stage

class mmedit.models.base_models.**ExponentialMovingAverage**(*model: torch.nn.Module*, *momentum: float = 0.0002*, *interval: int = 1*, *device: Optional[torch.device] = None*, *update_buffers: bool = False*)

Bases: mmengine.model.BaseAveragedModel

Implements the exponential moving average (EMA) of the model.

All parameters are updated by the formula as below:

$$Xema_{t+1} = (1 - momentum) * Xema_t + momentum * X_t$$

Parameters

- **model** (*nn.Module*) – The model to be averaged.
- **momentum** (*float*) – The momentum used for updating ema parameter. Defaults to 0.0002. Ema's parameter are updated with the formula $averaged\_param = (1 - momentum) * averaged\_param + momentum * source\_param$.
- **interval** (*int*) – Interval between two updates. Defaults to 1.
- **device** (*torch.device, optional*) – If provided, the averaged model will be stored on the device. Defaults to None.
- **update_buffers** (*bool*) – if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

avg_func(*averaged_param: torch.Tensor*, *source_param: torch.Tensor*, *steps: int*) → None

Compute the moving average of the parameters using exponential moving average.

Parameters

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

_load_from_state_dict(*state_dict: dict*, *prefix: str*, *local_metadata: dict*, *strict: bool*, *missing_keys: list*, *unexpected_keys: list*, *error_msgs: List[str]*) → None

Overrides `nn.Module._load_from_state_dict` to support loading `state_dict` without wrap ema module with `BaseAveragedModel`.

In OpenMMLab 1.0, model will not wrap ema submodule with `BaseAveragedModel`, and the ema weight key in *state_dict* will miss *module* prefix. Therefore, `BaseAveragedModel` need to automatically add the `module` prefix if the corresponding key in `state_dict` misses it.

Parameters

- **state_dict** (*dict*) – A dict containing parameters and persistent buffers.
- **prefix** (*str*) – The prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this module.
- **strict** (*bool*) – Whether to strictly enforce that the keys in `state_dict` with `prefix` match the names of parameters and buffers in this module
- **missing_keys** (*List[str]*) – if `strict=True`, add missing keys to this list
- **unexpected_keys** (*List[str]*) – if `strict=True`, add unexpected keys to this list
- **error_msgs** (*List[str]*) – error messages should be added to this list, and will be reported together in `load_state_dict()`.

sync_buffers(*model: torch.nn.Module*) → None

Copy buffer from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

sync_parameters(*model: torch.nn.Module*) → None

Copy buffer and parameters from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

class mmedit.models.base_models.**RampUpEMA**(*model: torch.nn.Module*, *interval: int = 1*, *ema_kimg: int = 10*, *ema_rampup: float = 0.05*, *batch_size: int = 32*, *eps: float = 1e-08*, *start_iter: int = 0*, *device: Optional[torch.device] = None*, *update_buffers: bool = False*)

Bases: `mmengine.model.BaseAveragedModel`

Implements the exponential moving average with ramping up momentum.

Ref: https://github.com/NVlabs/stylegan3/blob/master/training/training_loop.py # noqa

Parameters

- **model** (*nn.Module*) – The model to be averaged.
- **interval** (*int*) – Interval between two updates. Defaults to 1.
- **ema_kimg** (*int, optional*) – EMA kimgs. Defaults to 10.
- **ema_rampup** (*float, optional*) – Ramp up rate. Defaults to 0.05.
- **batch_size** (*int, optional*) – Global batch size. Defaults to 32.
- **eps** (*float, optional*) – Ramp up epsilon. Defaults to 1e-8.

- **start_iter** (*int, optional*) – EMA start iter. Defaults to 0.
- **device** (*torch.device, optional*) – If provided, the averaged model will be stored on the device. Defaults to None.
- **update_buffers** (*bool*) – if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

static rampup(*steps*, *ema_kimg=10*, *ema_rampup=0.05*, *batch_size=4*, *eps=1e-08*)

Ramp up ema momentum.

Ref: https://github.com/NVlabs/stylegan3/blob/a5a69f58294509598714d1e88c9646c3d7c6ec94/training/training_loop.py#L300-L308 # noqa

Parameters

- **steps** –
- **ema_kimg** (*int, optional*) – Half-life of the exponential moving average of generator weights. Defaults to 10.
- **ema_rampup** (*float, optional*) – EMA ramp-up coefficient.If set to None, then rampup will be disabled. Defaults to 0.05.
- **batch_size** (*int, optional*) – Total batch size for one training iteration. Defaults to 4.
- **eps** (*float, optional*) – Epsiolon to avoid batch_size divided by zero. Defaults to 1e-8.

Returns Updated momentum.

Return type dict

avg_func(*averaged_param: torch.Tensor*, *source_param: torch.Tensor*, *steps: int*) → None

Compute the moving average of the parameters using exponential moving average.

Parameters

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

_load_from_state_dict(*state_dict: dict*, *prefix: str*, *local_metadata: dict*, *strict: bool*, *missing_keys: list*, *unexpected_keys: list*, *error_msgs: List[str]*) → None

Overrides nn.Module._load_from_state_dict to support loading state_dict without wrap ema module with BaseAveragedModel.

In OpenMMLab 1.0, model will not wrap ema submodule with BaseAveragedModel, and the ema weight key in *state_dict* will miss *module* prefix. Therefore, BaseAveragedModel need to automatically add the module prefix if the corresponding key in state_dict misses it.

Parameters

- **state_dict** (*dict*) – A dict containing parameters and persistent buffers.
- **prefix** (*str*) – The prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this module.
- **strict** (*bool*) – Whether to strictly enforce that the keys in state_dict with prefix match the names of parameters and buffers in this module
- **missing_keys** (*List[str]*) – if strict=True, add missing keys to this list

- **unexpected_keys** (*List[str]*) – if `strict=True`, add unexpected keys to this list
- **error_msgs** (*List[str]*) – error messages should be added to this list, and will be reported together in `load_state_dict()`.

sync_buffers(*model: torch.nn.Module*) → None

Copy buffer from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

sync_parameters(*model: torch.nn.Module*) → None

Copy buffer and parameters from model to averaged model.

Parameters **model** (*nn.Module*) – The model whose parameters will be averaged.

class mmedit.models.base_models.**BaseConditionalGAN**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *num_classes: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Bases: `mmedit.models.base_models.base_gan.BaseGAN`

Base class for Conditional GAM models.

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or `GenDataPreprocessor`.
- **generator_steps** (*int*) – The number of times the generator is completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.
- **noise_size** (*Optional[int]*) – Size of the input noise vector. Default to None.
- **num_classes** (*Optional[int]*) – The number classes you would like to generate. Defaults to None.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

label_fn(*label: mmedit.utils.typing.LabelVar = None*, *num_batches: int = 1*) → torch.Tensor

Sampling function for label. There are three scenarios in this function:

- If *label* is a callable function, sample *num_batches* of labels with passed *label*.
- If *label* is *None*, sample *num_batches* of labels in range of *[0, self.num_classes-1]* uniformly.
- If *label* is a *torch.Tensor*, check the range of the tensor is in *[0, self.num_classes-1]*. If all values are in valid range, directly return *label*.

Parameters

- **label** (*Union[Tensor, Callable, List[int], None]*) – You can directly give a batch of label through a `torch.Tensor` or offer a callable function to sample a batch of label data. Otherwise, the `None` indicates to use the default label sampler. Defaults to *None*.
- **num_batches** (*int, optional*) – The number of batches label want to sample. If *label* is a Tensor, this will be ignored. Defaults to 1.

Returns Sampled label tensor.

Return type Tensor

data_sample_to_label(*data_sample: List[*mmedit.structures.EditDataSample*]*) → Optional[torch.Tensor]

Get labels from input *data_sample* and pack to *torch.Tensor*. If no label is found in the passed *data_sample*, *None* would be returned.

Parameters **data_sample** (*List[EditDataSample]*) – Input data samples.

Returns Packed label tensor.

Return type Optional[torch.Tensor]

static _get_valid_num_classes(*num_classes: Optional[int]*, *generator: ModelType*, *discriminator: Optional[ModelType]*) → int

Try to get the value of *num_classes* from input, *generator* and *discriminator* and check the consistency of these values. If no conflict is found, return the *num_classes*.

Parameters

- **num_classes** (*Optional[int]*) – *num_classes* passed to *BaseConditionalGAN_refactor*'s initialize function.
- **generator** (*ModelType*) – The config or the model of generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of discriminator.

Returns The number of classes to be generated.

Return type int

forward(*inputs: mmedit.utils.typing.ForwardInputs*, *data_samples: Optional[list] = None*, *mode: Optional[str] = None*) → List[*mmedit.structures.EditDataSample*]

Sample images with the given inputs. If forward mode is 'ema' or 'orig', the image generated by corresponding generator will be returned. If forward mode is 'ema/orig', images generated by original generator and EMA generator will both be returned in a dict.

Parameters

- **inputs** (*ForwardInputs*) – Dict containing the necessary information (e.g. noise, num_batches, mode) to generate image.
- **data_samples** (*Optional[list]*) – Data samples collated by `data_preprocessor`. Defaults to None.
- **mode** (*Optional[str]*) – *mode* is not used in *BaseConditionalGAN*. Defaults to None.

Returns Generated images or image dict.

Return type List[EditDataSample]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A dict of tensor for logging.

Return type Dict[str, Tensor]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A dict of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.base_models.**BaseEditModel**(*generator: dict*, *pixel_loss: dict*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = None*, *data_preprocessor: Optional[dict] = None*)

Bases: mmengine.model.BaseModel

Base model for image and video editing.

It must contain a generator that takes frames as inputs and outputs an interpolated frame. It also has a pixel-wise loss for training.

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.

init_cfg

Initialization config dict.

Type dict, optional

data_preprocessor

Used for pre-processing data sampled by dataloader to the format accepted by *forward()*. Default: None.

Type BaseDataPreprocessor

forward(*inputs: torch.Tensor*, *data_samples: Optional[List[*mmedit.structures.EditDataSample*]] = None*, *mode: str = 'tensor'*, ***kwargs*) → Union[torch.Tensor, List[*mmedit.structures.EditDataSample*], dict]

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

`forward` method of BaseModel is an abstract method, its subclasses must implement this method.

Accepts `inputs` and `data_samples` processed by *data_preprocessor*, and returns results according to mode arguments.

During non-distributed training, validation, and testing process, `forward` will be called by `BaseModel.train_step`, `BaseModel.val_step` and `BaseModel.val_step` directly.

During distributed data parallel training process, `MMSeparateDistributedDataParallel.train_step` will first call `DistributedDataParallel.forward` to enable automatic gradient synchronization, and then call `forward` to get training loss.

Parameters

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

Returns

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

Return type ForwardResults

convert_to_datasample(*inputs: List[*mmedit.structures.EditDataSample*]*, *data_samples: List[*mmedit.structures.EditDataSample*]*) → List[*mmedit.structures.EditDataSample*]

forward_tensor(*inputs: torch.Tensor*, *data_samples: Optional[List[*mmedit.structures.EditDataSample*]] = None*, ***kwargs*) → torch.Tensor

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.

Returns result of simple forward.

Return type Tensor

forward_inference(*inputs: torch.Tensor*, *data_samples: Optional[List[*mmedit.structures.EditDataSample*]] = None*, ***kwargs*) → List[*mmedit.structures.EditDataSample*]

Forward inference. Returns predictions of validation, testing, and simple inference.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.

Returns predictions.

Return type List[EditDataSample]

forward_train(*inputs: torch.Tensor*, *data_samples: Optional[List[*mmedit.structures.EditDataSample*]] = None*, ***kwargs*) → Dict[str, torch.Tensor]

Forward training. Returns dict of losses of training.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.

Returns Dict of losses.

Return type dict

class mmedit.models.base_models.**BaseGAN**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Bases: mmengine.model.BaseModel

Base class for GAN models.

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or GenDataPreprocessor.
- **generator_steps** (*int*) – The number of times the generator is completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

property generator_steps: int

The number of times the generator is completely updated before the discriminator is updated.

Type int

property discriminator_steps: int

The number of times the discriminator is completely updated before the generator is updated.

Type int

property device: torch.device

Get current device of the model.

Returns The current device of the model.

Return type torch.device

property with_ema_gen: bool

Whether the GAN adopts exponential moving average.

Returns

If *True*, means this GAN model is adopted to exponential moving average and vice versa.

Return type bool

static gather_log_vars(*log_vars_list: List[Dict[str, torch.Tensor]]*) → Dict[str, torch.Tensor]

Gather a list of log_vars. :param log_vars_list: List[Dict[str, Tensor]]

Returns Dict[str, Tensor]

_init_loss(*loss_config: Optional[Dict] = None*) → None

Initialize customized loss modules.

If loss_config is a dict, we allow kinds of value for each field.

1. ***loss_config* is None: Users will implement all loss calculations** in their own function. Weights for each loss terms are hard coded.
2. ***loss_config* is dict of scalar or string: Users will implement all** loss calculations and use passed *loss_config* to control the weight or behavior of the loss calculation. Users will unpack and use each field in this dict by themself.

   loss_config = dict(gp_norm_mode='HWC', gp_loss_weight=10)
3. ***loss_config* is dict of dict: Each field in *loss_config* will** used to build a corresponding loss module. And use loss calculation function predefined by *BaseGAN* to calculate the loss.

   loss_config = dict()

Example

loss_config = dict(# *BaseGAN* pre-defined fields gan_loss=dict(type='GANLoss', gan_type='wgan-logistic-ns'), disc_auxiliary_loss=dict(

type='R1GradientPenalty', loss_weight=10. / 2., interval=2, norm_mode='HWC', data_info=dict(

real_data='real_imgs', discriminator='disc')),

gen_auxiliary_loss=dict(type='GeneratorPathRegularizer', loss_weight=2, pl_batch_shrink=2, interval=g_reg_interval, data_info=dict(

generator='gen', num_batches='batch_size')),

user-defined field for loss weights or loss calculation my_loss_2=dict(weight=2, norm_mode='L1'), my_loss_3=2, my_loss_4_norm_type='L2')

Parameters **loss_config** (*Optional[Dict], optional*) – Loss config used to build loss modules or define the loss weights. Defaults to None.

noise_fn(*noise: mmedit.utils.typing.NoiseVar = None*, *num_batches: int = 1*)

Sampling function for noise. There are three scenarios in this function:

- If *noise* is a callable function, sample *num_batches* of noise with passed *noise*.
- If *noise* is *None*, sample *num_batches* of noise from gaussian distribution.
- If *noise* is a *torch.Tensor*, directly return *noise*.

Parameters

- **noise** (*Union[Tensor, Callable, List[int], None]*) – You can directly give a batch of label through a `torch.Tensor` or offer a callable function to sample a batch of label data. Otherwise, the `None` indicates to use the default noise sampler. Defaults to *None*.
- **num_batches** (*int, optional*) – The number of batches label want to sample. If *label* is a Tensor, this will be ignored. Defaults to 1.

Returns Sampled noise tensor.

Return type Tensor

_init_ema_model(*ema_config: dict*)

Initialize a EMA model corresponding to the given *ema_config*. If *ema_config* is an empty dict or *None*, EMA model will not be initialized.

Parameters **ema_config** (*dict*) – Config to initialize the EMA model.

_get_valid_model(*batch_inputs: mmedit.utils.typing.ForwardInputs*) → str

Try to get the valid forward model from inputs.

- If forward model is defined in *batch_inputs*, it will be used as forward model.
- If forward model is not defined in *batch_inputs*, 'ema' will returned if **:property:`with_ema_gen`** is true. Otherwise, 'orig' will be returned.

Parameters **batch_inputs** (*ForwardInputs*) – Inputs passed to *forward()*.

Returns

Forward model to generate image. ('orig', 'ema' or 'ema/orig').

Return type str

forward(*inputs: mmedit.utils.typing.ForwardInputs*, *data_samples: Optional[list] = None*, *mode: Optional[str] = None*) → mmedit.utils.typing.SampleList

Sample images with the given inputs. If forward mode is 'ema' or 'orig', the image generated by corresponding generator will be returned. If forward mode is 'ema/orig', images generated by original generator and EMA generator will both be returned in a dict.

Parameters

- **batch_inputs** (*ForwardInputs*) – Dict containing the necessary information (e.g. noise, num_batches, mode) to generate image.
- **data_samples** (*Optional[list]*) – Data samples collated by `data_preprocessor`. Defaults to None.

- **mode** (*Optional[str]*) – *mode* is not used in *BaseGAN*. Defaults to None.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

val_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data.

Calls `self.data_preprocessor(data)` and `self(inputs, data_sample, mode=None)` in order. Return the generated results which will be passed to evaluator.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type SampleList

test_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type List[EditDataSample]

train_step(*data: dict*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train GAN model. In the training of GAN models, generator and discriminator are updated alternatively. In MMEditing's design, *self.train_step* is called with data input. Therefore we always update discriminator, whose updating is relay on real data, and then determine if the generator needs to be updated based on the current number of iterations. More details about whether to update generator can be found in `should_gen_update()`.

Parameters

- **data** (*dict*) – Data sampled from dataloader.
- **optim_wrapper** (*OptimWrapperDict*) – OptimWrapperDict instance contains OptimWrapper of generator and discriminator.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

_get_gen_loss(*out_dict*)

_get_disc_loss(*out_dict*)

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Training function for discriminator. All GANs should implement this function by themselves.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.base_models.**BaseMattor**(*data_preprocessor: Union[dict, mmengine.config.Config]*, *backbone: dict*, *init_cfg: Optional[dict] = None*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*)

Bases: `mmengine.model.BaseModel`

Base class for trimap-based matting models.

A matting model must contain a backbone which produces *pred_alpha*, a dense prediction with the same height and width of input image. In some cases (such as DIM), the model has a refiner which refines the prediction of the backbone.

Subclasses should overwrite the following functions:

- `_forward_train()`, to return a loss
- `_forward_test()`, to return a prediction
- `_forward()`, to return raw tensors

For test, this base class provides functions to resize inputs and post-process pred_alphas to get predictions

Parameters

- **backbone** (*dict*) – Config of backbone.
- **data_preprocessor** (*dict*) – Config of data_preprocessor. See `MattorPreprocessor` for details.
- **init_cfg** (*dict, optional*) – Initialization config dict.
- **train_cfg** (*dict*) – Config of training. Customized by subclassesCustomized bu In `train_cfg`, `train_backbone` should be specified. If the model has a refiner, `train_refiner` should be specified.
- **test_cfg** (*dict*) – Config of testing. In `test_cfg`, If the model has a refiner, `train_refiner` should be specified.

resize_inputs(*batch_inputs: torch.Tensor*) → torch.Tensor

Pad or interpolate images and trimaps to multiple of given factor.

restore_size(*pred_alpha: torch.Tensor*, *data_sample:* mmedit.structures.EditDataSample) → torch.Tensor

Restore the predicted alpha to the original shape.

The shape of the predicted alpha may not be the same as the shape of original input image. This function restores the shape of the predicted alpha.

Parameters

- **pred_alpha** (*torch.Tensor*) – A single predicted alpha of shape (1, H, W).
- **data_sample** (*EditDataSample*) – Data sample containing original shape as meta data.

Returns The reshaped predicted alpha.

Return type torch.Tensor

postprocess(*batch_pred_alpha: torch.Tensor*, *data_samples: List[*mmedit.structures.EditDataSample*]*) → List[*mmedit.structures.EditDataSample*]

Post-process alpha predictions.

This function contains the following steps:

1. Restore padding or interpolation
2. Mask alpha prediction with trimap
3. Clamp alpha prediction to 0-1
4. Convert alpha prediction to uint8
5. Pack alpha prediction into EditDataSample

Currently only batch_size 1 is actually supported.

Parameters

- **batch_pred_alpha** (*torch.Tensor*) – A batch of predicted alpha of shape (N, 1, H, W).
- **data_samples** (*List[EditDataSample]*) – List of data samples.

Returns

A list of predictions. Each data sample contains a pred_alpha, which is a torch.Tensor with dtype=uint8, device=cuda:0

Return type List[EditDataSample]

forward(*inputs: torch.Tensor*, *data_samples: DataSamples = None*, *mode: str = 'tensor'*) → List[*mmedit.structures.EditDataSample*]

General forward function.

Parameters

- **inputs** (*torch.Tensor*) – A batch of inputs. with image and trimap concatenated alone channel dimension.
- **data_samples** (*List[EditDataSample], optional*) – A list of data samples, containing: - Ground-truth alpha / foreground / background to compute loss - other meta information
- **mode** (*str*) – mode should be one of `loss`, `predict` and `tensor`. Default: 'tensor'.
  - `loss`: Called by `train_step` and return loss `dict` used for logging
  - `predict`: Called by `val_step` and `test_step` and return list of `BaseDataElement` results used for computing metric.
  - `tensor`: Called by custom use to get `Tensor` type results.

Returns Sequence of predictions packed into EditDataElement

Return type List[EditDataElement]

convert_to_datasample(*inputs: DataSamples*, *data_samples: List[*mmedit.structures.EditDataSample*]*) → List[*mmedit.structures.EditDataSample*]

class mmedit.models.base_models.**BaseTranslationModel**(*generator*, *discriminator*, *default_domain: str*, *reachable_domains: List[str]*, *related_domains: List[str]*, *data_preprocessor*, *discriminator_steps: int = 1*, *disc_init_steps: int = 0*, *real_img_key: str = 'real_img'*, *loss_config: Optional[dict] = None*)

Bases: mmengine.model.BaseModel

Base Translation Model.

Translation models can transfer images from one domain to another. Domain information like *default_domain*, *reachable_domains* are needed to initialize the class. And we also provide query functions like *is_domain_reachable*, *get_other_domains*.

You can get a specific generator based on the domain, and by specifying *target_domain* in the forward function, you can decide the domain of generated images. Considering the difference among different image translation models, we only provide the external interfaces mentioned above. When you implement image translation with a specific method, you can inherit both *BaseTranslationModel* and the method (e.g BaseGAN) and implement abstract methods.

Parameters

- **default_domain** (*str*) – Default output domain.
- **reachable_domains** (*list[str]*) – Domains that can be generated by the model.
- **related_domains** (*list[str]*) – Domains involved in training and testing. *reachable_domains* must be contained in *related_domains*. However, related_domains may contain source domains that are used to retrieve source images from data_batch but not in reachable_domains.
- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.
- **disc_init_steps** (*int*) – The number of initial steps used only to train discriminators.

init_weights(*pretrained=None*)

Initialize weights for the model.

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Default: None.

get_module(*module*)

Get *nn.ModuleDict* to fit the *MMDistributedDataParallel* interface.

Parameters **module** (*MMDistributedDataParallel | nn.ModuleDict*) – The input module that needs processing.

Returns The ModuleDict of multiple networks.

Return type nn.ModuleDict

forward(*img*, *test_mode=False*, ***kwargs*)

Forward function.

Parameters

- **img** (*tensor*) – Input image tensor.
- **test_mode** (*bool*) – Whether in test mode or not. Default: False.

- **kwargs** (*dict*) – Other arguments.

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

Forward function for training.

Parameters

- **img** (*tensor*) – Input image tensor.
- **target_domain** (*str*) – Target domain of output image.
- **kwargs** (*dict*) – Other arguments.

Returns Forward results.

Return type dict

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

Forward function for testing.

Parameters

- **img** (*tensor*) – Input image tensor.
- **target_domain** (*str*) – Target domain of output image.
- **kwargs** (*dict*) – Other arguments.

Returns Forward results.

Return type dict

is_domain_reachable(*domain*)

Whether image of this domain can be generated.

get_other_domains(*domain*)

get other domains.

_get_target_generator(*domain*)

get target generator.

_get_target_discriminator(*domain*)

get target discriminator.

translation(*image*, *target_domain=None*, ***kwargs*)

Translation Image to target style.

Parameters

- **image** (*tensor*) – Image tensor with a shape of (N, C, H, W).
- **target_domain** (*str, optional*) – Target domain of output image. Default to None.

Returns Image tensor of target style.

Return type dict

class mmedit.models.base_models.**BasicInterpolator**(*generator: dict*, *pixel_loss: dict*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *required_frames: int = 2*, *step_frames: int = 1*, *init_cfg: Optional[dict] = None*, *data_preprocessor: Optional[dict] = None*)

Bases: mmedit.models.base_models.base_edit_model.BaseEditModel

Basic model for video interpolation.

It must contain a generator that takes frames as inputs and outputs an interpolated frame. It also has a pixel-wise loss for training.

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **required_frames** (*int*) – Required frames in each process. Default: 2
- **step_frames** (*int*) – Step size of video frame interpolation. Default: 1
- **init_cfg** (*dict, optional*) – The weight initialized config for `BaseModule`.
- **data_preprocessor** (*dict, optional*) – The pre-process config of `BaseDataPreprocessor`.

init_cfg

Initialization config dict.

Type dict, optional

data_preprocessor

Used for pre-processing data sampled by dataloader to the format accepted by `forward()`.

Type `BaseDataPreprocessor`

split_frames(*input_tensors: torch.Tensor*) → torch.Tensor

split input tensors for inference.

Parameters **input_tensors** (*Tensor*) – Tensor of input frames with shape [1, t, c, h, w]

Returns Split tensor with shape [t-1, 2, c, h, w]

Return type Tensor

static merge_frames(*input_tensors: torch.Tensor*, *output_tensors: torch.Tensor*) → list

merge input frames and output frames.

Interpolate a frame between the given two frames.

Merged from [[in1, in2], [in2, in3], [in3, in4], . . .] [[out1], [out2], [out3], . . .]

to [in1, out1, in2, out2, in3, out3, in4, . . .]

Parameters

- **input_tensors** (*Tensor*) – The input frames with shape [n, 2, c, h, w]
- **output_tensors** (*Tensor*) – The output frames with shape [n, 1, c, h, w].

Returns The final frames.

Return type list[np.array]

class mmedit.models.base_models.**OneStageInpaintor**(*data_preprocessor: Union[dict, mmengine.config.Config]*, *encdec: dict*, *disc: Optional[dict] = None*, *loss_gan: Optional[dict] = None*, *loss_gp: Optional[dict] = None*, *loss_disc_shift: Optional[dict] = None*, *loss_composed_percep: Optional[dict] = None*, *loss_out_percep: bool = False*, *loss_l1_hole: Optional[dict] = None*, *loss_l1_valid: Optional[dict] = None*, *loss_tv: Optional[dict] = None*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = None*)

Bases: mmengine.model.BaseModel

Standard one-stage inpaintor with commonly used losses.

An inpaintor must contain an encoder-decoder style generator to inpaint masked regions. A discriminator will be adopted when adversarial training is needed.

In this class, we provide a common interface for inpaintors. For other inpaintors, only some funcs may be modified to fit the input style or training schedule.

Parameters

- **data_preprocessor** (*dict*) – Config of data_preprocessor.
- **encdec** (*dict*) – Config for encoder-decoder style generator.
- **disc** (*dict*) – Config for discriminator.
- **loss_gan** (*dict*) – Config for adversarial loss.
- **loss_gp** (*dict*) – Config for gradient penalty loss.
- **loss_disc_shift** (*dict*) – Config for discriminator shift loss.
- **loss_composed_percep** (*dict*) – Config for perceptual and style loss with composed image as input.
- **loss_out_percep** (*dict*) – Config for perceptual and style loss with direct output as input.
- **loss_l1_hole** (*dict*) – Config for l1 loss in the hole.
- **loss_l1_valid** (*dict*) – Config for l1 loss in the valid region.
- **loss_tv** (*dict*) – Config for total variation loss.
- **train_cfg** (*dict*) – Configs for training scheduler. *disc_step* must be contained for indicates the discriminator updating steps in each training step.
- **test_cfg** (*dict*) – Configs for testing scheduler.
- **init_cfg** (*dict, optional*) – Initialization config dict.

forward(*inputs: torch.Tensor*, *data_samples: Optional[mmedit.utils.SampleList]*, *mode: str = 'tensor'*) → FORWARD_RETURN_TYPE

Forward function.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.

- **mode** (*str*) – mode should be one of `loss`, `predict` and `tensor`. Default: 'tensor'.
  - `loss`: Called by `train_step` and return loss `dict` used for logging
  - `predict`: Called by `val_step` and `test_step` and return list of `BaseDataElement` results used for computing metric.
  - `tensor`: Called by custom use to get `Tensor` type results.

Returns

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

Return type ForwardResults

train_step(*data: List[dict]*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → dict

Train step function.

In this function, the inpaintor will finish the train step following the pipeline:

1. get fake res/image
2. optimize discriminator (if have)
3. optimize generator

If *self.train_cfg.disc_step > 1*, the train step will contain multiple iterations for optimizing discriminator with different input data and only one iteration for optimizing gerator after *disc_step* iterations for discriminator.

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns

Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

abstract forward_train(**args*, ***kwargs*) → None

Forward function for training.

In this version, we do not use this interface.

forward_train_d(*data_batch: torch.Tensor*, *is_real: bool*, *is_disc: bool*) → dict

Forward function in discriminator training step.

In this function, we compute the prediction for each data batch (real or fake). Meanwhile, the standard gan loss will be computed with several proposed losses for stable training.

Parameters

- **data_batch** (*torch.Tensor*) – Batch of real data or fake data.
- **is_real** (*bool*) – If True, the gan loss will regard this batch as real data. Otherwise, the gan loss will regard this batch as fake data.

- **is_disc** (*bool*) – If True, this function is called in discriminator training step. Otherwise, this function is called in generator training step. This will help us to compute different types of adversarial loss, like LSGAN.

Returns Contains the loss items computed in this function.

Return type dict

generator_loss(*fake_res: torch.Tensor*, *fake_img: torch.Tensor*, *gt: torch.Tensor*, *mask: torch.Tensor*, *masked_img: torch.Tensor*) → Tuple[dict, dict]

Forward function in generator training step.

In this function, we mainly compute the loss items for generator with the given (fake_res, fake_img). In general, the *fake_res* is the direct output of the generator and the *fake_img* is the composition of direct output and ground-truth image.

Parameters

- **fake_res** (*torch.Tensor*) – Direct output of the generator.
- **fake_img** (*torch.Tensor*) – Composition of *fake_res* and ground-truth image.
- **gt** (*torch.Tensor*) – Ground-truth image.
- **mask** (*torch.Tensor*) – Mask image.
- **masked_img** (*torch.Tensor*) – Composition of mask image and ground-truth image.

Returns Dict contains the results computed within this function for visualization and dict contains the loss items computed in this function.

Return type tuple(dict)

forward_tensor(*inputs: torch.Tensor*, *data_samples: mmedit.utils.SampleList*) → Tuple[torch.Tensor, torch.Tensor]

Forward function in tensor mode.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns

Direct output of the generator and composition of *fake_res* and ground-truth image.

Return type tuple

forward_test(*inputs: torch.Tensor*, *data_samples: mmedit.utils.SampleList*) → mmedit.utils.SampleList

Forward function for testing.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns

List of prediction saved in DataSample.

Return type predictions (List[DataSample])

convert_to_datasample(*inputs: mmedit.utils.SampleList*, *data_samples: mmedit.utils.SampleList*) → mmedit.utils.SampleList

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

Forward dummy function for getting flops.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Results tensor with shape of (n, 3, h, w).

Return type torch.Tensor

class mmedit.models.base_models.**TwoStageInpaintor**(*data_preprocessor: Union[dict, mmengine.config.Config]*, *encdec: dict*, *disc: Optional[dict] = None*, *loss_gan: Optional[dict] = None*, *loss_gp: Optional[dict] = None*, *loss_disc_shift: Optional[dict] = None*, *loss_composed_percep: Optional[dict] = None*, *loss_out_percep: bool = False*, *loss_l1_hole: Optional[dict] = None*, *loss_l1_valid: Optional[dict] = None*, *loss_tv: Optional[dict] = None*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = None*, *stage1_loss_type: Optional[Sequence[str]] = ('loss_l1_hole',)*, *stage2_loss_type: Optional[Sequence[str]] = ('loss_l1_hole', 'loss_gan')*, *input_with_ones: bool = True*, *disc_input_with_mask: bool = False*)

Bases: mmedit.models.base_models.one_stage.OneStageInpaintor

Standard two-stage inpaintor with commonly used losses. A two-stage inpaintor contains two encoder-decoder style generators to inpaint masked regions. Currently, we support these loss types in each of two stage inpaintors:

['loss_gan', 'loss_l1_hole', 'loss_l1_valid', 'loss_composed_percep', 'loss_out_percep', 'loss_tv'] The *stage1_loss_type* and *stage2_loss_type* should be chosen from these loss types.

Parameters

- **data_preprocessor** (*dict*) – Config of data_preprocessor.
- **encdec** (*dict*) – Config for encoder-decoder style generator.
- **disc** (*dict*) – Config for discriminator.
- **loss_gan** (*dict*) – Config for adversarial loss.
- **loss_gp** (*dict*) – Config for gradient penalty loss.
- **loss_disc_shift** (*dict*) – Config for discriminator shift loss.
- **loss_composed_percep** (*dict*) – Config for perceptual and style loss with composed image as input.
- **loss_out_percep** (*dict*) – Config for perceptual and style loss with direct output as input.
- **loss_l1_hole** (*dict*) – Config for l1 loss in the hole.
- **loss_l1_valid** (*dict*) – Config for l1 loss in the valid region.
- **loss_tv** (*dict*) – Config for total variation loss.
- **train_cfg** (*dict*) – Configs for training scheduler. *disc_step* must be contained for indicates the discriminator updating steps in each training step.
- **test_cfg** (*dict*) – Configs for testing scheduler.
- **init_cfg** (*dict, optional*) – Initialization config dict.

- **stage1_loss_type** (*tuple[str]*) – Contains the loss names used in the first stage model. Default: ('loss_l1_hole').
- **stage2_loss_type** (*tuple[str]*) – Contains the loss names used in the second stage model. Default: ('loss_l1_hole', 'loss_gan').
- **input_with_ones** (*bool*) – Whether to concatenate an extra ones tensor in input. Default: True.
- **disc_input_with_mask** (*bool*) – Whether to add mask as input in discriminator. Default: False.

forward_tensor(*inputs: torch.Tensor*, *data_samples: mmedit.utils.SampleList*) → Tuple[torch.Tensor, torch.Tensor]

Forward function in tensor mode.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns Dict contains output results.

Return type dict

two_stage_loss(*stage1_data: dict*, *stage2_data: dict*, *gt: torch.Tensor*, *mask: torch.Tensor*, *masked_img: torch.Tensor*) → Tuple[dict, dict]

Calculate two-stage loss.

Parameters

- **stage1_data** (*dict*) – Contain stage1 results.
- **stage2_data** (*dict*) – Contain stage2 results..
- **gt** (*torch.Tensor*) – Ground-truth image.
- **mask** (*torch.Tensor*) – Mask image.
- **masked_img** (*torch.Tensor*) – Composition of mask image and ground-truth image.

Returns Dict contains the results computed within this function for visualization and dict contains the loss items computed in this function.

Return type tuple(dict)

calculate_loss_with_type(*loss_type: str*, *fake_res: torch.Tensor*, *fake_img: torch.Tensor*, *gt: torch.Tensor*, *mask: torch.Tensor*, *prefix: Optional[str] = 'stage1_'*) → dict

Calculate multiple types of losses.

Parameters

- **loss_type** (*str*) – Type of the loss.
- **fake_res** (*torch.Tensor*) – Direct results from model.
- **fake_img** (*torch.Tensor*) – Composited results from model.
- **gt** (*torch.Tensor*) – Ground-truth tensor.
- **mask** (*torch.Tensor*) – Mask tensor.
- **prefix** (*str, optional*) – Prefix for loss name. Defaults to 'stage1_'. # noqa

Returns Contain loss value with its name.

Return type dict

train_step(*data: List[dict]*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → dict

Train step function.

In this function, the inpaintor will finish the train step following the pipeline: 1. get fake res/image 2. optimize discriminator (if have) 3. optimize generator

If *self.train_cfg.disc_step > 1*, the train step will contain multiple iterations for optimizing discriminator with different input data and only one iteration for optimizing gerator after *disc_step* iterations for discriminator.

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

1.50 mmedit.models.losses

1.50.1 Package Contents

Classes

CLIPLoss	Clip loss. In styleclip, this loss is used to optimize the latent code
CharbonnierCompLoss	Charbonnier composition loss.
L1CompositionLoss	L1 composition loss.
MSECompositionLoss	MSE (L2) composition loss.
FaceIdLoss	Face similarity loss. Generally this loss is used to keep the id
LightCNNFeatureLoss	Feature loss of DICGAN, based on LightCNN.
DiscShiftLoss	Disc shift loss.
GANLoss	Define GAN loss.
GaussianBlur	A Gaussian filter which blurs a given tensor with a two-dimensional
GradientPenaltyLoss	Gradient penalty loss for wgan-gp.
GradientLoss	Gradient loss.
CLIPLossComps	Clip loss. In styleclip, this loss is used to optimize the latent code
DiscShiftLossComps	Disc Shift Loss.
FaceIdLossComps	Face similarity loss. Generally this loss is used to keep the id
GANLossComps	Define GAN loss.
GeneratorPathRegularizerComps	Generator Path Regularizer.
GradientPenaltyLossComps	Gradient Penalty for WGAN-GP.
R1GradientPenaltyComps	R1 gradient penalty for WGAN-GP.
PerceptualLoss	Perceptual loss with commonly used style loss.
PerceptualVGG	VGG network used in calculating perceptual loss.
TransferalPerceptualLoss	Transferal perceptual loss.
CharbonnierLoss	Charbonnier loss (one variant of Robust L1Loss, a differentiable variant
L1Loss	L1 (mean absolute error, MAE) loss.
MaskedTVLoss	Masked TV loss.
MSELoss	MSE (L2) loss.
PSNRLoss	PSNR Loss in "HINet: Half Instance Normalization Network for Image

Functions

disc_shift_loss(→ torch.Tensor)	Disc Shift loss.
gen_path_regularizer(→ Tuple[torch.Tensor])	Generator Path Regularization.
gradient_penalty_loss(→ torch.Tensor)	Calculate gradient penalty for wgan-gp.
r1_gradient_penalty_loss(→ torch.Tensor)	Calculate R1 gradient penalty for WGAN-GP.
mask_reduce_loss(→ torch.Tensor)	Apply element-wise weight and reduce loss.
reduce_loss(→ torch.Tensor)	Reduce loss as specified.
tv_loss(→ torch.Tensor)	L2 total variation loss, as in Mahendran et al.

class mmedit.models.losses.**CLIPLoss**(*loss_weight: float = 1.0*, *data_info: Optional[dict] = None*, *clip_model: dict = dict()*, *loss_name: str = 'loss_clip'*)

Bases: torch.nn.Module

Clip loss. In styleclip, this loss is used to optimize the latent code to generate image that match the text.

In this loss, we may need to provide `image`, `text`. Thus, an example of the `data_info` is:

```
data_info = dict(
    image='fake_imgs',
    text='descriptions')
```

Then, the module will automatically construct this mapping from the input data dictionary.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to `1.`.
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If `None`, this module will directly pass the input data to the loss function. Defaults to None.
- **clip_model** (*dict, optional*) – Kwargs for clip loss model. Defaults to dict().
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_clip'.

forward(*image: torch.Tensor*, *text: torch.Tensor*) → torch.Tensor

Forward function.

If `self.data_info` is not `None`, a dictionary containing all of the data and necessary modules should be passed into this function. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using keyword argument, please name it as *outputs_dict*.

If `self.data_info` is `None`, the input argument or key-word argument will be directly passed to loss function, `third_party_net_loss`.

class mmedit.models.losses.**CharbonnierCompLoss**(*loss_weight: float = 1.0*, *reduction: str = 'mean'*, *sample_wise: bool = False*, *eps: bool = 1e-12*)

Bases: `torch.nn.Module`

Charbonnier composition loss.

Parameters

- **loss_weight** (*float*) – Loss weight for L1 loss. Default: 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the output. Supported choices are 'none' | 'mean' | 'sum'. Default: 'mean'.
- **sample_wise** (*bool*) – Whether calculate the loss sample-wise. This argument only takes effect when *reduction* is 'mean' and *weight* (argument of *forward()*) is not None. It will first reduces loss with 'mean' per-sample, and then it means over all the samples. Default: False.
- **eps** (*float*) – A value used to control the curvature near zero. Default: 1e-12.

forward(*pred_alpha: torch.Tensor*, *fg: torch.Tensor*, *bg: torch.Tensor*, *ori_merged: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, ***kwargs*) → torch.Tensor

Parameters

- **pred_alpha** (*Tensor*) – of shape (N, 1, H, W). Predicted alpha matte.
- **fg** (*Tensor*) – of shape (N, 3, H, W). Tensor of foreground object.
- **bg** (*Tensor*) – of shape (N, 3, H, W). Tensor of background object.

- **ori_merged** (*Tensor*) – of shape (N, 3, H, W). Tensor of origin merged image before normalized by ImageNet mean and std.
- **weight** (*Tensor, optional*) – of shape (N, 1, H, W). It is an indicating matrix: weight[trimap == 128] = 1. Default: None.

class mmedit.models.losses.**L1CompositionLoss**(*loss_weight: float = 1.0*, *reduction: str = 'mean'*, *sample_wise: bool = False*)

Bases: torch.nn.Module

L1 composition loss.

Parameters

- **loss_weight** (*float*) – Loss weight for L1 loss. Default: 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the output. Supported choices are 'none' | 'mean' | 'sum'. Default: 'mean'.
- **sample_wise** (*bool*) – Whether calculate the loss sample-wise. This argument only takes effect when *reduction* is 'mean' and *weight* (argument of *forward()*) is not None. It will first reduces loss with 'mean' per-sample, and then it means over all the samples. Default: False.

forward(*pred_alpha: torch.Tensor*, *fg: torch.Tensor*, *bg: torch.Tensor*, *ori_merged: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, ***kwargs*) → torch.Tensor

Parameters

- **pred_alpha** (*Tensor*) – of shape (N, 1, H, W). Predicted alpha matte.
- **fg** (*Tensor*) – of shape (N, 3, H, W). Tensor of foreground object.
- **bg** (*Tensor*) – of shape (N, 3, H, W). Tensor of background object.
- **ori_merged** (*Tensor*) – of shape (N, 3, H, W). Tensor of origin merged image before normalized by ImageNet mean and std.
- **weight** (*Tensor, optional*) – of shape (N, 1, H, W). It is an indicating matrix: weight[trimap == 128] = 1. Default: None.

class mmedit.models.losses.**MSECompositionLoss**(*loss_weight: float = 1.0*, *reduction: str = 'mean'*, *sample_wise: bool = False*)

Bases: torch.nn.Module

MSE (L2) composition loss.

Parameters

- **loss_weight** (*float*) – Loss weight for MSE loss. Default: 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the output. Supported choices are 'none' | 'mean' | 'sum'. Default: 'mean'.
- **sample_wise** (*bool*) – Whether calculate the loss sample-wise. This argument only takes effect when *reduction* is 'mean' and *weight* (argument of *forward()*) is not None. It will first reduces loss with 'mean' per-sample, and then it means over all the samples. Default: False.

forward(*pred_alpha: torch.Tensor*, *fg: torch.Tensor*, *bg: torch.Tensor*, *ori_merged: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, ***kwargs*) → torch.Tensor

Parameters

- **pred_alpha** (*Tensor*) – of shape (N, 1, H, W). Predicted alpha matte.
- **fg** (*Tensor*) – of shape (N, 3, H, W). Tensor of foreground object.

- **bg** (*Tensor*) – of shape (N, 3, H, W). Tensor of background object.
- **ori_merged** (*Tensor*) – of shape (N, 3, H, W). Tensor of origin merged image before normalized by ImageNet mean and std.
- **weight** (*Tensor, optional*) – of shape (N, 1, H, W). It is an indicating matrix: weight[trimap == 128] = 1. Default: None.

class mmedit.models.losses.**FaceIdLoss**(*loss_weight: float = 1.0*, *data_info: Optional[dict] = None*, *facenet: dict = dict(type='ArcFace', ir_se50_weights=None)*, *loss_name: str = 'loss_id'*)

Bases: torch.nn.Module

Face similarity loss. Generally this loss is used to keep the id consistency of the input face image and output face image.

In this loss, we may need to provide `gt`, `pred` and `x`. Thus, an example of the `data_info` is:

```
data_info = dict(
    gt='real_imgs',
    pred='fake_imgs')
```

Then, the module will automatically construct this mapping from the input data dictionary.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to `1.`.
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If `None`, this module will directly pass the input data to the loss function. Defaults to None.
- **facenet** (*dict, optional*) – Config dict for facenet. Defaults to dict(type='ArcFace', ir_se50_weights=None, device='cuda').
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_id'.

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

Forward function.

class mmedit.models.losses.**LightCNNFeatureLoss**(*pretrained: str*, *loss_weight: float = 1.0*, *criterion: str = 'l1'*)

Bases: torch.nn.Module

Feature loss of DICGAN, based on LightCNN.

Parameters

- **pretrained** (*str*) – Path for pretrained weights.
- **loss_weight** (*float*) – Loss weight. Default: 1.0.
- **criterion** (*str*) – Criterion type. Options are 'l1' and 'mse'. Default: 'l1'.

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

Forward function.

Parameters

- **pred** (*Tensor*) – Predicted tensor.
- **gt** (*Tensor*) – GT tensor.

Returns Forward results.

Return type Tensor

class mmedit.models.losses.**DiscShiftLoss**(*loss_weight: float = 0.1*)

Bases: torch.nn.Module

Disc shift loss.

Parameters **loss_weight** (*float, optional*) – Loss weight. Defaults to 1.0.

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

Forward function.

Parameters **x** (*Tensor*) – Tensor with shape (n, c, h, w)

Returns Loss.

Return type Tensor

class mmedit.models.losses.**GANLoss**(*gan_type: str*, *real_label_val: float = 1.0*, *fake_label_val: float = 0.0*, *loss_weight: float = 1.0*)

Bases: torch.nn.Module

Define GAN loss.

Parameters

- **gan_type** (*str*) – Support 'vanilla', 'lsgan', 'wgan', 'hinge'.
- **real_label_val** (*float*) – The value for real label. Default: 1.0.
- **fake_label_val** (*float*) – The value for fake label. Default: 0.0.
- **loss_weight** (*float*) – Loss weight. Default: 1.0. Note that loss_weight is only for generators; and it is always 1.0 for discriminators.

_wgan_loss(*input: torch.Tensor*, *target: bool*) → torch.Tensor

wgan loss.

Parameters

- **input** (*Tensor*) – Input tensor.
- **target** (*bool*) – Target label.

Returns wgan loss.

Return type Tensor

get_target_label(*input: torch.Tensor*, *target_is_real: bool*) → Union[bool, torch.Tensor]

Get target label.

Parameters

- **input** (*Tensor*) – Input tensor.
- **target_is_real** (*bool*) – Whether the target is real or fake.

Returns

Target tensor. Return bool for wgan, otherwise, return Tensor.

Return type (bool | Tensor)

forward(*input: torch.Tensor*, *target_is_real: bool*, *is_disc: bool = False*, *mask: Optional[torch.Tensor] = None*) → torch.Tensor

Parameters

- **input** (*Tensor*) – The input for the loss module, i.e., the network prediction.
- **target_is_real** (*bool*) – Whether the target is real or fake.
- **is_disc** (*bool*) – Whether the loss for discriminators or not. Default: False.
- **mask** (*Tensor*) – The mask tensor. Default: None.

Returns GAN loss value.

Return type Tensor

class mmedit.models.losses.**GaussianBlur**(*kernel_size: Tuple[int, int] = (71, 71)*, *sigma: Tuple[float, float] = (10.0, 10.0)*)

Bases: torch.nn.Module

A Gaussian filter which blurs a given tensor with a two-dimensional gaussian kernel by convolving it along each channel. Batch operation is supported.

This function is modified from kornia.filters.gaussian: *<https://kornia.readthedocs.io/en/latest/_modules/kornia/filters/gaussian.ht*

Parameters

- **kernel_size** (*tuple[int]*) – The size of the kernel. Default: (71, 71).
- **sigma** (*tuple[float]*) – The standard deviation of the kernel.
- **Default** (*10.0, 10.0*) –

Returns The Gaussian-blurred tensor.

Return type Tensor

Shape:

- input: Tensor with shape of (n, c, h, w)
- output: Tensor with shape of (n, c, h, w)

static compute_zero_padding(*kernel_size: Tuple[int, int]*) → tuple

Compute zero padding tuple.

Parameters **kernel_size** (*tuple[int]*) – The size of the kernel.

Returns Padding of height and weight.

Return type tuple

get_2d_gaussian_kernel(*kernel_size: Tuple[int, int]*, *sigma: Tuple[float, float]*) → torch.Tensor

Get the two-dimensional Gaussian filter matrix coefficients.

Parameters

- **kernel_size** (*tuple[int]*) – Kernel filter size in the x and y direction. The kernel sizes should be odd and positive.
- **sigma** (*tuple[int]*) – Gaussian standard deviation in the x and y direction.

Returns

A 2D torch tensor with gaussian filter matrix coefficients.

Return type kernel_2d (Tensor)

get_1d_gaussian_kernel(*kernel_size: int*, *sigma: float*) → torch.Tensor

Get the Gaussian filter coefficients in one dimension (x or y direction).

Parameters

- **kernel_size** (*int*) – Kernel filter size in x or y direction. Should be odd and positive.
- **sigma** (*float*) – Gaussian standard deviation in x or y direction.

Returns

A 1D torch tensor with gaussian filter coefficients in x or y direction.

Return type kernel_1d (Tensor)

gaussian(*kernel_size: int*, *sigma: float*) → torch.Tensor

Gaussian function.

Parameters

- **kernel_size** (*int*) – Kernel filter size in x or y direction. Should be odd and positive.
- **sigma** (*float*) – Gaussian standard deviation in x or y direction.

Returns

A 1D torch tensor with gaussian filter coefficients in x or y direction.

Return type Tensor

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

Forward function.

Parameters **x** (*Tensor*) – Tensor with shape (n, c, h, w)

Returns The Gaussian-blurred tensor.

Return type Tensor

class mmedit.models.losses.**GradientPenaltyLoss**(*loss_weight: float = 1.0*)

Bases: torch.nn.Module

Gradient penalty loss for wgan-gp.

Parameters **loss_weight** (*float*) – Loss weight. Default: 1.0.

forward(*discriminator: torch.nn.Module*, *real_data: torch.Tensor*, *fake_data: torch.Tensor*, *mask: Optional[torch.Tensor] = None*) → torch.Tensor

Forward function.

Parameters

- **discriminator** (*nn.Module*) – Network for the discriminator.
- **real_data** (*Tensor*) – Real input data.
- **fake_data** (*Tensor*) – Fake input data.
- **mask** (*Tensor*) – Masks for inpainting. Default: None.

Returns Loss.

Return type Tensor

mmedit.models.losses.**disc_shift_loss**(*pred: torch.Tensor*) → torch.Tensor

Disc Shift loss.

This loss is proposed in PGGAN as an auxiliary loss for discriminator.

Parameters **pred** (*Tensor*) – Input tensor.

Returns loss tensor.

Return type torch.Tensor

mmedit.models.losses.**gen_path_regularizer**(*generator: torch.nn.Module*, *num_batches: int*, *mean_path_length: torch.Tensor*, *pl_batch_shrink: int = 1*, *decay: float = 0.01*, *weight: float = 1.0*, *pl_batch_size: Optional[int] = None*, *sync_mean_buffer: bool = False*, *loss_scaler: Optional[torch.cuda.amp.grad_scaler.GradScaler] = None*, *use_apex_amp: bool = False*) → Tuple[torch.Tensor]

Generator Path Regularization.

Path regularization is proposed in StyelGAN2, which can help the improve the continuity of the latent space. More details can be found in: Analyzing and Improving the Image Quality of StyleGAN, CVPR2020.

Parameters

- **generator** (*nn.Module*) – The generator module. Note that this loss requires that the generator contains `return_latents` interface, with which we can get the latent code of the current sample.
- **num_batches** (*int*) – The number of samples used in calculating this loss.
- **mean_path_length** (*Tensor*) – The mean path length, calculated by moving average.
- **pl_batch_shrink** (*int, optional*) – The factor of shrinking the batch size for saving GPU memory. Defaults to 1.
- **decay** (*float, optional*) – Decay for moving average of mean path length. Defaults to 0.01.
- **weight** (*float, optional*) – Weight of this loss item. Defaults to `1.`.
- **pl_batch_size** (*int | None, optional*) – The batch size in calculating generator path. Once this argument is set, the `num_batches` will be overridden with this argument and won't be affectted by `pl_batch_shrink`. Defaults to None.
- **sync_mean_buffer** (*bool, optional*) – Whether to sync mean path length across all of GPUs. Defaults to False.

Returns The penalty loss, detached mean path tensor, and current path length.

Return type tuple[Tensor]

mmedit.models.losses.**gradient_penalty_loss**(*discriminator: torch.nn.Module*, *real_data: torch.Tensor*, *fake_data: torch.Tensor*, *mask: Optional[torch.Tensor] = None*, *norm_mode: str = 'pixel'*) → torch.Tensor

Calculate gradient penalty for wgan-gp.

Parameters

- **discriminator** (*nn.Module*) – Network for the discriminator.
- **real_data** (*Tensor*) – Real input data.
- **fake_data** (*Tensor*) – Fake input data.

- **mask** (*Tensor*) – Masks for inpainting. Default: None.

Returns A tensor for gradient penalty.

Return type Tensor

mmedit.models.losses.**r1_gradient_penalty_loss**(*discriminator: torch.nn.Module*, *real_data: torch.Tensor*, *mask: Optional[torch.Tensor] = None*, *norm_mode: str = 'pixel'*, *loss_scaler: Optional[torch.cuda.amp.grad_scaler.GradScaler] = None*, *use_apex_amp: bool = False*) → torch.Tensor

Calculate R1 gradient penalty for WGAN-GP.

R1 regularizer comes from: "Which Training Methods for GANs do actually Converge?" ICML'2018

Different from original gradient penalty, this regularizer only penalized gradient w.r.t. real data.

Parameters

- **discriminator** (*nn.Module*) – Network for the discriminator.
- **real_data** (*Tensor*) – Real input data.
- **mask** (*Tensor*) – Masks for inpainting. Default: None.
- **norm_mode** (*str*) – This argument decides along which dimension the norm of the gradients will be calculated. Currently, we support ["pixel" , "HWC"]. Defaults to "pixel".

Returns A tensor for gradient penalty.

Return type Tensor

class mmedit.models.losses.**GradientLoss**(*loss_weight: float = 1.0*, *reduction: str = 'mean'*)

Bases: torch.nn.Module

Gradient loss.

Parameters

- **loss_weight** (*float*) – Loss weight for L1 loss. Default: 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the output. Supported choices are 'none' | 'mean' | 'sum'. Default: 'mean'.

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

Parameters

- **pred** (*Tensor*) – of shape (N, C, H, W). Predicted tensor.
- **target** (*Tensor*) – of shape (N, C, H, W). Ground truth tensor.
- **weight** (*Tensor, optional*) – of shape (N, C, H, W). Element-wise weights. Default: None.

class mmedit.models.losses.**CLIPLossComps**(*loss_weight: float = 1.0*, *data_info: Optional[dict] = None*, *clip_model: dict = dict()*, *loss_name: str = 'loss_clip'*)

Bases: torch.nn.Module

Clip loss. In styleclip, this loss is used to optimize the latent code to generate image that match the text.

In this loss, we may need to provide image, text. Thus, an example of the data_info is:

```
data_info = dict(
    image='fake_imgs',
    text='descriptions')
```

Then, the module will automatically construct this mapping from the input data dictionary.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to 1..
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If None, this module will directly pass the input data to the loss function. Defaults to None.
- **clip_model** (*dict, optional*) – Kwargs for clip loss model. Defaults to dict().
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_clip'.

forward(**args*, ***kwargs*) → torch.Tensor

Forward function.

If self.data_info is not None, a dictionary containing all of the data and necessary modules should be passed into this function. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using keyword argument, please name it as *outputs_dict*.

If self.data_info is None, the input argument or key-word argument will be directly passed to loss function, third_party_net_loss.

static loss_name() → str

Loss Name.

This function must be implemented and will return the name of this loss function. This name will be used to combine different loss items by simple sum operation. In addition, if you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name.

Returns The name of this loss item.

Return type str

class mmedit.models.losses.**DiscShiftLossComps**(*loss_weight: float = 1.0*, *data_info: Optional[dict] = None*, *loss_name: str = 'loss_disc_shift'*)

Bases: torch.nn.Module

Disc Shift Loss.

This loss is proposed in PGGAN as an auxiliary loss for discriminator.

Note for the design of ``data_info``: In MMEditing, almost all of loss modules contain the argument data_info, which can be used for constructing the link between the input items (needed in loss calculation) and the data from the generative model. For example, in the training of GAN model, we will collect all of important data/modules into a dictionary:

Listing 1: Code from StaticUnconditionalGAN, train_step

```
data_dict_ = dict(
    gen=self.generator,
    disc=self.discriminator,
    disc_pred_fake=disc_pred_fake,
    disc_pred_real=disc_pred_real,
```

(continues on next page)

(continued from previous page)

```
    fake_imgs=fake_imgs,
    real_imgs=real_imgs,
    iteration=curr_iter,
    batch_size=batch_size)
```

But in this loss, we will need to provide `pred` as input. Thus, an example of the `data_info` is:

```
data_info = dict(
    pred='disc_pred_fake')
```

Then, the module will automatically construct this mapping from the input data dictionary.

In addition, in general, `disc_shift_loss` will be applied over real and fake data. In this case, users just need to add this loss module twice, but with different `data_info`. Our model will automatically add these two items.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to `1.`.
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If `None`, this module will directly pass the input data to the loss function. Defaults to None.
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_disc_shift'.

forward(**args*, ***kwargs*) → torch.Tensor

Forward function.

If `self.data_info` is not `None`, a dictionary containing all of the data and necessary modules should be passed into this function. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using keyword argument, please name it as *outputs_dict*.

If `self.data_info` is `None`, the input argument or key-word argument will be directly passed to loss function, `disc_shift_loss`.

loss_name() → str

Loss Name.

This function must be implemented and will return the name of this loss function. This name will be used to combine different loss items by simple sum operation. In addition, if you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name.

Returns The name of this loss item.

Return type str

class mmedit.models.losses.**FaceIdLossComps**(*loss_weight: float = 1.0*, *data_info: Optional[dict] = None*, *facenet: dict = dict(type='ArcFace', ir_se50_weights=None)*, *loss_name: str = 'loss_id'*)

Bases: `torch.nn.Module`

Face similarity loss. Generally this loss is used to keep the id consistency of the input face image and output face image.

In this loss, we may need to provide `gt`, `pred` and `x`. Thus, an example of the `data_info` is:

```
data_info = dict(
    gt='real_imgs',
    pred='fake_imgs')
```

Then, the module will automatically construct this mapping from the input data dictionary.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to 1..
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If None, this module will directly pass the input data to the loss function. Defaults to None.
- **facenet** (*dict, optional*) – Config dict for facenet. Defaults to dict(type='ArcFace', ir_se50_weights=None).
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_id'.

forward(**args*, ***kwargs*) → torch.Tensor

Forward function.

If `self.data_info` is not `None`, a dictionary containing all of the data and necessary modules should be passed into this function. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using keyword argument, please name it as *outputs_dict*.

If `self.data_info` is `None`, the input argument or key-word argument will be directly passed to loss function, `third_party_net_loss`.

loss_name() → str

Loss Name.

This function must be implemented and will return the name of this loss function. This name will be used to combine different loss items by simple sum operation. In addition, if you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name.

Returns The name of this loss item.

Return type str

class mmedit.models.losses.**GANLossComps**(*gan_type: str*, *real_label_val: float = 1.0*, *fake_label_val: float = 0.0*, *loss_weight: float = 1.0*)

Bases: `torch.nn.Module`

Define GAN loss.

Parameters

- **gan_type** (*str*) – Support 'vanilla', 'lsgan', 'wgan', 'hinge', 'wgan-logistic-ns'.
- **real_label_val** (*float*) – The value for real label. Default: 1.0.
- **fake_label_val** (*float*) – The value for fake label. Default: 0.0.
- **loss_weight** (*float*) – Loss weight. Default: 1.0. Note that loss_weight is only for generators; and it is always 1.0 for discriminators.

_wgan_loss(*input: torch.Tensor*, *target: bool*) → torch.Tensor

wgan loss.

Parameters

- **input** (*Tensor*) – Input tensor.
- **target** (*bool*) – Target label.

Returns wgan loss.

Return type Tensor

_wgan_logistic_ns_loss(*input: torch.Tensor*, *target: bool*) → torch.Tensor

WGAN loss in logistically non-saturating mode.

This loss is widely used in StyleGANv2.

Parameters

- **input** (*Tensor*) – Input tensor.
- **target** (*bool*) – Target label.

Returns wgan loss.

Return type Tensor

get_target_label(*input: torch.Tensor*, *target_is_real: bool*) → Union[bool, torch.Tensor]

Get target label.

Parameters

- **input** (*Tensor*) – Input tensor.
- **target_is_real** (*bool*) – Whether the target is real or fake.

Returns Target tensor. Return bool for wgan, otherwise, return Tensor.

Return type (bool | Tensor)

forward(*input: torch.Tensor*, *target_is_real: bool*, *is_disc: bool = False*) → torch.Tensor

Parameters

- **input** (*Tensor*) – The input for the loss module, i.e., the network prediction.
- **target_is_real** (*bool*) – Whether the targe is real or fake.
- **is_disc** (*bool*) – Whether the loss for discriminators or not. Default: False.

Returns GAN loss value.

Return type Tensor

class mmedit.models.losses.**GeneratorPathRegularizerComps**(*loss_weight: float = 1.0*, *pl_batch_shrink: int = 1*, *decay: float = 0.01*, *pl_batch_size: Optional[int] = None*, *sync_mean_buffer: bool = False*, *interval: int = 1*, *data_info: Optional[dict] = None*, *use_apex_amp: bool = False*, *loss_name: str = 'loss_path_regular'*)

Bases: torch.nn.Module

Generator Path Regularizer.

Path regularization is proposed in StyelGAN2, which can help the improve the continuity of the latent space. More details can be found in: Analyzing and Improving the Image Quality of StyleGAN, CVPR2020.

Users can achieve lazy regularization by setting `interval` arguments here.

Note for the design of ``data_info``: In `MMEditing`, almost all of loss modules contain the argument `data_info`, which can be used for constructing the link between the input items (needed in loss calculation) and the data from the generative model. For example, in the training of GAN model, we will collect all of important data/modules into a dictionary:

Listing 2: Code from StaticUnconditionalGAN, train_step

```
data_dict_ = dict(
    gen=self.generator,
    disc=self.discriminator,
    fake_imgs=fake_imgs,
    disc_pred_fake_g=disc_pred_fake_g,
    iteration=curr_iter,
    batch_size=batch_size)
```

But in this loss, we will need to provide `generator` and `num_batches` as input. Thus an example of the `data_info` is:

```
data_info = dict(
    generator='gen',
    num_batches='batch_size')
```

Then, the module will automatically construct this mapping from the input data dictionary.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to `1.`.
- **pl_batch_shrink** (*int, optional*) – The factor of shrinking the batch size for saving GPU memory. Defaults to 1.
- **decay** (*float, optional*) – Decay for moving average of mean path length. Defaults to 0.01.
- **pl_batch_size** (*int | None, optional*) – The batch size in calculating generator path. Once this argument is set, the `num_batches` will be overridden with this argument and won't be affectted by `pl_batch_shrink`. Defaults to None.
- **sync_mean_buffer** (*bool, optional*) – Whether to sync mean path length across all of GPUs. Defaults to False.
- **interval** (*int, optional*) – The interval of calculating this loss. This argument is used to support lazy regularization. Defaults to 1.
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If `None`, this module will directly pass the input data to the loss function. Defaults to None.
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_path_regular'.

forward(**args*, ***kwargs*) → torch.Tensor

Forward function.

If `self.data_info` is not `None`, a dictionary containing all of the data and necessary modules should be passed into this function. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using keyword argument, please name it as *outputs_dict*.

If `self.data_info` is `None`, the input argument or key-word argument will be directly passed to loss function, `gen_path_regularizer`.

loss_name() → str

Loss Name.

This function must be implemented and will return the name of this loss function. This name will be used to combine different loss items by simple sum operation. In addition, if you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name.

Returns The name of this loss item.

Return type str

class mmedit.models.losses.**GradientPenaltyLossComps**(*loss_weight: float = 1.0*, *norm_mode: str = 'pixel'*, *data_info: Optional[dict] = None*, *loss_name: str = 'loss_gp'*)

Bases: `torch.nn.Module`

Gradient Penalty for WGAN-GP.

In the detailed implementation, there are two streams where one uses the pixel-wise gradient norm, but the other adopts normalization along instance (HWC) dimensions. Thus, `norm_mode` are offered to define which mode you want.

Note for the design of ``data_info``: In `MMEditing`, almost all of loss modules contain the argument `data_info`, which can be used for constructing the link between the input items (needed in loss calculation) and the data from the generative model. For example, in the training of GAN model, we will collect all of important data/modules into a dictionary:

Listing 3: Code from StaticUnconditionalGAN, train_step

```
data_dict_ = dict(
    gen=self.generator,
    disc=self.discriminator,
    disc_pred_fake=disc_pred_fake,
    disc_pred_real=disc_pred_real,
    fake_imgs=fake_imgs,
    real_imgs=real_imgs,
    iteration=curr_iter,
    batch_size=batch_size)
```

But in this loss, we will need to provide `discriminator`, `real_data`, and `fake_data` as input. Thus, an example of the `data_info` is:

```
data_info = dict(
    discriminator='disc',
    real_data='real_imgs',
    fake_data='fake_imgs')
```

Then, the module will automatically construct this mapping from the input data dictionary.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to `1.`.
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If `None`, this module will directly pass the input data to the loss function. Defaults to None.

- **norm_mode** (*str*) – This argument decides along which dimension the norm of the gradients will be calculated. Currently, we support ["pixel" , "HWC"]. Defaults to "pixel".
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_gp'.

forward(**args*, ***kwargs*) → torch.Tensor

Forward function.

If `self.data_info` is not `None`, a dictionary containing all of the data and necessary modules should be passed into this function. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using keyword argument, please name it as *outputs_dict*.

If `self.data_info` is `None`, the input argument or key-word argument will be directly passed to loss function, `gradient_penalty_loss`.

loss_name() → str

Loss Name.

This function must be implemented and will return the name of this loss function. This name will be used to combine different loss items by simple sum operation. In addition, if you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name.

Returns The name of this loss item.

Return type str

class mmedit.models.losses.**R1GradientPenaltyComps**(*loss_weight: float = 1.0*, *norm_mode: str = 'pixel'*, *interval: int = 1*, *data_info: Optional[dict] = None*, *use_apex_amp: bool = False*, *loss_name: str = 'loss_r1_gp'*)

Bases: `torch.nn.Module`

R1 gradient penalty for WGAN-GP.

R1 regularizer comes from: "Which Training Methods for GANs do actually Converge?" ICML'2018

Different from original gradient penalty, this regularizer only penalized gradient w.r.t. real data.

Note for the design of ``data_info``: In `MMEditing`, almost all of loss modules contain the argument `data_info`, which can be used for constructing the link between the input items (needed in loss calculation) and the data from the generative model. For example, in the training of GAN model, we will collect all of important data/modules into a dictionary:

Listing 4: Code from StaticUnconditionalGAN, train_step

```
data_dict_ = dict(
    gen=self.generator,
    disc=self.discriminator,
    disc_pred_fake=disc_pred_fake,
    disc_pred_real=disc_pred_real,
    fake_imgs=fake_imgs,
    real_imgs=real_imgs,
    iteration=curr_iter,
    batch_size=batch_size)
```

But in this loss, we will need to provide `discriminator` and `real_data` as input. Thus, an example of the `data_info` is:

```
data_info = dict(
    discriminator='disc',
    real_data='real_imgs')
```

Then, the module will automatically construct this mapping from the input data dictionary.

Parameters

- **loss_weight** (*float, optional*) – Weight of this loss item. Defaults to 1..
- **data_info** (*dict, optional*) – Dictionary contains the mapping between loss input args and data dictionary. If None, this module will directly pass the input data to the loss function. Defaults to None.
- **norm_mode** (*str*) – This argument decides along which dimension the norm of the gradients will be calculated. Currently, we support ["pixel" , "HWC"]. Defaults to "pixel".
- **interval** (*int, optional*) – The interval of calculating this loss. Defaults to 1.
- **loss_name** (*str, optional*) – Name of the loss item. If you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name. Defaults to 'loss_r1_gp'.

forward(**args*, ***kwargs*) → torch.Tensor

Forward function.

If `self.data_info` is not `None`, a dictionary containing all of the data and necessary modules should be passed into this function. If this dictionary is given as a non-keyword argument, it should be offered as the first argument. If you are using keyword argument, please name it as *outputs_dict*.

If `self.data_info` is `None`, the input argument or key-word argument will be directly passed to loss function, `r1_gradient_penalty_loss`.

loss_name() → str

Loss Name.

This function must be implemented and will return the name of this loss function. This name will be used to combine different loss items by simple sum operation. In addition, if you want this loss item to be included into the backward graph, *loss_* must be the prefix of the name.

Returns The name of this loss item.

Return type str

mmedit.models.losses.**mask_reduce_loss**(*loss: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, *reduction: str = 'mean'*, *sample_wise: bool = False*) → torch.Tensor

Apply element-wise weight and reduce loss.

Parameters

- **loss** (*Tensor*) – Element-wise loss.
- **weight** (*Tensor*) – Element-wise weights. Default: None.
- **reduction** (*str*) – Same as built-in losses of PyTorch. Options are "none", "mean" and "sum". Default: 'mean'.
- **sample_wise** (*bool*) – Whether calculate the loss sample-wise. This argument only takes effect when *reduction* is 'mean' and *weight* (argument of *forward()*) is not None. It will first reduces loss with 'mean' per-sample, and then it means over all the samples. Default: False.

Returns Processed loss values.

Return type Tensor

mmedit.models.losses.**reduce_loss**(*loss: torch.Tensor*, *reduction: str*) → torch.Tensor

Reduce loss as specified.

Parameters

- **loss** (*Tensor*) – Elementwise loss tensor.
- **reduction** (*str*) – Options are “none”, “mean” and “sum”.

Returns Reduced loss tensor.

Return type Tensor

class mmedit.models.losses.**PerceptualLoss**(*layer_weights: dict*, *layer_weights_style: Optional[dict] = None*, *vgg_type: str = 'vgg19'*, *use_input_norm: bool = True*, *perceptual_weight: float = 1.0*, *style_weight: float = 1.0*, *norm_img: bool = True*, *pretrained: str = 'torchvision://vgg19'*, *criterion: str = 'l1'*)

Bases: torch.nn.Module

Perceptual loss with commonly used style loss.

Parameters

- **layers_weights** (*dict*) – The weight for each layer of vgg feature for perceptual loss. Here is an example: {‘4’: 1., ‘9’: 1., ‘18’: 1.}, which means the 5th, 10th and 18th feature layer will be extracted with weight 1.0 in calculating losses.
- **layers_weights_style** (*dict*) – The weight for each layer of vgg feature for style loss. If set to ‘None’, the weights are set equal to the weights for perceptual loss. Default: None.
- **vgg_type** (*str*) – The type of vgg network used as feature extractor. Default: ‘vgg19’.
- **use_input_norm** (*bool*) – If True, normalize the input image in vgg. Default: True.
- **perceptual_weight** (*float*) – If *perceptual_weight > 0*, the perceptual loss will be calculated and the loss will multiplied by the weight. Default: 1.0.
- **style_weight** (*float*) – If *style_weight > 0*, the style loss will be calculated and the loss will multiplied by the weight. Default: 1.0.
- **norm_img** (*bool*) – If True, the image will be normed to [0, 1]. Note that this is different from the *use_input_norm* which norm the input in in forward function of vgg according to the statistics of dataset. Importantly, the input image must be in range [-1, 1].
- **pretrained** (*str*) – Path for pretrained weights. Default: ‘torchvision://vgg19’.
- **criterion** (*str*) – Criterion type. Options are ‘l1’ and ‘mse’. Default: ‘l1’.

forward(*x: torch.Tensor*, *gt: torch.Tensor*) → Tuple[torch.Tensor]

Forward function.

Parameters

- **x** (*Tensor*) – Input tensor with shape (n, c, h, w).
- **gt** (*Tensor*) – Ground-truth tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

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

Calculate Gram matrix.

Parameters **x** (*torch.Tensor*) – Tensor with shape of (n, c, h, w).

Returns Gram matrix.

Return type torch.Tensor

class mmedit.models.losses.**PerceptualVGG**(*layer_name_list: List[str]*, *vgg_type: str = 'vgg19'*, *use_input_norm: bool = True*, *pretrained: str = 'torchvision://vgg19'*)

Bases: torch.nn.Module

VGG network used in calculating perceptual loss.

In this implementation, we allow users to choose whether use normalization in the input feature and the type of vgg network. Note that the pretrained path must fit the vgg type.

Parameters

- **layer_name_list** (*list[str]*) – According to the name in this list, forward function will return the corresponding features. This list contains the name each layer in *vgg.feature*. An example of this list is ['4', '10'].
- **vgg_type** (*str*) – Set the type of vgg network. Default: 'vgg19'.
- **use_input_norm** (*bool*) – If True, normalize the input image. Importantly, the input feature must in the range [0, 1]. Default: True.
- **pretrained** (*str*) – Path for pretrained weights. Default: 'torchvision://vgg19'

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

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

init_weights(*model: torch.nn.Module*, *pretrained: str*) → None

Init weights.

Parameters

- **model** (*nn.Module*) – Models to be inited.
- **pretrained** (*str*) – Path for pretrained weights.

class mmedit.models.losses.**TransferalPerceptualLoss**(*loss_weight: float = 1.0*, *use_attention: bool = True*, *criterion: str = 'mse'*)

Bases: torch.nn.Module

Transferal perceptual loss.

Parameters

- **loss_weight** (*float*) – Loss weight. Default: 1.0.
- **use_attention** (*bool*) – If True, use soft-attention tensor. Default: True
- **criterion** (*str*) – Criterion type. Options are 'l1' and 'mse'. Default: 'mse'.

forward(*maps: Tuple[torch.Tensor]*, *soft_attention: torch.Tensor*, *textures: Tuple[torch.Tensor]*) → torch.Tensor

Forward function.

Parameters

- **maps** (*Tuple[Tensor]*) – Input tensors.
- **soft_attention** (*Tensor*) – Soft-attention tensor.
- **textures** (*Tuple[Tensor]*) – Ground-truth tensors.

Returns Forward results.

Return type Tensor

class mmedit.models.losses.**CharbonnierLoss**(*loss_weight: float = 1.0*, *reduction: str = 'mean'*, *sample_wise: bool = False*, *eps: float = 1e-12*)

Bases: torch.nn.Module

Charbonnier loss (one variant of Robust L1Loss, a differentiable variant of L1Loss).

Described in "Deep Laplacian Pyramid Networks for Fast and Accurate Super-Resolution".

Parameters

- **loss_weight** (*float*) – Loss weight for L1 loss. Default: 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the output. Supported choices are 'none' | 'mean' | 'sum'. Default: 'mean'.
- **sample_wise** (*bool*) – Whether calculate the loss sample-wise. This argument only takes effect when *reduction* is 'mean' and *weight* (argument of *forward()*) is not None. It will first reduces loss with 'mean' per-sample, and then it means over all the samples. Default: False.
- **eps** (*float*) – A value used to control the curvature near zero. Default: 1e-12.

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

Forward Function.

Parameters

- **pred** (*Tensor*) – of shape (N, C, H, W). Predicted tensor.
- **target** (*Tensor*) – of shape (N, C, H, W). Ground truth tensor.
- **weight** (*Tensor, optional*) – of shape (N, C, H, W). Element-wise weights. Default: None.

class mmedit.models.losses.**L1Loss**(*loss_weight: float = 1.0*, *reduction: str = 'mean'*, *sample_wise: bool = False*)

Bases: torch.nn.Module

L1 (mean absolute error, MAE) loss.

Parameters

- **loss_weight** (*float*) – Loss weight for L1 loss. Default: 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the output. Supported choices are 'none' | 'mean' | 'sum'. Default: 'mean'.

- **sample_wise** (*bool*) – Whether calculate the loss sample-wise. This argument only takes effect when *reduction* is 'mean' and *weight* (argument of *forward()*) is not None. It will first reduce loss with 'mean' per-sample, and then it means over all the samples. Default: False.

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

Forward Function.

Parameters

- **pred** (*Tensor*) – of shape (N, C, H, W). Predicted tensor.
- **target** (*Tensor*) – of shape (N, C, H, W). Ground truth tensor.
- **weight** (*Tensor, optional*) – of shape (N, C, H, W). Element-wise weights. Default: None.

class mmedit.models.losses.**MaskedTVLoss**(*loss_weight: float = 1.0*)

Bases: *L1Loss*

Masked TV loss.

Parameters **loss_weight** (*float, optional*) – Loss weight. Defaults to 1.0.

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

Forward function.

Parameters

- **pred** (*torch.Tensor*) – Tensor with shape of (n, c, h, w).
- **mask** (*torch.Tensor, optional*) – Tensor with shape of (n, 1, h, w). Defaults to None.

Returns [description]

Return type [type]

class mmedit.models.losses.**MSELoss**(*loss_weight: float = 1.0*, *reduction: str = 'mean'*, *sample_wise: bool = False*)

Bases: torch.nn.Module

MSE (L2) loss.

Parameters

- **loss_weight** (*float*) – Loss weight for MSE loss. Default: 1.0.
- **reduction** (*str*) – Specifies the reduction to apply to the output. Supported choices are 'none' | 'mean' | 'sum'. Default: 'mean'.
- **sample_wise** (*bool*) – Whether calculate the loss sample-wise. This argument only takes effect when *reduction* is 'mean' and *weight* (argument of *forward()*) is not None. It will first reduces loss with 'mean' per-sample, and then it means over all the samples. Default: False.

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

Forward Function.

Parameters

- **pred** (*Tensor*) – of shape (N, C, H, W). Predicted tensor.
- **target** (*Tensor*) – of shape (N, C, H, W). Ground truth tensor.

- **weight** (*Tensor, optional*) – of shape (N, C, H, W). Element-wise weights. Default: None.

class mmedit.models.losses.**PSNRLoss**(*loss_weight: float = 1.0*, *toY: bool = False*)

Bases: torch.nn.Module

PSNR Loss in "HINet: Half Instance Normalization Network for Image Restoration".

Parameters

- **loss_weight** (*float, optional*) – Loss weight. Defaults to 1.0.
- **reduction** – reduction for PSNR. Can only be mean here.
- **toY** – change to calculate the PSNR of Y channel in YCbCr format

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

mmedit.models.losses.**tv_loss**(*input: torch.Tensor*) → torch.Tensor

L2 total variation loss, as in Mahendran et al.

1.51 mmedit.models.data_preprocessors

1.51.1 Package Contents

Classes

EditDataPreprocessor	Basic data pre-processor used for collating and copying data to the
GenDataPreprocessor	Image pre-processor for generative models. This class provide
MattorPreprocessor	DataPreprocessor for matting models.

Functions

split_batch(batch_tensor, padded_sizes)	reverse operation of stack_batch.
stack_batch(tensor_list[, pad_size_divisor, pad_args])	Stack multiple tensors to form a batch and pad the images to the max

class mmedit.models.data_preprocessors.**EditDataPreprocessor**(*mean: Sequence[Union[float, int]] = (0, 0, 0)*, *std: Sequence[Union[float, int]] = (255, 255, 255)*, *pad_size_divisor: int = 1*, *input_view=(- 1, 1, 1)*, *output_view=None*, *pad_args: dict = dict()*)

Bases: mmengine.model.BaseDataPreprocessor

Basic data pre-processor used for collating and copying data to the target device in mmediting.

EditDataPreprocessor performs data pre-processing according to the following steps:

- Collates the data sampled from dataloader.

- Copies data to the target device.
- Stacks the input tensor at the first dimension.

and post-processing of the output tensor of model.

TODO: Most editing methods have crop inputs to a same size, batched padding will be faster.

Parameters

- **mean** (*Sequence[float or int]*) – The pixel mean of R, G, B channels. Defaults to (0, 0, 0). If `mean` and `std` are not specified, ImgDataPreprocessor will normalize images to [0, 1].
- **std** (*Sequence[float or int]*) – The pixel standard deviation of R, G, B channels. (255, 255, 255). If `mean` and `std` are not specified, ImgDataPreprocessor will normalize images to [0, 1].
- **pad_size_divisor** (*int*) – The size of padded image should be divisible by `pad_size_divisor`. Defaults to 1.
- **input_view** (*Tuple | List*) – Tensor view of mean and std for input (without batch). Defaults to (-1, 1, 1) for (C, H, W).
- **output_view** (*Tuple | List | None*) – Tensor view of mean and std for output (without batch). If None, output_view=input_view. Defaults: None.
- **pad_args** (*dict*) – Args of F.pad. Default: dict().

forward(*data: Sequence[dict]*, *training: bool = False*) → Tuple[torch.Tensor, Optional[list]]

Pre-process the data into the model input format.

After the data pre-processing of `collate_data()`, `forward` will stack the input tensor list to a batch tensor at the first dimension.

Parameters

- **data** (*Sequence[dict]*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. Default: False.

Returns Data in the same format as the model input.

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

destructor(*batch_tensor: torch.Tensor*)

Destructor of data processor. Destruct padding, normalization and dissolve batch.

Parameters **batch_tensor** (*Tensor*) – Batched output.

Returns Destructed output.

Return type Tensor

mmedit.models.data_preprocessors.**split_batch**(*batch_tensor: torch.Tensor*, *padded_sizes: torch.Tensor*)

reverse operation of `stack_batch`.

Parameters

- **batch_tensor** (*Tensor*) – The 4D-tensor or 5D-tensor. Tensor.dim == tensor_list[0].dim + 1
- **padded_sizes** (*Tensor*) – The padded sizes of each tensor.

Returns A list of tensors with the same dim.

Return type tensor_list (List[Tensor])

mmedit.models.data_preprocessors.**stack_batch**(*tensor_list: List[torch.Tensor]*, *pad_size_divisor: int = 1*, *pad_args: dict = dict()*)

Stack multiple tensors to form a batch and pad the images to the max shape use the right bottom padding mode in these images.

If `pad_size_divisor > 0`, add padding to ensure the shape of each dim is divisible by `pad_size_divisor`.

Parameters

- **tensor_list** (*List[Tensor]*) – A list of tensors with the same dim.
- **pad_size_divisor** (*int*) – If `pad_size_divisor > 0`, add padding to ensure the shape of each dim is divisible by `pad_size_divisor`. This depends on the model, and many models need to be divisible by 32. Defaults to 1
- **pad_args** (*dict*) – The padding args.

Returns The 4D-tensor or 5D-tensor. Tensor.dim == tensor_list[0].dim + 1 padded_sizes (Tensor): The padded sizes of each tensor.

Return type batch_tensor (Tensor)

class mmedit.models.data_preprocessors.**GenDataPreprocessor**(*mean: Sequence[Union[float, int]] = (127.5, 127.5, 127.5)*, *std: Sequence[Union[float, int]] = (127.5, 127.5, 127.5)*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *rgb_to_bgr: bool = False*, *non_image_keys: Optional[Tuple[str, List[str]]] = None*, *non_concentate_keys: Optional[Tuple[str, List[str]]] = None*)

Bases: `mmengine.model.ImgDataPreprocessor`

Image pre-processor for generative models. This class provide normalization and bgr to rgb conversion for image tensor inputs. The input of this classes should be dict which keys are *inputs* and *data_samples*.

Besides to process tensor *inputs*, this class support dict as *inputs*. - If the value is *Tensor* and the corresponding key is not contained in *_NON_IMAGE_KEYS*, it will be processed as image tensor. - If the value is *Tensor* and the corresponding key belongs to *_NON_IMAGE_KEYS*, it will not remains unchanged. - If value is string or integer, it will not remains unchanged.

Parameters

- **mean** (*Sequence[float or int], optional*) – The pixel mean of image channels. If `bgr_to_rgb=True` it means the mean value of R, G, B channels. If it is not specified, images will not be normalized. Defaults None.
- **std** (*Sequence[float or int], optional*) – The pixel standard deviation of image channels. If `bgr_to_rgb=True` it means the standard deviation of R, G, B channels. If it is not specified, images will not be normalized. Defaults None.
- **pad_size_divisor** (*int*) – The size of padded image should be divisible by `pad_size_divisor`. Defaults to 1.
- **pad_value** (*float or int*) – The padded pixel value. Defaults to 0.
- **bgr_to_rgb** (*bool*) – whether to convert image from BGR to RGB. Defaults to False.
- **rgb_to_bgr** (*bool*) – whether to convert image from RGB to RGB. Defaults to False.

_NON_IMAGE_KEYS = ['noise']

_NON_CONCENTATE_KEYS = ['num_batches', 'mode', 'sample_kwargs', 'eq_cfg']

cast_data(*data: CastData*) → CastData

Copying data to the target device.

Parameters **data** (*dict*) – Data returned by DataLoader.

Returns Inputs and data sample at target device.

Return type CollatedResult

_preprocess_image_tensor(*inputs: torch.Tensor*) → torch.Tensor

Process image tensor.

Parameters **inputs** (*Tensor*) – List of image tensor to process.

Returns Processed and stacked image tensor.

Return type Tensor

process_dict_inputs(*batch_inputs: dict*) → dict

Preprocess dict type inputs.

Parameters **batch_inputs** (*dict*) – Input dict.

Returns Preprocessed dict.

Return type dict

forward(*data: dict*, *training: bool = False*) → dict

Performs normalizationpadding and bgr2rgb conversion based on BaseDataPreprocessor.

Parameters

- **data** (*dict*) – Input data to process.
- **training** (*bool*) – Whether to enable training time augmentation. This is ignored for *GenDataPreprocessor*. Defaults to False.

Returns Data in the same format as the model input.

Return type dict

destructor(*batch_tensor: torch.Tensor*)

Destructor of data processor. Destruct padding, normalization and dissolve batch.

Parameters **batch_tensor** (*Tensor*) – Batched output.

Returns Destructed output.

Return type Tensor

class mmedit.models.data_preprocessors.**MattorPreprocessor**(*mean: float = [123.675, 116.28, 103.53]*, *std: float = [58.395, 57.12, 57.375]*, *bgr_to_rgb: bool = True*, *proc_inputs: str = 'normalize'*, *proc_trimap: str = 'rescale_to_zero_one'*, *proc_gt: str = 'rescale_to_zero_one'*)

Bases: mmengine.model.BaseDataPreprocessor

DataPreprocessor for matting models.

See base class BaseDataPreprocessor for detailed information.

Workflow as follow :

- Collate and move data to the target device.
- Convert inputs from bgr to rgb if the shape of input is (3, H, W).
- Normalize image with defined std and mean.
- Stack inputs to batch_inputs.

Parameters

- **mean** (*Sequence[float or int]*) – The pixel mean of R, G, B channels. Defaults to [123.675, 116.28, 103.53].
- **std** (*Sequence[float or int]*) – The pixel standard deviation of R, G, B channels. [58.395, 57.12, 57.375].
- **bgr_to_rgb** (*bool*) – whether to convert image from BGR to RGB. Defaults to True.
- **proc_inputs** (*str*) – Methods to process inputs. Default: 'normalize'. Available options are `normalize`.
- **proc_trimap** (*str*) – Methods to process gt tensors. Default: 'rescale_to_zero_one'. Available options are `rescale_to_zero_one` and `as-is`.
- **proc_gt** (*str*) – Methods to process gt tensors. Default: 'rescale_to_zero_one'. Available options are `rescale_to_zero_one` and `ignore`.

_proc_inputs(*inputs: List[torch.Tensor]*)

_proc_trimap(*trimaps: List[torch.Tensor]*)

_proc_gt(*data_samples*, *key*)

forward(*data: Sequence[dict]*, *training: bool = False*) → Tuple[torch.Tensor, list]

Pre-process input images, trimaps, ground-truth as configured.

Parameters

- **data** (*Sequence[dict]*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. Default: False.

Returns Batched inputs and list of data samples.

Return type Tuple[torch.Tensor, list]

collate_data(*data: Sequence[dict]*) → Tuple[list, list, list]

Collating and moving data to the target device.

See base class `BaseDataPreprocessor` for detailed information.

1.52 mmedit.models.editors

1.52.1 Package Contents

Classes

AOTBlockNeck	Dilation backbone used in AOT-GAN model.
AOTEncoderDecoder	Encoder-Decoder used in AOT-GAN model.
AOTInpaintor	Inpaintor for AOT-GAN method.
IDLossModel	Face id loss model.
BasicVSR	BasicVSR model for video super-resolution.
BasicVSRNet	BasicVSR network structure for video super-resolution.
BasicVSRPlusPlusNet	BasicVSR++ network structure.
BigGAN	Impelmentation of `Large Scale GAN Training for High Fidelity Natural
CAIN	CAIN model for Video Interpolation.
CAINNet	CAIN network structure.
CycleGAN	CycleGAN model for unpaired image-to-image translation.
DCGAN	Impelmentation of `Unsupervised Representation Learning with Deep
DDIMScheduler	`DDIMScheduler` support the diffusion and reverse process formulated
DDPMScheduler	
DenoisingUnet	Denoising Unet. This network receives a diffused image `x_t` and
ContextualAttentionModule	Contexture attention module.
ContextualAttentionNeck	Neck with contextual attention module.
DeepFillDecoder	Decoder used in DeepFill model.
DeepFillEncoder	Encoder used in DeepFill model.
DeepFillRefiner	Refiner used in DeepFill model.
DeepFillv1Discriminators	Discriminators used in DeepFillv1 model.
DeepFillv1Inpaintor	Inpaintor for deepfillv1 method.
DeepFillEncoderDecoder	Two-stage encoder-decoder structure used in DeepFill model.
DIC	DIC model for Face Super-Resolution.
DICNet	DIC network structure for face super-resolution.
FeedbackBlock	Feedback Block of DIC.
FeedbackBlockCustom	Custom feedback block, will be used as the first feedback block.
FeedbackBlockHeatmapAttention	Feedback block with HeatmapAttention.
LightCNN	LightCNN discriminator with input size 128 x 128.
MaxFeature	Conv2d or Linear layer with max feature selector.
DIM	Deep Image Matting model.
ClipWrapper	Clip Models wrapper for disco-diffusion.
DiscoDiffusion	Disco Diffusion (DD) is a Google Colab Notebook which leverages an AI
EDSRNet	EDSR network structure.
EDVR	EDVR model for video super-resolution.

continues on next page

Table 2 – continued from previous page

EDVRNet	EDVR network structure for video super-resolution.
EG3D	Implementation of `Efficient Geometry-aware 3D Generative Adversarial
ESRGAN	Enhanced SRGAN model for single image super-resolution.
RRDBNet	Networks consisting of Residual in Residual Dense Block, which is used
FBADecoder	Decoder for FBA matting.
FBAResnetDilated	ResNet-based encoder for FBA image matting.
FLAVR	FLAVR model for video interpolation.
FLAVRNet	PyTorch implementation of FLAVR for video frame interpolation.
GCA	Guided Contextual Attention image matting model.
GGAN	Impelmentation of *Geomoetric GAN*.
GLEANStyleGANv2	GLEAN (using StyleGANv2) architecture for super-resolution.
GLDecoder	Decoder used in Global&Local model.
GLDilationNeck	Dilation Backbone used in Global&Local model.
GLEncoder	Encoder used in Global&Local model.
GLEncoderDecoder	Encoder-Decoder used in Global&Local model.
AblatedDiffusionModel	Guided diffusion Model.
IconVSRNet	IconVSR network structure for video super-resolution.
DepthwiseIndexBlock	Depthwise index block.
HolisticIndexBlock	Holistic Index Block.
IndexedUpsample	Indexed upsample module.
IndexNet	IndexNet matting model.
IndexNetDecoder	Decoder for IndexNet.
IndexNetEncoder	Encoder for IndexNet.
InstColorization	Colorization InstColorization method.
LIIF	LIIF model for single image super-resolution.
MLPRefiner	Multilayer perceptrons (MLPs), refiner used in LIIF.
LSGAN	Impelmentation of *Least Squares Generative Adversarial Networks*.
MSPIEStyleGAN2	MS-PIE StyleGAN2.
PESinGAN	Positional Encoding in SinGAN.
NAFBaseline	The original version of Baseline model in "Simple Baseline for Image
NAFBaselineLocal	The original version of Baseline model in "Simple Baseline for Image
NAFNet	NAFNet.
NAFNetLocal	The original version of NAFNetLocal in "Simple Baseline for Image
MaskConvModule	Mask convolution module.
PartialConv2d	Implementation for partial convolution.
PConvDecoder	Decoder with partial conv.
PConvEncoder	Encoder with partial conv.
PConvEncoderDecoder	Encoder-Decoder with partial conv module.
PConvInpaintor	Inpaintor for Partial Convolution method.
ProgressiveGrowingGAN	Progressive Growing Unconditional GAN.
Pix2Pix	Pix2Pix model for paired image-to-image translation.
PlainDecoder	Simple decoder from Deep Image Matting.

continues on next page

Table 2 – continued from previous page

PlainRefiner	Simple refiner from Deep Image Matting.
RDNNet	RDN model for single image super-resolution.
RealBasicVSR	RealBasicVSR model for real-world video super-resolution.
RealBasicVSRNet	RealBasicVSR network structure for real-world video super-resolution.
RealESRGAN	Real-ESRGAN model for single image super-resolution.
UNetDiscriminatorWithSpectralNorm	A U-Net discriminator with spectral normalization.
Restormer	Restormer A PyTorch impl of: `Restormer: Efficient Transformer for High-
SAGAN	Impelmentation of *Self-Attention Generative Adversarial Networks*.
SinGAN	SinGAN.
SRCNNNet	SRCNN network structure for image super resolution.
SRGAN	SRGAN model for single image super-resolution.
ModifiedVGG	A modified VGG discriminator with input size 128 x 128.
MSRResNet	Modified SRResNet.
StableDiffusion	class to run stable diffsuion pipeline.
StyleGAN1	Implementation of `A Style-Based Generator Architecture for Generative
StyleGAN2	Impelmentation of `Analyzing and Improving the Image Quality of
StyleGAN3	Impelmentation of *Alias-Free Generative Adversarial Networks*. # noqa.
StyleGAN3Generator	StyleGAN3 Generator.
SwinIRNet	SwinIR
TDAN	TDAN model for video super-resolution.
TDANNet	TDAN network structure for video super-resolution.
TOFlowVFINet	PyTorch implementation of TOFlow for video frame interpolation.
TOFlowVSRNet	PyTorch implementation of TOFlow.
ToFResBlock	ResNet architecture.
LTE	Learnable Texture Extractor.
TTSR	TTSR model for Reference-based Image Super-Resolution.
SearchTransformer	Search texture reference by transformer.
TTSRDiscriminator	A discriminator for TTSR.
TTSRNet	TTSR network structure (main-net) for reference-based super-resolution.
WGANGP	Impelmentation of *Improved Training of Wasserstein GANs*.

class mmedit.models.editors.**AOTBlockNeck**(*in_channels=256*, *dilation_rates=(1, 2, 4, 8)*, *num_aotblock=8*, *act_cfg=dict(type='ReLU')*, ***kwargs*)

Bases: mmengine.model.BaseModule

Dilation backbone used in AOT-GAN model.

This implementation follows: Aggregated Contextual Transformations for High-Resolution Image Inpainting

Parameters

- **in_channels** (*int, optional*) – Channel number of input feature. Default: 256.

- **dilation_rates** (*Tuple[int], optional*) – The dilation rates used
- **Default** (*for AOT block.*) – (1, 2, 4, 8).
- **num_aotblock** (*int, optional*) – Number of AOT blocks. Default: 8.
- **act_cfg** (*dict, optional*) – Config dict for activation layer, "relu" by default.
- **kwargs** (*keyword arguments*) –

forward(*x*)

class mmedit.models.editors.**AOTEncoderDecoder**(*encoder=dict(type='AOTEncoder')*, *decoder=dict(type='AOTDecoder')*, *dilation_neck=dict(type='AOTBlockNeck')*)

Bases: mmedit.models.editors.global_local.GLEncoderDecoder

Encoder-Decoder used in AOT-GAN model.

This implementation follows: Aggregated Contextual Transformations for High-Resolution Image Inpainting The architecture of the encoder-decoder is: (conv2d x 3) –> (dilated conv2d x 8) –> (conv2d or deconv2d x 3).

Parameters

- **encoder** (*dict*) – Config dict to encoder.
- **decoder** (*dict*) – Config dict to build decoder.
- **dilation_neck** (*dict*) – Config dict to build dilation neck.

class mmedit.models.editors.**AOTInpaintor**(*data_preprocessor: Union[dict, mmengine.config.Config]*, *encdec: dict*, *disc: Optional[dict] = None*, *loss_gan: Optional[dict] = None*, *loss_gp: Optional[dict] = None*, *loss_disc_shift: Optional[dict] = None*, *loss_composed_percep: Optional[dict] = None*, *loss_out_percep: bool = False*, *loss_l1_hole: Optional[dict] = None*, *loss_l1_valid: Optional[dict] = None*, *loss_tv: Optional[dict] = None*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = None*)

Bases: *mmedit.models.base_models.OneStageInpaintor*

Inpaintor for AOT-GAN method.

This inpaintor is implemented according to the paper: Aggregated Contextual Transformations for High-Resolution Image Inpainting

forward_train_d(*data_batch*, *is_real*, *is_disc*, *mask*)

Forward function in discriminator training step.

In this function, we compute the prediction for each data batch (real or fake). Meanwhile, the standard gan loss will be computed with several proposed losses for stable training.

Parameters

- **data_batch** (*torch.Tensor*) – Batch of real data or fake data.
- **is_real** (*bool*) – If True, the gan loss will regard this batch as real data. Otherwise, the gan loss will regard this batch as fake data.
- **is_disc** (*bool*) – If True, this function is called in discriminator training step. Otherwise, this function is called in generator training step. This will help us to compute different types of adversarial loss, like LSGAN.

- **mask** (*torch.Tensor*) – Mask of data.

Returns Contains the loss items computed in this function.

Return type dict

generator_loss(*fake_res*, *fake_img*, *gt*, *mask*, *masked_img*)

Forward function in generator training step.

In this function, we mainly compute the loss items for generator with the given (fake_res, fake_img). In general, the *fake_res* is the direct output of the generator and the *fake_img* is the composition of direct output and ground-truth image.

Parameters

- **fake_res** (*torch.Tensor*) – Direct output of the generator.
- **fake_img** (*torch.Tensor*) – Composition of *fake_res* and ground-truth image.
- **gt** (*torch.Tensor*) – Ground-truth image.
- **mask** (*torch.Tensor*) – Mask image.
- **masked_img** (*torch.Tensor*) – Composition of mask image and ground-truth image.

Returns

Dict contains the results computed within this function for visualization and dict contains the loss items computed in this function.

Return type tuple(dict)

forward_tensor(*inputs*, *data_samples*)

Forward function in tensor mode.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns

Direct output of the generator and composition of *fake_res* and ground-truth image.

Return type tuple

train_step(*data: List[dict]*, *optim_wrapper*)

Train step function.

In this function, the inpaintor will finish the train step following the pipeline: 1. get fake res/image 2. compute reconstruction losses for generator 3. compute adversarial loss for discriminator 4. optimize generator 5. optimize discriminator

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns

Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

class mmedit.models.editors.**IDLossModel**(*ir_se50_weights=None*)

Bases: torch.nn.Module

Face id loss model.

Parameters **ir_se50_weights** (*str, optional*) – Url of ir-se50 weights. Defaults to None.

_ir_se50_url = https://gg0ltg.by.files.1drv.com/y4m3fNNszG03z9n8JQ7EhdtQKW8tQVQMFBisPVRgoXi_UfP8pKSSqv8RJNmHy2Ja...

extract_feats(*x*)

Extracting face features.

Parameters **x** (*torch.Tensor*) – Image tensor of faces.

Returns Face features.

Return type torch.Tensor

forward(*pred=None*, *gt=None*)

Calculate face loss.

Parameters

- **pred** (*torch.Tensor, optional*) – Predictions of face images. Defaults to None.
- **gt** (*torch.Tensor, optional*) – Ground truth of face images. Defaults to None.

Returns

A tuple contain face similarity loss and improvement.

Return type Tuple(float, float)

class mmedit.models.editors.**BasicVSR**(*generator*, *pixel_loss*, *ensemble=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.BaseEditModel

BasicVSR model for video super-resolution.

Note that this model is used for IconVSR.

Paper: BasicVSR: The Search for Essential Components in Video Super-Resolution and Beyond, CVPR, 2021

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **ensemble** (*dict*) – Config for ensemble. Default: None.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.

check_if_mirror_extended(*lrs*)

Check whether the input is a mirror-extended sequence.

If mirror-extended, the i-th (i=0, …, t-1) frame is equal to the (t-1-i)-th frame.

Parameters **lrs** (*tensor*) – Input LR images with shape (n, t, c, h, w)

forward_train(*inputs*, *data_samples=None*, ***kwargs*)

Forward training. Returns dict of losses of training.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.

Returns Dict of losses.

Return type dict

forward_inference(*inputs*, *data_samples=None*, ***kwargs*)

Forward inference. Returns predictions of validation, testing.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.

Returns predictions.

Return type List[EditDataSample]

class mmedit.models.editors.**BasicVSRNet**(*mid_channels=64*, *num_blocks=30*, *spynet_pretrained=None*)

Bases: mmengine.model.BaseModule

BasicVSR network structure for video super-resolution.

Support only x4 upsampling.

Paper: BasicVSR: The Search for Essential Components in Video Super-Resolution and Beyond, CVPR, 2021

Parameters

- **mid_channels** (*int*) – Channel number of the intermediate features. Default: 64.
- **num_blocks** (*int*) – Number of residual blocks in each propagation branch. Default: 30.
- **spynet_pretrained** (*str*) – Pre-trained model path of SPyNet. Default: None.

check_if_mirror_extended(*lrs*)

Check whether the input is a mirror-extended sequence.

If mirror-extended, the i-th (i=0, …, t-1) frame is equal to the (t-1-i)-th frame.

Parameters **lrs** (*tensor*) – Input LR images with shape (n, t, c, h, w)

compute_flow(*lrs*)

Compute optical flow using SPyNet for feature warping.

Note that if the input is an mirror-extended sequence, ‘flows_forward’ is not needed, since it is equal to ‘flows_backward.flip(1)’.

Parameters **lrs** (*tensor*) – Input LR images with shape (n, t, c, h, w)

Returns

Optical flow. 'flows_forward' corresponds to the flows used for forward-time propagation (current to previous). 'flows_backward' corresponds to the flows used for backward-time propagation (current to next).

Return type tuple(Tensor)

forward(*lrs*)

Forward function for BasicVSR.

Parameters **lrs** (*Tensor*) – Input LR sequence with shape (n, t, c, h, w).

Returns Output HR sequence with shape (n, t, c, 4h, 4w).

Return type Tensor

class mmedit.models.editors.**BasicVSRPlusPlusNet**(*mid_channels=64*, *num_blocks=7*, *max_residue_magnitude=10*, *is_low_res_input=True*, *spynet_pretrained=None*, *cpu_cache_length=100*)

Bases: mmengine.model.BaseModule

BasicVSR++ network structure.

Support either x4 upsampling or same size output.

Paper: BasicVSR++: Improving Video Super-Resolution with Enhanced Propagation and Alignment

Parameters

- **mid_channels** (*int, optional*) – Channel number of the intermediate features. Default: 64.
- **num_blocks** (*int, optional*) – The number of residual blocks in each propagation branch. Default: 7.
- **max_residue_magnitude** (*int*) – The maximum magnitude of the offset residue (Eq. 6 in paper). Default: 10.
- **is_low_res_input** (*bool, optional*) – Whether the input is low-resolution or not. If False, the output resolution is equal to the input resolution. Default: True.
- **spynet_pretrained** (*str, optional*) – Pre-trained model path of SPyNet. Default: None.
- **cpu_cache_length** (*int, optional*) – When the length of sequence is larger than this value, the intermediate features are sent to CPU. This saves GPU memory, but slows down the inference speed. You can increase this number if you have a GPU with large memory. Default: 100.

check_if_mirror_extended(*lqs*)

Check whether the input is a mirror-extended sequence.

If mirror-extended, the i-th (i=0, ..., t-1) frame is equal to the (t-1-i)-th frame.

Parameters **lqs** (*tensor*) – Input low quality (LQ) sequence with shape (n, t, c, h, w).

compute_flow(*lqs*)

Compute optical flow using SPyNet for feature alignment.

Note that if the input is an mirror-extended sequence, 'flows_forward' is not needed, since it is equal to 'flows_backward.flip(1)'.

Parameters **lqs** (*tensor*) – Input low quality (LQ) sequence with shape (n, t, c, h, w).

Returns

Optical flow. 'flows_forward' corresponds to the flows used for forward-time propagation (current to previous). 'flows_backward' corresponds to the flows used for backward-time propagation (current to next).

Return type tuple(Tensor)

`propagate`(*feats*, *flows*, *module_name*)

Propagate the latent features throughout the sequence.

Parameters

- **dict** (*feats*) – Features from previous branches. Each component is a list of tensors with shape (n, c, h, w).
- **flows** (*tensor*) – Optical flows with shape (n, t - 1, 2, h, w).
- **module_name** (*str*) – The name of the propagation branches. Can either be 'backward_1', 'forward_1', 'backward_2', 'forward_2'.

Returns

A dictionary containing all the propagated features. Each key in the dictionary corresponds to a propagation branch, which is represented by a list of tensors.

Return type dict(list[tensor])

`upsample`(*lqs*, *feats*)

Compute the output image given the features.

Parameters

- **lqs** (*tensor*) – Input low quality (LQ) sequence with shape (n, t, c, h, w).
- **feats** (*dict*) – The features from the propagation branches.

Returns Output HR sequence with shape (n, t, c, 4h, 4w).

Return type Tensor

`forward`(*lqs*)

Forward function for BasicVSR++.

Parameters **lqs** (*tensor*) – Input low quality (LQ) sequence with shape (n, t, c, h, w).

Returns Output HR sequence with shape (n, t, c, 4h, 4w).

Return type Tensor

`class mmedit.models.editors.BigGAN`(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *num_classes: Optional[int] = None*, *ema_config: Optional[Dict] = None*)

Bases: *mmedit.models.base_models.BaseConditionalGAN*

Impelmentation of Large Scale GAN Training for High Fidelity Natural Image Synthesis (BigGAN).

Detailed architecture can be found in `BigGANGenerator` and `BigGANDiscriminator`

Parameters

- **generator** (*ModelType*) – The config or model of the generator.

- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or GenDataPreprocessor.
- **generator_steps** (*int*) – Number of times the generator was completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – Number of times the discriminator was completely updated before the generator is updated. Defaults to 1.
- **noise_size** (*Optional[int]*) – Size of the input noise vector. Default to 128.
- **num_classes** (*Optional[int]*) – The number classes you would like to generate. Defaults to None.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*) → Tuple

Get disc loss. BigGAN use hinge loss to train the discriminator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

gen_loss(*disc_pred_fake*)

Get disc loss. BigGAN use hinge loss to train the generator.

Parameters **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.

- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.editors.**CAIN**(*generator: dict*, *pixel_loss: dict*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *required_frames: int = 2*, *step_frames: int = 1*, *init_cfg: Optional[dict] = None*, *data_preprocessor: Optional[dict] = None*)

Bases: *mmedit.models.base_models.BasicInterpolator*

CAIN model for Video Interpolation.

Paper: Channel Attention Is All You Need for Video Frame Interpolation Ref repo: https://github.com/myungsub/CAIN

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **required_frames** (*int*) – Required frames in each process. Default: 2
- **step_frames** (*int*) – Step size of video frame interpolation. Default: 1
- **init_cfg** (*dict, optional*) – The weight initialized config for `BaseModule`.
- **data_preprocessor** (*dict, optional*) – The pre-process config of `BaseDataPreprocessor`.

init_cfg

Initialization config dict.

Type dict, optional

data_preprocessor

Used for pre-processing data sampled by dataloader to the format accepted by `forward()`.

Type `BaseDataPreprocessor`

forward_inference(*inputs*, *data_samples=None*)

Forward inference. Returns predictions of validation, testing, and simple inference.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by *data_preprocessor*.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by *data_preprocessor*.

Returns predictions.

Return type List[EditDataSample]

class mmedit.models.editors.**CAINNet**(*in_channels=3*, *kernel_size=3*, *num_block_groups=5*, *num_block_layers=12*, *depth=3*, *reduction=16*, *norm=None*, *padding=7*, *act=nn.LeakyReLU(0.2, True)*, *init_cfg=None*)

Bases: mmengine.model.BaseModule

CAIN network structure.

Paper: Channel Attention Is All You Need for Video Frame Interpolation. Ref repo: https://github.com/myungsub/CAIN

Parameters

- **in_channels** (*int*) – Channel number of inputs. Default: 3.
- **kernel_size** (*int*) – Kernel size of CAINNet. Default: 3.
- **num_block_groups** (*int*) – Number of block groups. Default: 5.
- **num_block_layers** (*int*) – Number of blocks in a group. Default: 12.
- **depth** (*int*) – Down scale depth, scale = 2**depth. Default: 3.
- **reduction** (*int*) – Channel reduction of CA. Default: 16.
- **norm** (*str | None*) – Normalization layer. If it is None, no normalization is performed. Default: None.
- **padding** (*int*) – Padding of CAINNet. Default: 7.
- **act** (*function*) – activate function. Default: nn.LeakyReLU(0.2, True).
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.

forward(*imgs*, *padding_flag=False*)

Forward function.

Parameters

- **imgs** (*Tensor*) – Input tensor with shape (n, 2, c, h, w).
- **padding_flag** (*bool*) – Padding or not. Default: False.

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**CycleGAN**(**args*, *buffer_size=50*, *loss_config=dict(cycle_loss_weight=10.0, id_loss_weight=0.5)*, ***kwargs*)

Bases: *mmedit.models.base_models.BaseTranslationModel*

CycleGAN model for unpaired image-to-image translation.

Ref: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

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

Forward function for testing.

Parameters

- **img** (*tensor*) – Input image tensor.
- **target_domain** (*str*) – Target domain of output image.
- **kwargs** (*dict*) – Other arguments.

Returns Forward results.

Return type dict

`_get_disc_loss`(*outputs*)

Backward function for the discriminators.

Parameters **outputs** (*dict*) – Dict of forward results.

Returns Discriminators' loss and loss dict.

Return type dict

`_get_gen_loss`(*outputs*)

Backward function for the generators.

Parameters **outputs** (*dict*) – Dict of forward results.

Returns Generators' loss and loss dict.

Return type dict

`_get_opposite_domain`(*domain*)

Get the opposite domain respect to the input domain.

Parameters **domain** (*str*) – The input domain.

Returns The opposite domain.

Return type str

`train_step`(*data: dict*, *optim_wrapper: mmengine.optim.OptimWrapperDict*)

Training step function.

Parameters

- **data_batch** (*dict*) – Dict of the input data batch.
- **optimizer** (*dict[torch.optim.Optimizer]*) – Dict of optimizers for the generators and discriminators.
- **ddp_reducer** (Reducer | None, optional) – Reducer from ddp. It is used to prepare for `backward()` in ddp. Defaults to None.
- **running_status** (*dict | None, optional*) – Contains necessary basic information for training, e.g., iteration number. Defaults to None.

Returns Dict of loss, information for logger, the number of samples and results for visualization.

Return type dict

`test_step`(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

`val_step`(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

class mmedit.models.editors.**DCGAN**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Bases: *mmedit.models.base_models.BaseGAN*

Impelmentation of *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*.

Paper link: <https://arxiv.org/abs/1511.06434>`_ (DCGAN).

Detailed architecture can be found in `DCGANGenerator` # noqa and `DCGANDiscriminator` # noqa

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*) → Tuple

Get disc loss. DCGAN use the vanilla gan loss to train the discriminator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

gen_loss(*disc_pred_fake: torch.Tensor*) → Tuple

Get gen loss. DCGAN use the vanilla gan loss to train the generator.

Parameters **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.

- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.editors.**DDIMScheduler**(*num_train_timesteps=1000*, *beta_start=0.0001*, *beta_end=0.02*, *beta_schedule='linear'*, *variance_type='learned_range'*, *timestep_values=None*, *clip_sample=True*, *set_alpha_to_one=True*)

`DDIMScheduler` support the diffusion and reverse process formulated in https://arxiv.org/abs/2010.02502.

The code is heavily influenced by https://github.com/huggingface/diffusers/blob/main/src/diffusers/schedulers/scheduling_ddim.py. # noqa The difference is that we ensemble gradient-guided sampling in step function.

Parameters

- **num_train_timesteps** (*int, optional*) – _description_. Defaults to 1000.
- **beta_start** (*float, optional*) – _description_. Defaults to 0.0001.
- **beta_end** (*float, optional*) – _description_. Defaults to 0.02.
- **beta_schedule** (*str, optional*) – _description_. Defaults to “linear”.
- **variance_type** (*str, optional*) – _description_. Defaults to ‘learned_range’.
- **timestep_values** (*_type_, optional*) – _description_. Defaults to None.
- **clip_sample** (*bool, optional*) – _description_. Defaults to True.
- **set_alpha_to_one** (*bool, optional*) – _description_. Defaults to True.

set_timesteps(*num_inference_steps*, *offset=0*)

set time steps.

_get_variance(*timestep*, *prev_timestep*)

get variance.

step(*model_output: Union[torch.FloatTensor, numpy.ndarray]*, *timestep: int*, *sample: Union[torch.FloatTensor, numpy.ndarray]*, *cond_fn=None*, *cond_kwargs={}*, *eta: float = 0.0*, *use_clipped_model_output: bool = False*, *generator=None*)

step forward.

add_noise(*original_samples*, *noise*, *timesteps*)

add noise.

__len__()

class mmedit.models.editors.**DDPMScheduler**(*num_train_timesteps: int = 1000*, *beta_start: float = 0.0001*, *beta_end: float = 0.02*, *beta_schedule: str = 'linear'*, *trained_betas: Optional[Union[numpy.array, list]] = None*, *variance_type='fixed_small'*, *clip_sample=True*)

set_timesteps(*num_inference_steps*)

set timesteps.

_get_variance(*t*, *predicted_variance=None*, *variance_type=None*)

get variance.

step(*model_output: torch.FloatTensor*, *timestep: int*, *sample: torch.FloatTensor*, *predict_epsilon=True*, *cond_fn=None*, *cond_kwargs={}*, *generator=None*)

add_noise(*original_samples*, *noise*, *timesteps*)

add noise.

abstract training_loss(*model*, *x_0*, *t*)

abstract sample_timestep()

__len__()

class mmedit.models.editors.**DenoisingUnet**(*image_size*, *in_channels=3*, *base_channels=128*, *resblocks_per_downsample=3*, *num_timesteps=1000*, *use_rescale_timesteps=False*, *dropout=0*, *embedding_channels=- 1*, *num_classes=0*, *use_fp16=False*, *channels_cfg=None*, *output_cfg=dict(mean='eps', var='learned_range')*, *norm_cfg=dict(type='GN', num_groups=32)*, *act_cfg=dict(type='SiLU', inplace=False)*, *shortcut_kernel_size=1*, *use_scale_shift_norm=False*, *resblock_updown=False*, *num_heads=4*, *time_embedding_mode='sin'*, *time_embedding_cfg=None*, *resblock_cfg=dict(type='DenoisingResBlock')*, *attention_cfg=dict(type='MultiHeadAttention')*, *encoder_channels=None*, *downsample_conv=True*, *upsample_conv=True*, *downsample_cfg=dict(type='DenoisingDownsample')*, *upsample_cfg=dict(type='DenoisingUpsample')*, *attention_res=[16, 8]*, *pretrained=None*, *unet_type=''*, *down_block_types: Tuple[str] = ()*, *up_block_types: Tuple[str] = ()*, *cross_attention_dim=768*, *layers_per_block: int = 2*)

Bases: `mmengine.model.BaseModule`

Denoising Unet. This network receives a diffused image `x_t` and current timestep `t`, and returns a `output_dict` corresponding to the passed `output_cfg`.

`output_cfg` defines the number of channels and the meaning of the output. `output_cfg` mainly contains keys of `mean` and `var`, denoting how the network outputs mean and variance required for the denoising process. For `mean`: 1. `dict(mean='EPS')`: Model will predict noise added in the

diffusion process, and the `output_dict` will contain a key named `eps_t_pred`.

2. **`dict(mean='START_X')`: Model will direct predict the mean of the** original image *x_0*, and the `output_dict` will contain a key named `x_0_pred`.
3. **`dict(mean='X_TM1_PRED')`: Model will predict the mean of diffused** image at *t-1* timestep, and the `output_dict` will contain a key named `x_tm1_pred`.

For `var`: 1. `dict(var='FIXED_SMALL')` or `dict(var='FIXED_LARGE')`: Variance in

the denoising process is regarded as a fixed value. Therefore only 'mean' will be predicted, and the output channels will equal to the input image (e.g., three channels for RGB image.)

2. **`dict(var='LEARNED')`: Model will predict *log_variance* in the** denoising process, and the `output_dict` will contain a key named `log_var`.

3. **dict(var='LEARNED_RANGE'): Model will predict an interpolation** factor and the *log_variance* will be calculated as *factor * upper_bound + (1-factor) * lower_bound*. The `output_dict` will contain a key named `factor`.

If `var` is not `FIXED_SMALL` or `FIXED_LARGE`, the number of output channels will be the double of input channels, where the first half part contains predicted mean values and the other part is the predicted variance values. Otherwise, the number of output channels equals to the input channels, only containing the predicted mean values.

Parameters

- **image_size** (*int | list[int]*) – The size of image to denoise.
- **in_channels** (*int, optional*) – The input channels of the input image. Defaults as 3.
- **base_channels** (*int, optional*) – The basic channel number of the generator. The other layers contain channels based on this number. Defaults to 128.
- **resblocks_per_downsample** (*int, optional*) – Number of ResBlock used between two downsample operations. The number of ResBlock between upsample operations will be the same value to keep symmetry. Defaults to 3.
- **num_timesteps** (*int, optional*) – The total timestep of the denoising process and the diffusion process. Defaults to 1000.
- **use_rescale_timesteps** (*bool, optional*) – Whether rescale the input timesteps in range of [0, 1000]. Defaults to `True`.
- **dropout** (*float, optional*) – The probability of dropout operation of each ResBlock. Pass 0 to do not use dropout. Defaults as 0.
- **embedding_channels** (*int, optional*) – The output channels of time embedding layer and label embedding layer. If not passed (or passed -1), output channels of the embedding layers will set as four times of `base_channels`. Defaults to -1.
- **num_classes** (*int, optional*) – The number of conditional classes. If set to 0, this model will be degraded to an unconditional model. Defaults to 0.
- **channels_cfg** (*list | dict[list], optional*) – Config for input channels of the intermedia blocks. If list is passed, each element of the list indicates the scale factor for the input channels of the current block with regard to the `base_channels`. For block `i`, the input and output channels should be `channels_cfg[i] * base_channels` and `channels_cfg[i+1] * base_channels` If dict is provided, the key of the dict should be the output scale and corresponding value should be a list to define channels. Default: Please refer to `_defualt_channels_cfg`.
- **output_cfg** (*dict, optional*) – Config for output variables. Defaults to `dict(mean='eps', var='learned_range')`.
- **norm_cfg** (*dict, optional*) – The config for normalization layers. Defaults to `dict(type='GN', num_groups=32)`.
- **act_cfg** (*dict, optional*) – The config for activation layers. Defaults to `dict(type='SiLU', inplace=False)`.
- **shortcut_kernel_size** (*int, optional*) – The kernel size for shortcut conv in ResBlocks. The value of this argument will overwrite the default value of *resblock_cfg*. Defaults to *3*.
- **use_scale_shift_norm** (*bool, optional*) – Whether perform scale and shift after normalization operation. Defaults to True.

- **num_heads** (*int, optional*) – The number of attention heads. Defaults to 4.
- **time_embedding_mode** (*str, optional*) – Embedding method of `time_embedding`. Defaults to 'sin'.
- **time_embedding_cfg** (*dict, optional*) – Config for `time_embedding`. Defaults to None.
- **resblock_cfg** (*dict, optional*) – Config for ResBlock. Defaults to `dict(type='DenoisingResBlock')`.
- **attention_cfg** (*dict, optional*) – Config for attention operation. Defaults to `dict(type='MultiHeadAttention')`.
- **upsample_conv** (*bool, optional*) – Whether use conv in upsample block. Defaults to `True`.
- **downsample_conv** (*bool, optional*) – Whether use conv operation in downsample block. Defaults to `True`.
- **upsample_cfg** (*dict, optional*) – Config for upsample blocks. Defaults to `dict(type='DenoisingDownsample')`.
- **downsample_cfg** (*dict, optional*) – Config for downsample blocks. Defaults to `dict(type='DenoisingUpsample')`.
- **attention_res** (*int | list[int], optional*) – Resolution of feature maps to apply attention operation. Defaults to `[16, 8]`.
- **pretrained** (*str | dict, optional*) – Path for the pretrained model or dict containing information for pretained models whose necessary key is 'ckpt_path'. Besides, you can also provide 'prefix' to load the generator part from the whole state dict. Defaults to None.

_default_channels_cfg

forward(*x_t*, *t*, *encoder_hidden_states=None*, *label=None*, *return_noise=False*)

Forward function. :param x_t: Diffused image at timestep *t* to denoise. :type x_t: torch.Tensor :param t: Current timestep. :type t: torch.Tensor :param label: You can directly give a

batch of label through a `torch.Tensor` or offer a callable function to sample a batch of label data. Otherwise, the `None` indicates to use the default label sampler.

Parameters **return_noise** (*bool, optional*) – If True, inputted `x_t` and `t` will be returned in a dict with output desired by `output_cfg`. Defaults to False.

Returns If not `return_noise`

Return type torch.Tensor | dict

init_weights(*pretrained=None*)

Init weights for models.

We just use the initialization method proposed in the original paper.

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Defaults to None.

convert_to_fp16()

Convert the precision of the model to float16.

convert_to_fp32()

Convert the precision of the model to float32.

class mmedit.models.editors.**ContextualAttentionModule**(*unfold_raw_kernel_size=4*, *unfold_raw_stride=2*, *unfold_raw_padding=1*, *unfold_corr_kernel_size=3*, *unfold_corr_stride=1*, *unfold_corr_dilation=1*, *unfold_corr_padding=1*, *scale=0.5*, *fuse_kernel_size=3*, *softmax_scale=10*, *return_attention_score=True*)

Bases: mmengine.model.BaseModule

Contexture attention module.

The details of this module can be found in: Generative Image Inpainting with Contextual Attention

Parameters

- **unfold_raw_kernel_size** (*int*) – Kernel size used in unfolding raw feature. Default: 4.
- **unfold_raw_stride** (*int*) – Stride used in unfolding raw feature. Default: 2.
- **unfold_raw_padding** (*int*) – Padding used in unfolding raw feature. Default: 1.
- **unfold_corr_kernel_size** (*int*) – Kernel size used in unfolding context for computing correlation maps. Default: 3.
- **unfold_corr_stride** (*int*) – Stride used in unfolding context for computing correlation maps. Default: 1.
- **unfold_corr_dilation** (*int*) – Dilation used in unfolding context for computing correlation maps. Default: 1.
- **unfold_corr_padding** (*int*) – Padding used in unfolding context for computing correlation maps. Default: 1.
- **scale** (*float*) – The resale factor used in resize input features. Default: 0.5.
- **fuse_kernel_size** (*int*) – The kernel size used in fusion module. Default: 3.
- **softmax_scale** (*float*) – The scale factor for softmax function. Default: 10.
- **return_attention_score** (*bool*) – If True, the attention score will be returned. Default: True.

forward(*x*, *context*, *mask=None*)

Forward Function.

Parameters

- **x** (*torch.Tensor*) – Tensor with shape (n, c, h, w).
- **context** (*torch.Tensor*) – Tensor with shape (n, c, h, w).
- **mask** (*torch.Tensor*) – Tensor with shape (n, 1, h, w). Default: None.

Returns Features after contextural attention.

Return type tuple(torch.Tensor)

patch_correlation(*x*, *kernel*)

Calculate patch correlation.

Parameters

- **x** (*torch.Tensor*) – Input tensor.

- **kernel** (*torch.Tensor*) – Kernel tensor.

Returns Tensor with shape of (n, l, h, w).

Return type torch.Tensor

patch_copy_deconv(*attention_score*, *context_filter*)

Copy patches using deconv.

Parameters

- **attention_score** (*torch.Tensor*) – Tensor with shape of (n, l , h, w).
- **context_filter** (*torch.Tensor*) – Filter kernel.

Returns Tensor with shape of (n, c, h, w).

Return type torch.Tensor

fuse_correlation_map(*correlation_map*, *h_unfold*, *w_unfold*)

Fuse correlation map.

This operation is to fuse correlation map for increasing large consistent correlation regions.

The mechanism behind this op is simple and easy to understand. A standard 'Eye' matrix will be applied as a filter on the correlation map in horizontal and vertical direction.

The shape of input correlation map is (n, h_unfold*w_unfold, h, w). When adopting fusing, we will apply convolutional filter in the reshaped feature map with shape of (n, 1, h_unfold*w_fold, h*w).

A simple specification for horizontal direction is shown below:

```
      (h, (h, (h, (h,
       0)  1)  2)  3)  ...
(h, 0)
(h, 1)      1
(h, 2)          1
(h, 3)              1
...
```

calculate_unfold_hw(*input_size*, *kernel_size=3*, *stride=1*, *dilation=1*, *padding=0*)

Calculate (h, w) after unfolding.

The official implementation of *unfold* in pytorch will put the dimension (h, w) into *L*. Thus, this function is just to calculate the (h, w) according to the equation in: https://pytorch.org/docs/stable/nn.html#torch.nn.Unfold

calculate_overlap_factor(*attention_score*)

Calculate the overlap factor after applying deconv.

Parameters **attention_score** (*torch.Tensor*) – The attention score with shape of (n, c, h, w).

Returns The overlap factor will be returned.

Return type torch.Tensor

mask_correlation_map(*correlation_map*, *mask*)

Add mask weight for correlation map.

Add a negative infinity number to the masked regions so that softmax function will result in 'zero' in those regions.

Parameters

- **correlation_map** (*torch.Tensor*) – Correlation map with shape of (n, h_unfold*w_unfold, h_map, w_map).
- **mask** (*torch.Tensor*) – Mask tensor with shape of (n, c, h, w). '1' in the mask indicates masked region while '0' indicates valid region.

Returns Updated correlation map with mask.

Return type torch.Tensor

im2col(*img*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *normalize=False*, *return_cols=False*)

Reshape image-style feature to columns.

This function is used for unfold feature maps to columns. The details of this function can be found in: https://pytorch.org/docs/1.1.0/nn.html?highlight=unfold#torch.nn.Unfold

Parameters

- **img** (*torch.Tensor*) – Features to be unfolded. The shape of this feature should be (n, c, h, w).
- **kernel_size** (*int*) – In this function, we only support square kernel with same height and width.
- **stride** (*int*) – Stride number in unfolding. Default: 1.
- **padding** (*int*) – Padding number in unfolding. Default: 0.
- **dilation** (*int*) – Dilation number in unfolding. Default: 1.
- **normalize** (*bool*) – If True, the unfolded feature will be normalized. Default: False.
- **return_cols** (*bool*) – The official implementation in PyTorch of unfolding will return features with shape of (n, c*$prod{kernel_size}$, L). If True, the features will be reshaped to (n, L, c, kernel_size, kernel_size). Otherwise, the results will maintain the shape as the official implementation.

Returns Unfolded columns. If *return_cols* is True, the shape of output tensor is *(n, L, c, kernel_size, kernel_size)*. Otherwise, the shape will be *(n, c*$prod{kernel_size}$, L)*.

Return type torch.Tensor

class mmedit.models.editors.**ContextualAttentionNeck**(*in_channels*, *conv_type='conv'*, *conv_cfg=None*, *norm_cfg=None*, *act_cfg=dict(type='ELU')*, *contextual_attention_args=dict(softmax_scale=10.0)*, ***kwargs*)

Bases: mmengine.model.BaseModule

Neck with contextual attention module.

Parameters

- **in_channels** (*int*) – The number of input channels.
- **conv_type** (*str*) – The type of conv module. In DeepFillv1 model, the *conv_type* should be 'conv'. In DeepFillv2 model, the *conv_type* should be 'gated_conv'.
- **conv_cfg** (*dict | None*) – Config of conv module. Default: None.
- **norm_cfg** (*dict | None*) – Config of norm module. Default: None.
- **act_cfg** (*dict | None*) – Config of activation layer. Default: dict(type='ELU').

- **contextual_attention_args** (*dict*) – Config of contextual attention module. Default: dict(softmax_scale=10.).
- **kwargs** (*keyword arguments*) –

_conv_type

forward(*x*, *mask*)

Forward Function.

Parameters

- **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).
- **mask** (*torch.Tensor*) – Input tensor with shape of (n, 1, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**DeepFillDecoder**(*in_channels*, *conv_type='conv'*, *norm_cfg=None*, *act_cfg=dict(type='ELU')*, *out_act_cfg=dict(type='clip', min=- 1.0, max=1.0)*, *channel_factor=1.0*, ***kwargs*)

Bases: mmengine.model.BaseModule

Decoder used in DeepFill model.

This implementation follows: Generative Image Inpainting with Contextual Attention

Parameters

- **in_channels** (*int*) – The number of input channels.
- **conv_type** (*str*) – The type of conv module. In DeepFillv1 model, the *conv_type* should be 'conv'. In DeepFillv2 model, the *conv_type* should be 'gated_conv'.
- **norm_cfg** (*dict*) – Config dict to build norm layer. Default: None.
- **act_cfg** (*dict*) – Config dict for activation layer, "elu" by default.
- **out_act_cfg** (*dict*) – Config dict for output activation layer. Here, we provide commonly used *clamp* or *clip* operation.
- **channel_factor** (*float*) – The scale factor for channel size. Default: 1.
- **kwargs** (*keyword arguments*) –

_conv_type

forward(*input_dict*)

Forward Function.

Parameters **input_dict** (*dict | torch.Tensor*) – Input dict with middle features or torch.Tensor.

Returns Output tensor with shape of (n, c, h, w).

Return type torch.Tensor

class mmedit.models.editors.**DeepFillEncoder**(*in_channels=5*, *conv_type='conv'*, *norm_cfg=None*, *act_cfg=dict(type='ELU')*, *encoder_type='stage1'*, *channel_factor=1.0*, ***kwargs*)

Bases: mmengine.model.BaseModule

Encoder used in DeepFill model.

This implementation follows: Generative Image Inpainting with Contextual Attention

Parameters

- **in_channels** (*int*) – The number of input channels. Default: 5.
- **conv_type** (*str*) – The type of conv module. In DeepFillv1 model, the *conv_type* should be 'conv'. In DeepFillv2 model, the *conv_type* should be 'gated_conv'.
- **norm_cfg** (*dict*) – Config dict to build norm layer. Default: None.
- **act_cfg** (*dict*) – Config dict for activation layer, "elu" by default.
- **encoder_type** (*str*) – Type of the encoder. Should be one of ['stage1', 'stage2_conv', 'stage2_attention']. Default: 'stage1'.
- **channel_factor** (*float*) – The scale factor for channel size. Default: 1.
- **kwargs** (*keyword arguments*) –

_conv_type

forward(*x*)

Forward Function.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**DeepFillRefiner**(*encoder_attention=dict(type='DeepFillEncoder', encoder_type='stage2_attention')*, *encoder_conv=dict(type='DeepFillEncoder', encoder_type='stage2_conv')*, *dilation_neck=dict(type='GLDilationNeck', in_channels=128, act_cfg=dict(type='ELU'))*, *contextual_attention=dict(type='ContextualAttentionNeck', in_channels=128)*, *decoder=dict(type='DeepFillDecoder', in_channels=256)*)

Bases: mmengine.model.BaseModule

Refiner used in DeepFill model.

This implementation follows: Generative Image Inpainting with Contextual Attention.

Parameters

- **encoder_attention** (*dict*) – Config dict for encoder used in branch with contextual attention module.
- **encoder_conv** (*dict*) – Config dict for encoder used in branch with just convolutional operation.
- **dilation_neck** (*dict*) – Config dict for dilation neck in branch with just convolutional operation.
- **contextual_attention** (*dict*) – Config dict for contextual attention neck.
- **decoder** (*dict*) – Config dict for decoder used to fuse and decode features.

forward(*x*, *mask*)

Forward Function.

Parameters

- **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).
- **mask** (*torch.Tensor*) – Input tensor with shape of (n, 1, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**DeepFillv1Discriminators**(*global_disc_cfg*, *local_disc_cfg*)

Bases: mmengine.model.BaseModule

Discriminators used in DeepFillv1 model.

In DeepFillv1 model, the discriminators are independent without any concatenation like Global&Local model. Thus, we call this model *DeepFillv1Discriminators*. There exist a global discriminator and a local discriminator with global and local input respectively.

The details can be found in: Generative Image Inpainting with Contextual Attention.

Parameters

- **global_disc_cfg** (*dict*) – Config dict for global discriminator.
- **local_disc_cfg** (*dict*) – Config dict for local discriminator.

forward(*x*)

Forward function.

Parameters **x** (*tuple[torch.Tensor]*) – Contains global image and the local image patch.

Returns Contains the prediction from discriminators in global image and local image patch.

Return type tuple[torch.Tensor]

init_weights()

Init weights for models.

class mmedit.models.editors.**DeepFillv1Inpaintor**(*data_preprocessor: dict*, *encdec: dict*, *disc=None*, *loss_gan=None*, *loss_gp=None*, *loss_disc_shift=None*, *loss_composed_percep=None*, *loss_out_percep=False*, *loss_l1_hole=None*, *loss_l1_valid=None*, *loss_tv=None*, *stage1_loss_type=None*, *stage2_loss_type=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg: Optional[dict] = None*)

Bases: *mmedit.models.base_models.TwoStageInpaintor*

Inpaintor for deepfillv1 method.

This inpaintor is implemented according to the paper: Generative image inpainting with contextual attention

Importantly, this inpaintor is an example for using custom training schedule based on *TwoStageInpaintor*.

The training pipeline of deepfillv1 is as following:

```
if cur_iter < iter_tc:
    update generator with only l1 loss
else:
    update discriminator
    if cur_iter > iter_td:
        update generator with l1 loss and adversarial loss
```

The new attribute *cur_iter* is added for recording current number of iteration. The *train_cfg* contains the setting of the training schedule:

```
train_cfg = dict(
    start_iter=0,
    disc_step=1,
    iter_tc=90000,
    iter_td=100000
)
```

iter_tc and *iter_td* correspond to the notation T_C and T_D of the original paper.

Parameters

- **generator** (*dict*) – Config for encoder-decoder style generator.
- **disc** (*dict*) – Config for discriminator.
- **loss_gan** (*dict*) – Config for adversarial loss.
- **loss_gp** (*dict*) – Config for gradient penalty loss.
- **loss_disc_shift** (*dict*) – Config for discriminator shift loss.
- **loss_composed_percep** (*dict*) – Config for perceptual and style loss with composed image as input.
- **loss_out_percep** (*dict*) – Config for perceptual and style loss with direct output as input.
- **loss_l1_hole** (*dict*) – Config for l1 loss in the hole.
- **loss_l1_valid** (*dict*) – Config for l1 loss in the valid region.
- **loss_tv** (*dict*) – Config for total variation loss.
- **train_cfg** (*dict*) – Configs for training scheduler. *disc_step* must be contained for indicates the discriminator updating steps in each training step.
- **test_cfg** (*dict*) – Configs for testing scheduler.
- **init_cfg** (*dict, optional*) – Initialization config dict.

forward_train_d(*data_batch*, *is_real*, *is_disc*)

Forward function in discriminator training step.

In this function, we modify the default implementation with only one discriminator. In DeepFillv1 model, they use two separated discriminators for global and local consistency.

Parameters

- **data_batch** (*torch.Tensor*) – Batch of real data or fake data.
- **is_real** (*bool*) – If True, the gan loss will regard this batch as real data. Otherwise, the gan loss will regard this batch as fake data.
- **is_disc** (*bool*) – If True, this function is called in discriminator training step. Otherwise, this function is called in generator training step. This will help us to compute different types of adversarial loss, like LSGAN.

Returns Contains the loss items computed in this function.

Return type dict

two_stage_loss(*stage1_data*, *stage2_data*, *gt*, *mask*, *masked_img*)

Calculate two-stage loss.

Parameters

- **stage1_data** (*dict*) – Contain stage1 results.
- **stage2_data** (*dict*) – Contain stage2 results.
- **gt** (*torch.Tensor*) – Ground-truth image.
- **mask** (*torch.Tensor*) – Mask image.
- **masked_img** (*torch.Tensor*) – Composition of mask image and ground-truth image.

Returns Dict contains the results computed within this function for visualization and dict contains the loss items computed in this function.

Return type tuple(dict)

calculate_loss_with_type(*loss_type*, *fake_res*, *fake_img*, *gt*, *mask*, *prefix='stage1_'*, *fake_local=None*)

Calculate multiple types of losses.

Parameters

- **loss_type** (*str*) – Type of the loss.
- **fake_res** (*torch.Tensor*) – Direct results from model.
- **fake_img** (*torch.Tensor*) – Composited results from model.
- **gt** (*torch.Tensor*) – Ground-truth tensor.
- **mask** (*torch.Tensor*) – Mask tensor.
- **prefix** (*str, optional*) – Prefix for loss name. Defaults to 'stage1_'. # noqa
- **fake_local** (*torch.Tensor, optional*) – Local results from model. Defaults to None.

Returns Contain loss value with its name.

Return type dict

train_step(*data: List[dict]*, *optim_wrapper*)

Train step function.

In this function, the inpaintor will finish the train step following the pipeline:

1. get fake res/image
2. optimize discriminator (if have)
3. optimize generator

If *self.train_cfg.disc_step > 1*, the train step will contain multiple iterations for optimizing discriminator with different input data and only one iteration for optimizing gerator after *disc_step* iterations for discriminator.

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

class mmedit.models.editors.**DeepFillEncoderDecoder**(*stage1=dict(type='GLEncoderDecoder', encoder=dict(type='DeepFillEncoder'), decoder=dict(type='DeepFillDecoder', in_channels=128), dilation_neck=dict(type='GLDilationNeck', in_channels=128, act_cfg=dict(type='ELU'))), stage2=dict(type='DeepFillRefiner'), return_offset=False*)

Bases: mmengine.model.BaseModule

Two-stage encoder-decoder structure used in DeepFill model.

The details are in: Generative Image Inpainting with Contextual Attention

Parameters

- **stage1** (*dict*) – Config dict for building stage1 model. As DeepFill model uses Global&Local model as baseline in first stage, the stage1 model can be easily built with *GLEncoderDecoder*.
- **stage2** (*dict*) – Config dict for building stage2 model.
- **return_offset** (*bool*) – Whether to return offset feature in contextual attention module. Default: False.

forward(*x*)

Forward function.

Parameters **x** (*torch.Tensor*) – This input tensor has the shape of (n, 5, h, w). In channel dimension, we concatenate [masked_img, ones, mask] as DeepFillv1 models do.

Returns The first two item is the results from first and second stage. If set *return_offset* as True, the offset will be returned as the third item.

Return type tuple[torch.Tensor]

init_weights()

Init weights for models.

class mmedit.models.editors.**DIC**(*generator*, *pixel_loss*, *align_loss*, *discriminator=None*, *gan_loss=None*, *feature_loss=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.editors.srgan.SRGAN

DIC model for Face Super-Resolution.

Paper: Deep Face Super-Resolution with Iterative Collaboration between Attentive Recovery and Landmark Estimation.

Parameters

- **generator** (*dict*) – Config for the generator.
- **pixel_loss** (*dict*) – Config for the pixel loss.
- **align_loss** (*dict*) – Config for the align loss.
- **discriminator** (*dict*) – Config for the discriminator. Default: None.
- **gan_loss** (*dict*) – Config for the gan loss. Default: None.

- **feature_loss** (*dict*) – Config for the feature loss. Default: None.
- **train_cfg** (*dict*) – Config for train. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule. Default: None.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor. Default: None.

forward_tensor(*inputs*, *data_samples=None*, *training=False*)

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.
- **training** (*bool*) – Whether is training. Default: False.

Returns

results of forward inference and forward train.

Return type (Tensor | Tuple[List[Tensor]])

if_run_g()

Calculates whether need to run the generator step.

if_run_d()

Calculates whether need to run the discriminator step.

g_step(*batch_outputs*, *batch_gt_data*)

G step of GAN: Calculate losses of generator.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Dict of losses.

Return type dict

d_step_with_optim(*batch_outputs*, *batch_gt_data*, *optim_wrapper*)

D step with optim of GAN: Calculate losses of discriminator and run optim.

Parameters

- **batch_outputs** (*Tuple[Tensor]*) – Batch output of generator.
- **batch_gt_data** (*Tuple[Tensor]*) – Batch GT data.
- **optim_wrapper** (*OptimWrapper*) – Optim wrapper of discriminator.

Returns Dict of parsed losses.

Return type dict

static extract_gt_data(*data_samples*)

extract gt data from data samples.

Parameters **data_samples** (*list*) – List of EditDataSample.

Returns Extract gt data.

Return type Tensor

class mmedit.models.editors.**DICNet**(*in_channels*, *out_channels*, *mid_channels*, *num_blocks=6*, *hg_mid_channels=256*, *hg_num_keypoints=68*, *num_steps=4*, *upscale_factor=8*, *detach_attention=False*, *prelu_init=0.2*, *num_heatmaps=5*, *num_fusion_blocks=7*)

Bases: mmengine.model.BaseModule

DIC network structure for face super-resolution.

Paper: Deep Face Super-Resolution with Iterative Collaboration between Attentive Recovery and Landmark Estimation

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels in the output image
- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64
- **num_blocks** (*tuple[int]*) – Block numbers in the trunk network. Default: 6
- **hg_mid_channels** (*int*) – Channel number of intermediate features of HourGlass. Default: 256
- **hg_num_keypoints** (*int*) – Keypoint number of HourGlass. Default: 68
- **num_steps** (*int*) – Number of iterative steps. Default: 4
- **upscale_factor** (*int*) – Upsampling factor. Default: 8
- **detach_attention** (*bool*) – Detached from the current tensor for heatmap or not.
- **prelu_init** (*float*) – *init* of PReLU. Default: 0.2
- **num_heatmaps** (*int*) – Number of heatmaps. Default: 5
- **num_fusion_blocks** (*int*) – Number of fusion blocks. Default: 7

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor.

Returns Forward results. sr_outputs (list[Tensor]): forward sr results. heatmap_outputs (list[Tensor]): forward heatmap results.

Return type Tensor

class mmedit.models.editors.**FeedbackBlock**(*mid_channels*, *num_blocks*, *upscale_factor*, *padding=2*, *prelu_init=0.2*)

Bases: torch.nn.Module

Feedback Block of DIC.

It has a style of:

```
----- Module ----->
  ^             |
  |_____________|
```

Parameters

- **mid_channels** (*int*) – Number of channels in the intermediate features.
- **num_blocks** (*int*) – Number of blocks.
- **upscale_factor** (*int*) – upscale factor.
- **padding** (*int*) – Padding size. Default: 2.
- **prelu_init** (*float*) – *init* of PReLU. Default: 0.2

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**FeedbackBlockCustom**(*in_channels*, *mid_channels*, *num_blocks*, *upscale_factor*)

Bases: *FeedbackBlock*

Custom feedback block, will be used as the first feedback block.

Parameters

- **in_channels** (*int*) – Number of channels in the input features.
- **mid_channels** (*int*) – Number of channels in the intermediate features.
- **num_blocks** (*int*) – Number of blocks.
- **upscale_factor** (*int*) – upscale factor.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**FeedbackBlockHeatmapAttention**(*mid_channels*, *num_blocks*, *upscale_factor*, *num_heatmaps*, *num_fusion_blocks*, *padding=2*, *prelu_init=0.2*)

Bases: *FeedbackBlock*

Feedback block with HeatmapAttention.

Parameters

- **in_channels** (*int*) – Number of channels in the input features.
- **mid_channels** (*int*) – Number of channels in the intermediate features.

- **num_blocks** (*int*) – Number of blocks.
- **upscale_factor** (*int*) – upscale factor.
- **padding** (*int*) – Padding size. Default: 2.
- **prelu_init** (*float*) – *init* of PReLU. Default: 0.2

forward(*x*, *heatmap*)

Forward function.

Parameters

- **x** (*Tensor*) – Input feature tensor.
- **heatmap** (*Tensor*) – Input heatmap tensor.

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**LightCNN**(*in_channels*)

Bases: mmengine.model.BaseModule

LightCNN discriminator with input size 128 x 128.

It is used to train DICGAN.

Parameters **in_channels** (*int*) – Channel number of inputs.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor.

Returns Forward results.

Return type Tensor

init_weights(*pretrained=None*, *strict=True*)

Init weights for models.

Parameters

- **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Defaults to None.
- **strict** (*boo, optional*) – Whether strictly load the pretrained model. Defaults to True.

class mmedit.models.editors.**MaxFeature**(*in_channels*, *out_channels*, *kernel_size=3*, *stride=1*, *padding=1*, *filter_type='conv2d'*)

Bases: torch.nn.Module

Conv2d or Linear layer with max feature selector.

Generate feature maps with double channels, split them and select the max feature.

Parameters

- **in_channels** (*int*) – Channel number of inputs.
- **out_channels** (*int*) – Channel number of outputs.
- **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: 1
- **filter_type** (*str*) – Type of filter. Options are 'conv2d' and 'linear'. Default: 'conv2d'.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor.

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**DIM**(*data_preprocessor*, *backbone*, *refiner=None*, *train_cfg=None*, *test_cfg=None*, *loss_alpha=None*, *loss_comp=None*, *loss_refine=None*, *init_cfg: Optional[dict] = None*)

Bases: *mmedit.models.base_models.BaseMattor*

Deep Image Matting model.

https://arxiv.org/abs/1703.03872

Note: For `(self.train_cfg.train_backbone, self.train_cfg.train_refiner)`:

- `(True, False)` corresponds to the encoder-decoder stage in the paper.
- `(False, True)` corresponds to the refinement stage in the paper.
- `(True, True)` corresponds to the fine-tune stage in the paper.

Parameters

- **data_preprocessor** (*dict, optional*) – Config of data pre-processor.
- **backbone** (*dict*) – Config of backbone.
- **refiner** (*dict*) – Config of refiner.
- **loss_alpha** (*dict*) – Config of the alpha prediction loss. Default: None.
- **loss_comp** (*dict*) – Config of the composition loss. Default: None.
- **loss_refine** (*dict*) – Config of the loss of the refiner. Default: None.
- **train_cfg** (*dict*) – Config of training. In `train_cfg`, `train_backbone` should be specified. If the model has a refiner, `train_refiner` should be specified.
- **test_cfg** (*dict*) – Config of testing. In `test_cfg`, If the model has a refiner, `train_refiner` should be specified.
- **init_cfg** (*dict, optional*) – The weight initialized config for `BaseModule`. Default: None.

property with_refiner

Whether the matting model has a refiner.

init_weights()

Initialize the model network weights.

`train`(*mode=True*)

Mode switcher.

Parameters `mode` (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

`freeze_backbone`()

Freeze the backbone and only train the refiner.

`_forward`(*x: torch.Tensor*, *, *refine: bool = True*) → Tuple[torch.Tensor, torch.Tensor]

Raw forward function.

Parameters

- `x` (*torch.Tensor*) – Concatenation of merged image and trimap with shape (N, 4, H, W)
- `refine` (*bool*) – if forward through refiner

Returns pred_alpha, with shape (N, 1, H, W) torch.Tensor: pred_refine, with shape (N, 4, H, W)

Return type torch.Tensor

`_forward_test`(*inputs*)

Forward to get alpha prediction.

`_forward_train`(*inputs*, *data_samples*)

Defines the computation performed at every training call.

Parameters

- `inputs` (*torch.Tensor*) – Concatenation of normalized image and trimap shape (N, 4, H, W)
- `data_samples` (*list[EditDataSample]*) – Data samples containing: - gt_alpha (Tensor): Ground-truth of alpha

  shape (N, 1, H, W), normalized to 0 to 1.

  – **gt_fg (Tensor): Ground-truth of foreground** shape (N, C, H, W), normalized to 0 to 1.

  – **gt_bg (Tensor): Ground-truth of background** shape (N, C, H, W), normalized to 0 to 1.

Returns Contains the loss items and batch information.

Return type dict

`class` `mmedit.models.editors.`**`ClipWrapper`**(*clip_type*, **args*, ***kwargs*)

Bases: `torch.nn.Module`

Clip Models wrapper for disco-diffusion.

We provide wrappers for the clip models of `openai` and `mlfoundations`, where the user can specify `clip_type` as `clip` or `open_clip`, and then initialize a clip model using the same arguments as in the original codebase. The following clip models settings are provided in the official repo of disco diffusion:

Setting | Source | Arguments | # noqa

|:----------:|:----------:|:----------:| # noqa | ViTB32 | clip | name='ViT-B/32', jit=False | # noqa | ViTB16 | clip | name='ViT-B/16', jit=False | # noqa | ViTL14 | clip | name='ViT-L/14', jit=False | # noqa | ViTL14_336px | clip | name='ViT-L/14@336px', jit=False | # noqa | RN50 | clip | name='RN50', jit=False | # noqa | RN50x4 | clip | name='RN50x4', jit=False | # noqa | RN50x16 | clip | name='RN50x16', jit=False | # noqa | RN50x64 | clip | name='RN50x64', jit=False | # noqa | RN101 | clip | name='RN101', jit=False | # noqa | ViTB32_laion2b_e16 | open_clip | name='ViT-B-32', pretrained='laion2b_e16' | # noqa | ViTB32_laion400m_e31 | open_clip | model_name='ViT-B-32', pretrained='laion400m_e31' | # noqa | ViTB32_laion400m_32 | open_clip | model_name='ViT-B-32', pretrained='laion400m_e32' | # noqa | ViTB32quickgelu_laion400m_e31 | open_clip | model_name='ViT-B-32-quickgelu', pretrained='laion400m_e31' | # noqa | ViTB32quickgelu_laion400m_e32 | open_clip | model_name='ViT-B-32-quickgelu', pretrained='laion400m_e32' | # noqa | ViTB16_laion400m_e31 | open_clip | model_name='ViT-B-16', pretrained='laion400m_e31' | # noqa | ViTB16_laion400m_e32 | open_clip | model_name='ViT-B-16', pretrained='laion400m_e32' | # noqa | RN50_yffcc15m | open_clip | model_name='RN50', pretrained='yfcc15m' | # noqa | RN50_cc12m | open_clip | model_name='RN50', pretrained='cc12m' | # noqa | RN50_quickgelu_yfcc15m | open_clip | model_name='RN50-quickgelu', pretrained='yfcc15m' | # noqa | RN50_quickgelu_cc12m | open_clip | model_name='RN50-quickgelu', pretrained='cc12m' | # noqa | RN101_yfcc15m | open_clip | model_name='RN101', pretrained='yfcc15m' | # noqa | RN101_quickgelu_yfcc15m | open_clip | model_name='RN101-quickgelu', pretrained='yfcc15m' | # noqa

An example of a `clip_modes_cfg` is as follows: .. code-block:: python

clip_models = [dict(type='ClipWrapper', clip_type='clip', name='ViT-B/32', jit=False), dict(type='ClipWrapper', clip_type='clip', name='ViT-B/16', jit=False), dict(type='ClipWrapper', clip_type='clip', name='RN50', jit=False)

]

Parameters **clip_type** (*List[Dict]*) – The original source of the clip model. Whether be `clip` or `open_clip`.

forward(**args*, ***kwargs*)

Forward function.

class mmedit.models.editors.**DiscoDiffusion**(*unet*, *diffusion_scheduler*, *secondary_model=None*, *clip_models=[]*, *use_fp16=False*, *pretrained_cfgs=None*)

Bases: `torch.nn.Module`

Disco Diffusion (DD) is a Google Colab Notebook which leverages an AI Image generating technique called CLIP-Guided Diffusion to allow you to create compelling and beautiful images from just text inputs. Created by Somnai, augmented by Gandamu, and building on the work of RiversHaveWings, nshepperd, and many others.

Ref: Github Repo: https://github.com/alembics/disco-diffusion Colab: https://colab.research.google.com/github/alembics/disco-diffusion/blob/main/Disco_Diffusion.ipynb # noqa

Parameters

- **unet** (*ModelType*) – Config of denoising Unet.
- **diffusion_scheduler** (*ModelType*) – Config of diffusion_scheduler scheduler.
- **secondary_model** (*ModelType*) – A smaller secondary diffusion model trained by Katherine Crowson to remove noise from intermediate timesteps to prepare them for CLIP. Ref: https://twitter.com/rivershavewings/status/1462859669454536711 # noqa Defaults to None.
- **clip_models** (*list*) – Config of clip models. Defaults to [].
- **use_fp16** (*bool*) – Whether to use fp16 for unet model. Defaults to False.

- **pretrained_cfgs** (*dict*) – Path Config for pretrained weights. Usually this is a dict contains module name and the corresponding ckpt path. Defaults to None.

property device

Get current device of the model.

Returns The current device of the model.

Return type torch.device

load_pretrained_models(*pretrained_cfgs*)

Loading pretrained weights to model. pretrained_cfgs is a dict consist of module name as key and checkpoint path as value.

Parameters

- **pretrained_cfgs** (*dict*) – Path Config for pretrained weights.
- **the** (*Usually this is a dict contains module name and*) –
- **None.** (*corresponding ckpt path. Defaults to*) –

infer(*scheduler_kwargs=None*, *height=None*, *width=None*, *init_image=None*, *batch_size=1*, *num_inference_steps=1000*, *skip_steps=0*, *show_progress=False*, *text_prompts=[]*, *image_prompts=[]*, *eta=0.8*, *clip_guidance_scale=5000*, *init_scale=1000*, *tv_scale=0.0*, *sat_scale=0.0*, *range_scale=150*, *cut_overview=[12] * 400 + [4] * 600*, *cut_innercut=[4] * 400 + [12] * 600*, *cut_ic_pow=[1] * 1000*, *cut_icgray_p=[0.2] * 400 + [0] * 600*, *cutn_batches=4*, *seed=None*)

Inference API for disco diffusion.

Parameters

- **scheduler_kwargs** (*dict*) – Args for infer time diffusion scheduler. Defaults to None.
- **height** (*int*) – Height of output image. Defaults to None.
- **width** (*int*) – Width of output image. Defaults to None.
- **init_image** (*str*) – Initial image at the start point of denoising. Defaults to None.
- **batch_size** (*int*) – Batch size. Defaults to 1.
- **num_inference_steps** (*int*) – Number of inference steps. Defaults to 1000.
- **skip_steps** (*int*) – Denoising steps to skip, usually set with init_image. Defaults to 0.
- **show_progress** (*bool*) – Whether to show progress. Defaults to False.
- **text_prompts** (*list*) – Text prompts. Defaults to [].
- **image_prompts** (*list*) – Image prompts, this is not the same as init_image, they works the same way with text_prompts. Defaults to [].
- **eta** (*float*) – Eta for ddim sampling. Defaults to 0.8.
- **clip_guidance_scale** (*int*) – The Scale of influence of prompts on output image. Defaults to 1000.
- **seed** (*int*) – Sampling seed. Defaults to None.

class mmedit.models.editors.**EDSRNet**(*in_channels*, *out_channels*, *mid_channels=64*, *num_blocks=16*, *upscale_factor=4*, *res_scale=1*, *rgb_mean=[0.4488, 0.4371, 0.404]*, *rgb_std=[1.0, 1.0, 1.0]*)

Bases: mmengine.model.BaseModule

EDSR network structure.

Paper: Enhanced Deep Residual Networks for Single Image Super-Resolution. Ref repo: https://github.com/thstkdgus35/EDSR-PyTorch

Parameters

- **in_channels** (*int*) – Channel number of inputs.
- **out_channels** (*int*) – Channel number of outputs.
- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64.
- **num_blocks** (*int*) – Block number in the trunk network. Default: 16.
- **upscale_factor** (*int*) – Upsampling factor. Support 2^n and 3. Default: 4.
- **res_scale** (*float*) – Used to scale the residual in residual block. Default: 1.
- **rgb_mean** (*list[float]*) – Image mean in RGB orders. Default: [0.4488, 0.4371, 0.4040], calculated from DIV2K dataset.
- **rgb_std** (*list[float]*) – Image std in RGB orders. In EDSR, it uses [1.0, 1.0, 1.0]. Default: [1.0, 1.0, 1.0].

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**EDVR**(*generator*, *pixel_loss*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.BaseEditModel

EDVR model for video super-resolution.

EDVR: Video Restoration with Enhanced Deformable Convolutional Networks.

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.

forward_train(*inputs*, *data_samples=None*)

Forward training. Returns dict of losses of training.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.

Returns Dict of losses.

Return type dict

class mmedit.models.editors.**EDVRNet**(*in_channels*, *out_channels*, *mid_channels=64*, *num_frames=5*, *deform_groups=8*, *num_blocks_extraction=5*, *num_blocks_reconstruction=10*, *center_frame_idx=2*, *with_tsa=True*, *init_cfg=None*)

Bases: mmengine.model.BaseModule

EDVR network structure for video super-resolution.

Now only support X4 upsampling factor. Paper: EDVR: Video Restoration with Enhanced Deformable Convolutional Networks.

Parameters

- **in_channels** (*int*) – Channel number of inputs.
- **out_channels** (*int*) – Channel number of outputs.
- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64.
- **num_frames** (*int*) – Number of input frames. Default: 5.
- **deform_groups** (*int*) – Deformable groups. Defaults: 8.
- **num_blocks_extraction** (*int*) – Number of blocks for feature extraction. Default: 5.
- **num_blocks_reconstruction** (*int*) – Number of blocks for reconstruction. Default: 10.
- **center_frame_idx** (*int*) – The index of center frame. Frame counting from 0. Default: 2.
- **with_tsa** (*bool*) – Whether to use TSA module. Default: True.
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.

forward(*x*)

Forward function for EDVRNet.

Parameters **x** (*Tensor*) – Input tensor with shape (n, t, c, h, w).

Returns SR center frame with shape (n, c, h, w).

Return type Tensor

init_weights()

Init weights for models.

class mmedit.models.editors.**EG3D**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *camera: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Bases: *mmedit.models.base_models.BaseConditionalGAN*

Implementation of *Efficient Geometry-aware 3D Generative Adversarial Networks*

<https://openaccess.thecvf.com/content/CVPR2022/papers/Chan_Efficient_Geometry-Aware_3D_Generative_Adversarial_Networks_CVPR_2022_paper.pdf>_ (EG3D). # noqa

Detailed architecture can be found in `TriplaneGenerator` and `DualDiscriminator`

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **camera** (*Optional[ModelType]*) – The pre-defined camera to sample random camera position. If you want to generate images or videos via high-level API, you must set this argument. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or `GenDataPreprocessor`.
- **generator_steps** (*int*) – Number of times the generator was completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – Number of times the discriminator was completely updated before the generator is updated. Defaults to 1.
- **noise_size** (*Optional[int]*) – Size of the input noise vector. Default to 128.
- **num_classes** (*Optional[int]*) – The number classes you would like to generate. Defaults to None.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.
- **loss_config** (*Optional[Dict]*) – The config for training losses. Defaults to None.

label_fn(*label: Optional[torch.Tensor] = None*, *num_batches: int = 1*) → torch.Tensor

Label sampling function for EG3D model.

Parameters **label** (*Optional[Tensor]*) – Conditional for EG3D model. If not passed, `self.camera` will be used to sample random camera-to-world and intrinsics matrix. Defaults to None.

Returns Conditional input for EG3D model.

Return type torch.Tensor

data_sample_to_label(*data_sample: mmedit.utils.typing.SampleList*) → Optional[torch.Tensor]

Get labels from input *data_sample* and pack to *torch.Tensor*. If no label is found in the passed *data_sample*, *None* would be returned.

Parameters **data_sample** (*List[EditDataSample]*) – Input data samples.

Returns Packed label tensor.

Return type Optional[torch.Tensor]

pack_to_data_sample(*output: Dict[str, torch.Tensor]*, *index: int*, *data_sample: Optional[*mmedit.structures.EditDataSample*] = None*) → *mmedit.structures.EditDataSample*

Pack output to data sample. If `data_sample` is not passed, a new EditDataSample will be instantiated. Otherwise, outputs will be added to the passed datasample.

Parameters

- **output** (*Dict[Tensor]*) – Output of the model.

- **index** (*int*) – The index to save.
- **data_sample** (*EditDataSample, optional*) – Data sample to save outputs. Defaults to None.

Returns Data sample with packed outputs.

Return type EditDataSample

forward(*inputs: mmedit.utils.typing.ForwardInputs*, *data_samples: Optional[list] = None*, *mode: Optional[str] = None*) → List[*mmedit.structures.EditDataSample*]

Sample images with the given inputs. If forward mode is 'ema' or 'orig', the image generated by corresponding generator will be returned. If forward mode is 'ema/orig', images generated by original generator and EMA generator will both be returned in a dict.

Parameters

- **inputs** (*ForwardInputs*) – Dict containing the necessary information (e.g. noise, num_batches, mode) to generate image.
- **data_samples** (*Optional[list]*) – Data samples collated by `data_preprocessor`. Defaults to None.
- **mode** (*Optional[str]*) – *mode* is not used in `BaseConditionalGAN`. Defaults to None.

Returns Generated images or image dict.

Return type List[EditDataSample]

interpolation(*num_images: int*, *num_batches: int = 4*, *mode: str = 'both'*, *sample_model: str = 'orig'*, *show_pbar: bool = True*) → List[dict]

Interpolation input and return a list of output results. We support three kinds of interpolation mode:

- **'camera': First generate style code with random noise and forward** camera. Then synthesis images with interpolated camera position and fixed style code.
- **'conditioning': First generate style code with fixed noise and** interpolated camera. Then synthesis images with style codes and forward camera.
- 'both': Generate images with interpolated camera position.

Parameters

- **num_images** (*int*) – The number of images want to generate.
- **num_batches** (*int, optional*) – The number of batches to generate at one time. Defaults to 4.
- **mode** (*str, optional*) – The interpolation mode. Supported choices are 'both', 'camera', and 'conditioning'. Defaults to 'both'.
- **sample_model** (*str, optional*) – The model used to generate images, support 'orig' and 'ema'. Defaults to 'orig'.
- **show_pbar** (*bool, optional*) – Whether display a progress bar during interpolation. Defaults to True.

Returns The list of output dict of each frame.

Return type List[dict]

class mmedit.models.editors.**ESRGAN**(*generator*, *discriminator=None*, *gan_loss=None*, *pixel_loss=None*, *perceptual_loss=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.editors.srgan.SRGAN

Enhanced SRGAN model for single image super-resolution.

Ref: ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks. It uses RaGAN for GAN updates: The relativistic discriminator: a key element missing from standard GAN.

Parameters

- **generator** (*dict*) – Config for the generator.
- **discriminator** (*dict*) – Config for the discriminator. Default: None.
- **gan_loss** (*dict*) – Config for the gan loss. Note that the loss weight in gan loss is only for the generator.
- **pixel_loss** (*dict*) – Config for the pixel loss. Default: None.
- **perceptual_loss** (*dict*) – Config for the perceptual loss. Default: None.
- **train_cfg** (*dict*) – Config for training. Default: None. You may change the training of gan by setting: *disc_steps*: how many discriminator updates after one generate update; *disc_init_steps*: how many discriminator updates at the start of the training. These two keys are useful when training with WGAN.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule. Default: None.

g_step(*batch_outputs: torch.Tensor*, *batch_gt_data: torch.Tensor*)

G step of GAN: Calculate losses of generator.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Dict of losses.

Return type dict

d_step_real(*batch_outputs: torch.Tensor*, *batch_gt_data: torch.Tensor*)

D step of real data.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Dict of losses.

Return type dict

d_step_fake(*batch_outputs: torch.Tensor*, *batch_gt_data*)

D step of fake data.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.

- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Dict of losses.

Return type dict

class mmedit.models.editors.**RRDBNet**(*in_channels*, *out_channels*, *mid_channels=64*, *num_blocks=23*, *growth_channels=32*, *upscale_factor=4*, *init_cfg=None*)

Bases: mmengine.model.BaseModule

Networks consisting of Residual in Residual Dense Block, which is used in ESRGAN and Real-ESRGAN.

ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks. Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data. # noqa: E501 Currently, it supports [x1/x2/x4] upsampling scale factor.

Parameters

- **in_channels** (*int*) – Channel number of inputs.
- **out_channels** (*int*) – Channel number of outputs.
- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64
- **num_blocks** (*int*) – Block number in the trunk network. Defaults: 23
- **growth_channels** (*int*) – Channels for each growth. Default: 32.
- **upscale_factor** (*int*) – Upsampling factor. Support x1, x2 and x4. Default: 4.

_supported_upscale_factors = [1, 2, 4]

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

init_weights()

Init weights for models.

class mmedit.models.editors.**FBADecoder**(*pool_scales*, *in_channels*, *channels*, *conv_cfg=None*, *norm_cfg=dict(type='BN')*, *act_cfg=dict(type='ReLU')*, *align_corners=False*)

Bases: torch.nn.Module

Decoder for FBA matting.

Parameters

- **pool_scales** (*tuple[int]*) – Pooling scales used in
- **Module.** (*Pooling Pyramid*) –
- **in_channels** (*int*) – Input channels.
- **channels** (*int*) – Channels after modules, before conv_seg.
- **conv_cfg** (*dict|None*) – Config of conv layers.
- **norm_cfg** (*dict|None*) – Config of norm layers.
- **act_cfg** (*dict*) – Config of activation layers.

- **align_corners** (*bool*) – align_corners argument of F.interpolate.

init_weights(*pretrained=None*)

Init weights for the model.

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Defaults to None.

forward(*inputs*)

Forward function.

Parameters **inputs** (*dict*) – Output dict of FbaEncoder.

Returns Predicted alpha, fg and bg of the current batch.

Return type tuple(Tensor)

class mmedit.models.editors.**FBAResnetDilated**(*depth: int*, *in_channels: int = 3*, *stem_channels: int = 64*, *base_channels: int = 64*, *num_stages: int = 4*, *strides: Sequence[int] = (1, 2, 2, 2)*, *dilations: Sequence[int] = (1, 1, 2, 4)*, *deep_stem: bool = False*, *avg_down: bool = False*, *frozen_stages: int = - 1*, *act_cfg: dict = dict(type='ReLU')*, *conv_cfg: Optional[dict] = None*, *norm_cfg: dict = dict(type='BN')*, *with_cp: bool = False*, *multi_grid: Optional[Sequence[int]] = None*, *contract_dilation: bool = False*, *zero_init_residual: bool = True*)

Bases: *mmedit.models.base_archs.ResNet*

ResNet-based encoder for FBA image matting.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (N, C, H, W).

Returns Output tensor.

Return type Tensor

class mmedit.models.editors.**FLAVR**(*generator: dict*, *pixel_loss: dict*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *required_frames: int = 2*, *step_frames: int = 1*, *init_cfg: Optional[dict] = None*, *data_preprocessor: Optional[dict] = None*)

Bases: *mmedit.models.base_models.BasicInterpolator*

FLAVR model for video interpolation.

Paper: FLAVR: Flow-Agnostic Video Representations for Fast Frame Interpolation

Ref repo: https://github.com/tarun005/FLAVR

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **required_frames** (*int*) – Required frames in each process. Default: 2

- **step_frames** (*int*) – Step size of video frame interpolation. Default: 1
- **init_cfg** (*dict, optional*) – The weight initialized config for `BaseModule`.
- **data_preprocessor** (*dict, optional*) – The pre-process config of `BaseDataPreprocessor`.

init_cfg

Initialization config dict.

Type dict, optional

data_preprocessor

Used for pre-processing data sampled by dataloader to the format accepted by `forward()`.

Type `BaseDataPreprocessor`

static merge_frames(*input_tensors*, *output_tensors*)

merge input frames and output frames.

Interpolate a frame between the given two frames.

Merged from [[in1, in2, in3, in4], [in2, in3, in4, in5], ...] [[out1], [out2], [out3], ...]

to [in1, in2, out1, in3, out2, ..., in(-3), out(-1), in(-2), in(-1)]

Parameters

- **input_tensors** (*Tensor*) – The input frames with shape [n, 4, c, h, w]
- **output_tensors** (*Tensor*) – The output frames with shape [n, 1, c, h, w].

Returns The final frames.

Return type list[np.array]

class mmedit.models.editors.**FLAVRNet**(*num_input_frames*, *num_output_frames*, *mid_channels_list=[512, 256, 128, 64]*, *encoder_layers_list=[2, 2, 2, 2]*, *bias=False*, *norm_cfg=None*, *join_type='concat'*, *up_mode='transpose'*, *init_cfg=None*)

Bases: `mmengine.model.BaseModule`

PyTorch implementation of FLAVR for video frame interpolation.

Paper: FLAVR: Flow-Agnostic Video Representations for Fast Frame Interpolation

Ref repo: https://github.com/tarun005/FLAVR

Parameters

- **num_input_frames** (*int*) – Number of input frames.
- **num_output_frames** (*int*) – Number of output frames.
- **mid_channels_list** (*list[int]*) – List of number of mid channels. Default: [512, 256, 128, 64]
- **encoder_layers_list** (*list[int]*) – List of number of layers in encoder. Default: [2, 2, 2, 2]
- **bias** (*bool*) – If `True`, adds a learnable bias to the conv layers. Default: `True`
- **norm_cfg** (*dict | None*) – Config dict for normalization layer. Default: None

- **join_type** (*str*) – Join type of tensors from decoder and encoder. Candidates are `concat` and `add`. Default: `concat`
- **up_mode** (*str*) – Up-mode UpConv3d, candidates are `transpose` and `trilinear`. Default: `transpose`
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.

forward(*images: torch.Tensor*)

Forward function.

Parameters **images** (*Tensor*) – Input frames tensor with shape (N, T, C, H, W).

Returns Output tensor.

Return type out (Tensor)

class mmedit.models.editors.**GCA**(*data_preprocessor*, *backbone*, *loss_alpha=None*, *init_cfg: Optional[dict] = None*, *train_cfg=None*, *test_cfg=None*)

Bases: *mmedit.models.base_models.BaseMattor*

Guided Contextual Attention image matting model.

https://arxiv.org/abs/2001.04069

Parameters

- **data_preprocessor** (*dict, optional*) – The pre-process config of `BaseDataPreprocessor`.
- **backbone** (*dict*) – Config of backbone.
- **loss_alpha** (*dict*) – Config of the alpha prediction loss. Default: None.
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.
- **train_cfg** (*dict*) – Config of training. In `train_cfg`, `train_backbone` should be specified. If the model has a refiner, `train_refiner` should be specified.
- **test_cfg** (*dict*) – Config of testing. In `test_cfg`, If the model has a refiner, `train_refiner` should be specified.

_forward(*inputs*)

Forward function.

Parameters **inputs** (*torch.Tensor*) – Input tensor.

Returns Output tensor.

Return type Tensor

_forward_test(*inputs*)

Forward function for testing GCA model.

Parameters **inputs** (*torch.Tensor*) – batch input tensor.

Returns Output tensor of model.

Return type Tensor

_forward_train(*inputs*, *data_samples*)

Forward function for training GCA model.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by `data_preprocessor`.

- **data_samples** (*List[BaseDataElement]*) – data samples collated by data_preprocessor.

Returns Contains the loss items and batch information.

Return type dict

class mmedit.models.editors.**GGAN**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Bases: *mmedit.models.base_models.BaseGAN*

Impelmentation of *Geomoetric GAN*.

<https://arxiv.org/abs/1705.02894>`_(GGAN).

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*) → Tuple

Get disc loss. GGAN use hinge loss to train the discriminator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

gen_loss(*disc_pred_fake*)

Get disc loss. GGAN use hinge loss to train the generator.

Parameters **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A dict of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.

- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.editors.**GLEANStyleGANv2**(*in_size*, *out_size*, *img_channels=3*, *rrdb_channels=64*, *num_rrdbs=23*, *style_channels=512*, *num_mlps=8*, *channel_multiplier=2*, *blur_kernel=[1, 3, 3, 1]*, *lr_mlp=0.01*, *default_style_mode='mix'*, *eval_style_mode='single'*, *mix_prob=0.9*, *init_cfg=None*, *fp16_enabled=False*, *bgr2rgb=False*)

Bases: `mmengine.model.BaseModule`

GLEAN (using StyleGANv2) architecture for super-resolution.

Paper: GLEAN: Generative Latent Bank for Large-Factor Image Super-Resolution, CVPR, 2021

This method makes use of StyleGAN2 and hence the arguments mostly follow that in 'StyleGAN2v2Generator'.

In StyleGAN2, we use a static architecture composing of a style mapping module and number of covolutional style blocks. More details can be found in: Analyzing and Improving the Image Quality of StyleGAN CVPR2020.

You can load pretrained model through passing information into `pretrained` argument. We have already offered official weights as follows:

- styelgan2-ffhq-config-f: http://download.openmmlab.com/mmediting/stylegan2/official_weights/stylegan2-ffhq-config-f-official_20210327_171224-bce9310c.pth # noqa
- stylegan2-horse-config-f: http://download.openmmlab.com/mmediting/stylegan2/official_weights/stylegan2-horse-config-f-official_20210327_173203-ef3e69ca.pth # noqa
- stylegan2-car-config-f: http://download.openmmlab.com/mmediting/stylegan2/official_weights/stylegan2-car-config-f-official_20210327_172340-8cfe053c.pth # noqa
- styelgan2-cat-config-f: http://download.openmmlab.com/mmediting/stylegan2/official_weights/stylegan2-cat-config-f-official_20210327_172444-15bc485b.pth # noqa
- stylegan2-church-config-f: http://download.openmmlab.com/mmediting/stylegan2/official_weights/stylegan2-church-config-f-official_20210327_172657-1d42b7d1.pth # noqa

If you want to load the ema model, you can just use following codes:

```
# ckpt_http is one of the valid path from http source
generator = StyleGANv2Generator(1024, 512,
                                pretrained=dict(
                                    ckpt_path=ckpt_http,
                                    prefix='generator_ema'))
```

Of course, you can also download the checkpoint in advance and set `ckpt_path` with local path. If you just want to load the original generator (not the ema model), please set the prefix with 'generator'.

Note that our implementation allows to generate BGR image, while the original StyleGAN2 outputs RGB images by default. Thus, we provide `bgr2rgb` argument to convert the image space.

Parameters

- **in_size** (*int*) – The size of the input image.

- **out_size** (*int*) – The output size of the StyleGAN2 generator.
- **img_channels** (*int*) – Number of channels of the input images. 3 for RGB image and 1 for grayscale image. Default: 3.
- **rrdb_channels** (*int*) – Number of channels of the RRDB features. Default: 64.
- **num_rrdbs** (*int*) – Number of RRDB blocks in the encoder. Default: 23.
- **style_channels** (*int*) – The number of channels for style code. Default: 512.
- **num_mlps** (*int, optional*) – The number of MLP layers. Defaults to 8.
- **channel_multiplier** (*int, optional*) – The mulitiplier factor for the channel number. Defaults to 2.
- **blur_kernel** (*list, optional*) – The blurry kernel. Defaults to [1, 3, 3, 1].
- **lr_mlp** (*float, optional*) – The learning rate for the style mapping layer. Defaults to 0.01.
- **default_style_mode** (*str, optional*) – The default mode of style mixing. In training, we defaultly adopt mixing style mode. However, in the evaluation, we use 'single' style mode. *['mix', 'single']* are currently supported. Defaults to 'mix'.
- **eval_style_mode** (*str, optional*) – The evaluation mode of style mixing. Defaults to 'single'.
- **mix_prob** (*float, optional*) – Mixing probability. The value should be in range of [0, 1]. Defaults to 0.9.
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.
- **fp16_enabled** (*bool, optional*) – Whether to use fp16 training in this module. Defaults to False.
- **bgr2rgb** (*bool, optional*) – Whether to flip the image channel dimension. Defaults to False.

forward(*lq*)

Forward function.

Parameters **lq** (*Tensor*) – Input LR image with shape (n, c, h, w).

Returns Output HR image.

Return type Tensor

class mmedit.models.editors.**GLDecoder**(*in_channels=256*, *norm_cfg=None*, *act_cfg=dict(type='ReLU')*, *out_act='clip'*)

Bases: mmengine.model.BaseModule

Decoder used in Global&Local model.

This implementation follows: Globally and locally Consistent Image Completion

Parameters

- **in_channels** (*int*) – Channel number of input feature.
- **norm_cfg** (*dict*) – Config dict to build norm layer.
- **act_cfg** (*dict*) – Config dict for activation layer, "relu" by default.
- **out_act** (*str*) – Output activation type, "clip" by default. Noted that in our implementation, we clip the output with range [-1, 1].

forward(*x*)

Forward Function.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**GLDilationNeck**(*in_channels=256*, *conv_type='conv'*, *norm_cfg=None*, *act_cfg=dict(type='ReLU')*, ***kwargs*)

Bases: mmengine.model.BaseModule

Dilation Backbone used in Global&Local model.

This implementation follows: Globally and locally Consistent Image Completion

Parameters

- **in_channels** (*int*) – Channel number of input feature.
- **conv_type** (*str*) – The type of conv module. In DeepFillv1 model, the *conv_type* should be 'conv'. In DeepFillv2 model, the *conv_type* should be 'gated_conv'.
- **norm_cfg** (*dict*) – Config dict to build norm layer.
- **act_cfg** (*dict*) – Config dict for activation layer, "relu" by default.
- **kwargs** (*keyword arguments*) –

_conv_type

forward(*x*)

Forward Function.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**GLEncoder**(*norm_cfg=None*, *act_cfg=dict(type='ReLU')*)

Bases: mmengine.model.BaseModule

Encoder used in Global&Local model.

This implementation follows: Globally and locally Consistent Image Completion

Parameters

- **norm_cfg** (*dict*) – Config dict to build norm layer.
- **act_cfg** (*dict*) – Config dict for activation layer, "relu" by default.

forward(*x*)

Forward Function.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**GLEncoderDecoder**(*encoder=dict(type='GLEncoder')*, *decoder=dict(type='GLDecoder')*, *dilation_neck=dict(type='GLDilationNeck')*)

Bases: mmengine.model.BaseModule

Encoder-Decoder used in Global&Local model.

This implementation follows: Globally and locally Consistent Image Completion

The architecture of the encoder-decoder is: (conv2d x 6) –> (dilated conv2d x 4) –> (conv2d or deconv2d x 7)

Parameters

- **encoder** (*dict*) – Config dict to encoder.
- **decoder** (*dict*) – Config dict to build decoder.
- **dilation_neck** (*dict*) – Config dict to build dilation neck.

forward(*x*)

Forward Function.

Parameters **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**AblatedDiffusionModel**(*data_preprocessor*, *unet*, *diffusion_scheduler*, *use_fp16=False*, *classifier=None*, *classifier_scale=1.0*, *rgb2bgr=False*, *pretrained_cfgs=None*)

Bases: mmengine.model.BaseModel

Guided diffusion Model.

Parameters

- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.
- **unet** (*ModelType*) – Config of denoising Unet.
- **diffusion_scheduler** (*ModelType*) – Config of diffusion_scheduler scheduler.
- **use_fp16** (*bool*) – Whether to use fp16 for unet model. Defaults to False.
- **classifier** (*ModelType*) – Config of classifier. Defaults to None.
- **pretrained_cfgs** (*dict*) – Path Config for pretrained weights. Usually this is a dict contains module name and the corresponding ckpt path.Defaults to None.

property device

Get current device of the model.

Returns The current device of the model.

Return type torch.device

load_pretrained_models(*pretrained_cfgs*)

summary

Parameters **pretrained_cfgs** (*_type_*) – _description_

infer(*scheduler_kwargs=None*, *init_image=None*, *batch_size=1*, *num_inference_steps=1000*, *labels=None*, *classifier_scale=0.0*, *show_progress=False*)

summary

Parameters

- **init_image** (*_type_, optional*) – _description_. Defaults to None.
- **batch_size** (*int, optional*) – _description_. Defaults to 1.
- **num_inference_steps** (*int, optional*) – _description_. Defaults to 1000.
- **labels** (*_type_, optional*) – _description_. Defaults to None.
- **show_progress** (*bool, optional*) – _description_. Defaults to False.

Returns _description_

Return type _type_

forward(*inputs: mmedit.utils.typing.ForwardInputs*, *data_samples: Optional[list] = None*, *mode: Optional[str] = None*) → List[*mmedit.structures.EditDataSample*]

summary

Parameters

- **inputs** (*ForwardInputs*) – _description_
- **data_samples** (*Optional[list], optional*) – _description_. Defaults to None.
- **mode** (*Optional[str], optional*) – _description_. Defaults to None.

Returns _description_

Return type List[EditDataSample]

val_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data.

Calls `self.data_preprocessor(data)` and `self(inputs, data_sample, mode=None)` in order. Return the generated results which will be passed to evaluator.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type SampleList

test_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type List[EditDataSample]

train_step(*data: dict*, *optim_wrapper: mmengine.optim.OptimWrapperDict*)

summary

Parameters

- **data** (*dict*) – _description_
- **optim_wrapper** (*OptimWrapperDict*) – _description_

Returns _description_

Return type _type_

`get_module`(*model: torch.nn.Module*, *module_name: str*) → torch.nn.Module

Get an inner module from model.

Since we will wrapper DDP for some model, we have to judge whether the module can be indexed directly.

Parameters

- **model** (*nn.Module*) – This model may wrapped with DDP or not.
- **module_name** (*str*) – The name of specific module.

Returns Returned sub module.

Return type nn.Module

`class mmedit.models.editors.IconVSRNet`(*mid_channels=64*, *num_blocks=30*, *keyframe_stride=5*, *padding=2*, *spynet_pretrained=None*, *edvr_pretrained=None*)

Bases: `mmengine.model.BaseModule`

IconVSR network structure for video super-resolution.

Support only x4 upsampling.

Paper: BasicVSR: The Search for Essential Components in Video Super-Resolution and Beyond, CVPR, 2021

Parameters

- **mid_channels** (*int*) – Channel number of the intermediate features. Default: 64.
- **num_blocks** (*int*) – Number of residual blocks in each propagation branch. Default: 30.
- **keyframe_stride** (*int*) – Number determining the keyframes. If stride=5, then the (0, 5, 10, 15, ...)-th frame will be the keyframes. Default: 5.
- **padding** (*int*) – Number of frames to be padded at two ends of the sequence. 2 for REDS and 3 for Vimeo-90K. Default: 2.
- **spynet_pretrained** (*str*) – Pre-trained model path of SPyNet. Default: None.
- **edvr_pretrained** (*str*) – Pre-trained model path of EDVR (for refill). Default: None.

`spatial_padding`(*lrs*)

Apply pdding spatially.

Since the PCD module in EDVR requires that the resolution is a multiple of 4, we apply padding to the input LR images if their resolution is not divisible by 4.

Parameters **lrs** (*Tensor*) – Input LR sequence with shape (n, t, c, h, w).

Returns Padded LR sequence with shape (n, t, c, h_pad, w_pad).

Return type Tensor

`check_if_mirror_extended`(*lrs*)

Check whether the input is a mirror-extended sequence.

If mirror-extended, the i-th (i=0, ..., t-1) frame is equal to the (t-1-i)-th frame.

Parameters **lrs** (*tensor*) – Input LR images with shape (n, t, c, h, w)

compute_refill_features(*lrs*, *keyframe_idx*)

Compute keyframe features for information-refill.

Since EDVR-M is used, padding is performed before feature computation. :param lrs: Input LR images with shape (n, t, c, h, w) :type lrs: Tensor :param keyframe_idx: The indices specifying the keyframes. :type keyframe_idx: list(int)

Returns

The keyframe features. Each key corresponds to the indices in keyframe_idx.

Return type dict(Tensor)

compute_flow(*lrs*)

Compute optical flow using SPyNet for feature warping.

Note that if the input is an mirror-extended sequence, 'flows_forward' is not needed, since it is equal to 'flows_backward.flip(1)'.

Parameters **lrs** (*tensor*) – Input LR images with shape (n, t, c, h, w)

Returns

Optical flow. 'flows_forward' corresponds to the flows used for forward-time propagation (current to previous). 'flows_backward' corresponds to the flows used for backward-time propagation (current to next).

Return type tuple(Tensor)

forward(*lrs*)

Forward function for IconVSR.

Parameters **lrs** (*Tensor*) – Input LR tensor with shape (n, t, c, h, w).

Returns Output HR tensor with shape (n, t, c, 4h, 4w).

Return type Tensor

class mmedit.models.editors.**DepthwiseIndexBlock**(*in_channels*, *norm_cfg=dict(type='BN')*, *use_context=False*, *use_nonlinear=False*, *mode='o2o'*)

Bases: mmengine.model.BaseModule

Depthwise index block.

From https://arxiv.org/abs/1908.00672.

Parameters

- **in_channels** (*int*) – Input channels of the holistic index block.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: dict(type='BN').
- **use_context** (*bool, optional*) – Whether use larger kernel size in index block. Refer to the paper for more information. Defaults to False.
- **use_nonlinear** (*bool*) – Whether add a non-linear conv layer in the index blocks. Default: False.
- **mode** (*str*) – Mode of index block. Should be 'o2o' or 'm2o'. In 'o2o' mode, the group of the conv layers is 1; In 'm2o' mode, the group of the conv layer is *in_channels*.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input feature map with shape (N, C, H, W).

Returns Encoder index feature and decoder index feature.

Return type tuple(Tensor)

class mmedit.models.editors.**HolisticIndexBlock**(*in_channels*, *norm_cfg=dict(type='BN')*, *use_context=False*, *use_nonlinear=False*)

Bases: mmengine.model.BaseModule

Holistic Index Block.

From https://arxiv.org/abs/1908.00672.

Parameters

- **in_channels** (*int*) – Input channels of the holistic index block.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: dict(type='BN').
- **use_context** (*bool, optional*) – Whether use larger kernel size in index block. Refer to the paper for more information. Defaults to False.
- **use_nonlinear** (*bool*) – Whether add a non-linear conv layer in the index block. Default: False.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input feature map with shape (N, C, H, W).

Returns Encoder index feature and decoder index feature.

Return type tuple(Tensor)

class mmedit.models.editors.**IndexedUpsample**(*in_channels*, *out_channels*, *kernel_size=5*, *norm_cfg=dict(type='BN')*, *conv_module=ConvModule*, *init_cfg: Optional[dict] = None*)

Bases: mmengine.model.BaseModule

Indexed upsample module.

Parameters

- **in_channels** (*int*) – Input channels.
- **out_channels** (*int*) – Output channels.
- **kernel_size** (*int, optional*) – Kernel size of the convolution layer. Defaults to 5.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Defaults to dict(type='BN').
- **conv_module** (*ConvModule | DepthwiseSeparableConvModule, optional*) – Conv module. Defaults to ConvModule.
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.

init_weights()

Init weights for the module.

forward(*x*, *shortcut*, *dec_idx_feat=None*)

Forward function.

Parameters

- **x** (*Tensor*) – Input feature map with shape (N, C, H, W).
- **shortcut** (*Tensor*) – The shortcut connection with shape (N, C, H', W').
- **dec_idx_feat** (*Tensor, optional*) – The decode index feature map with shape (N, C, H', W'). Defaults to None.

Returns Output tensor with shape (N, C, H', W').

Return type Tensor

class mmedit.models.editors.**IndexNet**(*data_preprocessor*, *backbone*, *loss_alpha=None*, *loss_comp=None*, *init_cfg=None*, *train_cfg=None*, *test_cfg=None*)

Bases: *mmedit.models.base_models.BaseMattor*

IndexNet matting model.

This implementation follows: Indices Matter: Learning to Index for Deep Image Matting

Parameters

- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.
- **backbone** (*dict*) – Config of backbone.
- **train_cfg** (*dict*) – Config of training. In 'train_cfg', 'train_backbone' should be specified.
- **test_cfg** (*dict*) – Config of testing.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule.
- **loss_alpha** (*dict*) – Config of the alpha prediction loss. Default: None.
- **loss_comp** (*dict*) – Config of the composition loss. Default: None.

_forward(*inputs*)

Forward function.

Parameters **inputs** (*torch.Tensor*) – Input tensor.

Returns Output tensor.

Return type Tensor

_forward_test(*inputs*)

Forward function for testing IndexNet model.

Parameters **inputs** (*torch.Tensor*) – batch input tensor.

Returns Output tensor of model.

Return type Tensor

_forward_train(*inputs*, *data_samples*)

Forward function for training IndexNet model.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.

- **data_samples** (*List[BaseDataElement]*) – data samples collated by data_preprocessor.

Returns Contains the loss items and batch information.

Return type dict

class mmedit.models.editors.**IndexNetDecoder**(*in_channels*, *kernel_size=5*, *norm_cfg=dict(type='BN')*, *separable_conv=False*, *init_cfg: Optional[dict] = None*)

Bases: mmengine.model.BaseModule

Decoder for IndexNet.

Please refer to https://arxiv.org/abs/1908.00672.

Parameters

- **in_channels** (*int*) – Input channels of the decoder.
- **kernel_size** (*int, optional*) – Kernel size of the convolution layer. Defaults to 5.
- **norm_cfg** (*None | dict, optional*) – Config dict for normalization layer. Defaults to dict(type='BN').
- **separable_conv** (*bool*) – Whether to use separable conv. Default: False.
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.

init_weights()

Init weights for the module.

forward(*inputs*)

Forward function.

Parameters **inputs** (*dict*) – Output dict of IndexNetEncoder.

Returns Predicted alpha matte of the current batch.

Return type Tensor

class mmedit.models.editors.**IndexNetEncoder**(*in_channels*, *out_stride=32*, *width_mult=1*, *index_mode='m2o'*, *aspp=True*, *norm_cfg=dict(type='BN')*, *freeze_bn=False*, *use_nonlinear=True*, *use_context=True*, *init_cfg: Optional[dict] = None*)

Bases: mmengine.model.BaseModule

Encoder for IndexNet.

Please refer to https://arxiv.org/abs/1908.00672.

Parameters

- **in_channels** (*int, optional*) – Input channels of the encoder.
- **out_stride** (*int, optional*) – Output stride of the encoder. For example, if *out_stride* is 32, the input feature map or image will be downsample to the 1/32 of original size. Defaults to 32.
- **width_mult** (*int, optional*) – Width multiplication factor of channel dimension in MobileNetV2. Defaults to 1.
- **index_mode** (*str, optional*) – Index mode of the index network. It must be one of {'holistic', 'o2o', 'm2o'}. If it is set to 'holistic', then Holistic index network will be used

as the index network. If it is set to 'o2o' (or 'm2o'), when O2O (or M2O) Depthwise index network will be used as the index network. Defaults to 'm2o'.

- **aspp** (*bool, optional*) – Whether use ASPP module to augment output feature. Defaults to True.
- **norm_cfg** (*None | dict, optional*) – Config dict for normalization layer. Defaults to dict(type='BN').
- **freeze_bn** (*bool, optional*) – Whether freeze batch norm layer. Defaults to False.
- **use_nonlinear** (*bool, optional*) – Whether use nonlinearty in index network. Refer to the paper for more information. Defaults to True.
- **use_context** (*bool, optional*) – Whether use larger kernel size in index network. Refer to the paper for more information. Defaults to True.
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.

Raises

- **ValueError** – out_stride must 16 or 32.
- **NameError** – Supported index_mode are {'holistic', 'o2o', 'm2o'}.

_make_layer(*layer_setting*, *norm_cfg*)

train(*mode=True*)

Set BatchNorm modules in the model to evaluation mode.

init_weights()

Init weights for the model.

Initialization is based on self._init_cfg

Parameters **pretrained** (*str, optional*) – Path for pretrained weights. If given None, pretrained weights will not be loaded. Defaults to None.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input feature map with shape (N, C, H, W).

Returns Output tensor, shortcut feature and decoder index feature.

Return type dict

class mmedit.models.editors.**InstColorization**(*data_preprocessor: Union[dict, mmengine.config.Config]*, *image_model*, *instance_model*, *fusion_model*, *color_data_opt*, *which_direction='AtoB'*, *loss=None*, *init_cfg=None*, *train_cfg=None*, *test_cfg=None*)

Bases: mmengine.model.BaseModel

Colorization InstColorization method.

This Colorization is implemented according to the paper: Instance-aware Image Colorization, CVPR 2020

Adapted from 'https://github.com/ericsujw/InstColorization.git' 'InstColorization/models/train_model' Copyright (c) 2020, Su, under MIT License.

Parameters

- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.

- **image_model** (*dict*) – Config for single image model
- **instance_model** (*dict*) – Config for instance model
- **fusion_model** (*dict*) – Config for fusion model
- **color_data_opt** (*dict*) – Option for colorspace conversion
- **which_direction** (*str*) – AtoB or BtoA
- **loss** (*dict*) – Config for loss.
- **init_cfg** (*str*) – Initialization config dict. Default: None.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.

forward(*inputs: torch.Tensor*, *data_samples: Optional[List[*mmedit.structures.EditDataSample*]] = None*, *mode: str = 'tensor'*, ***kwargs*)

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

`forward` method of BaseModel is an abstract method, its subclasses must implement this method.

Accepts `inputs` and `data_samples` processed by `data_preprocessor`, and returns results according to mode arguments.

During non-distributed training, validation, and testing process, `forward` will be called by `BaseModel.train_step`, `BaseModel.val_step` and `BaseModel.val_step` directly.

During distributed data parallel training process, `MMSeparateDistributedDataParallel.train_step` will first call `DistributedDataParallel.forward` to enable automatic gradient synchronization, and then call `forward` to get training loss.

Parameters

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

Returns

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

Return type ForwardResults

convert_to_datasample(*inputs*, *data_samples*)

abstract forward_train(*inputs*, *data_samples=None*, ***kwargs*)

Forward function for training.

abstract train_step(*data: List[dict]*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train step function.

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns

Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

forward_inference(*inputs*, *data_samples=None*, ***kwargs*)

Forward inference. Returns predictions of validation, testing.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by `data_preprocessor`.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by `data_preprocessor`.

Returns predictions.

Return type List[EditDataSample]

forward_tensor(*inputs*, *data_samples*)

Forward function in tensor mode.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_sample** (*dict*) – Dict contains data sample.

Returns Dict contains output results.

Return type dict

class mmedit.models.editors.**LIIF**(*generator: dict*, *pixel_loss: dict*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = None*, *data_preprocessor: Optional[dict] = None*)

Bases: *mmedit.models.base_models.BaseEditModel*

LIIF model for single image super-resolution.

Paper: Learning Continuous Image Representation with Local Implicit Image Function

Parameters

- **generator** (*dict*) – Config for the generator.
- **pixel_loss** (*dict*) – Config for the pixel loss.
- **pretrained** (*str*) – Path for pretrained model. Default: None.
- **data_preprocessor** (*dict, optional*) – The pre-process config of `BaseDataPreprocessor`.

forward_tensor(*inputs*, *data_samples=None*, ***kwargs*)

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by `data_preprocessor`.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by `data_preprocessor`.

Returns result of simple forward.

Return type Tensor

forward_inference(*inputs*, *data_samples=None*, ***kwargs*)

Forward inference. Returns predictions of validation, testing, and simple inference.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by `data_preprocessor`.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by `data_preprocessor`.

Returns predictions.

Return type List[EditDataSample]

class mmedit.models.editors.**MLPRefiner**(*in_dim*, *out_dim*, *hidden_list*)

Bases: `mmengine.model.BaseModule`

Multilayer perceptrons (MLPs), refiner used in LIIF.

Parameters

- **in_dim** (*int*) – Input dimension.
- **out_dim** (*int*) – Output dimension.
- **hidden_list** (*list[int]*) – List of hidden dimensions.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – The input of MLP.

Returns The output of MLP.

Return type Tensor

class mmedit.models.editors.**LSGAN**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = None*, *ema_config: Optional[Dict] = None*, *loss_config: Optional[Dict] = None*)

Bases: *mmedit.models.base_models.BaseGAN*

Impelmentation of *Least Squares Generative Adversarial Networks*.

Paper link: https://arxiv.org/pdf/1611.04076.pdf

Detailed architecture can be found in `LSGANGenerator` and `LSGANDiscriminator`

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*) → Tuple

Get disc loss. LSGAN use the least squares loss to train the discriminator.

$$L_D = \left(D\left(X_{\text{data}}\right) - 1\right)^2 + \left(D(G(z))\right)^2$$

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

gen_loss(*disc_pred_fake: torch.Tensor*) → Tuple

Get gen loss. LSGAN use the least squares loss to train the generator.

$$L_G = \left(D(G(z)) - 1\right)^2$$

Parameters **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.editors.**MSPIEStyleGAN2**(**args*, *train_settings=dict()*, ***kwargs*)

Bases: mmedit.models.editors.stylegan2.StyleGAN2

MS-PIE StyleGAN2.

In this GAN, we adopt the MS-PIE training schedule so that multi-scale images can be generated with a single generator. Details can be found in: Positional Encoding as Spatial Inductive Bias in GANs, CVPR2021.

Parameters **train_settings** (*dict*) – Config for training settings. Defaults to *dict()*.

train_step(*data: dict*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train GAN model. In the training of GAN models, generator and discriminator are updated alternatively. In MMGeneration's design, *self.train_step* is called with data input. Therefore we always update discriminator, whose updating is relay on real data, and then determine if the generator needs to be updated based on the current number of iterations. More details about whether to update generator can be found in should_gen_update().

Parameters

- **data** (*dict*) – Data sampled from dataloader.
- **optim_wrapper** (*OptimWrapperDict*) – OptimWrapperDict instance contains OptimWrapper of generator and discriminator.

Returns A dict of tensor for logging.

Return type Dict[str, torch.Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*TrainInput*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A dict of tensor for logging.

Return type Dict[str, Tensor]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*TrainInput*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A dict of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.editors.**PESinGAN**(*generator: ModelType*, *discriminator: Optional[ModelType]*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *num_scales: Optional[int] = None*, *fixed_noise_with_pad: bool = False*, *first_fixed_noises_ch: int = 1*, *iters_per_scale: int = 200*, *noise_weight_init: int = 0.1*, *lr_scheduler_args: Optional[dict] = None*, *test_pkl_data: Optional[str] = None*, *ema_confg: Optional[dict] = None*)

Bases: mmedit.models.editors.singan.SinGAN

Positional Encoding in SinGAN.

This modified SinGAN is used to reimplement the experiments in: Positional Encoding as Spatial Inductive Bias in GANs, CVPR2021.

construct_fixed_noises()

Construct the fixed noises list used in SinGAN.

class mmedit.models.editors.**NAFBaseline**(*img_channel=3*, *mid_channels=16*, *middle_blk_num=1*, *enc_blk_nums=[1, 1, 1, 28]*, *dec_blk_nums=[1, 1, 1, 1]*, *dw_expand=1*, *ffn_expand=2*)

Bases: mmengine.model.BaseModule

The original version of Baseline model in "Simple Baseline for Image Restoration".

Parameters

- **img_channels** (*int*) – Channel number of inputs.
- **mid_channels** (*int*) – Channel number of intermediate features.
- **middle_blk_num** (*int*) – Number of middle blocks.
- **enc_blk_nums** (*List of int*) – Number of blocks for each encoder.
- **dec_blk_nums** (*List of int*) – Number of blocks for each decoder.

forward(*inp*)

Forward function.

Parameters **inp** – input tensor image with (B, C, H, W) shape

check_image_size(*x*)

Check image size and pad images so that it has enough dimension do downsample.

Parameters **x** – input tensor image with (B, C, H, W) shape.

class mmedit.models.editors.**NAFBaselineLocal**(**args*, *train_size=(1, 3, 256, 256)*, *fast_imp=False*, ***kwargs*)

Bases: mmedit.models.editors.nafnet.naf_avgpool2d.Local_Base, *NAFBaseline*

The original version of Baseline model in "Simple Baseline for Image Restoration".

Parameters

- **img_channels** (*int*) – Channel number of inputs.
- **mid_channels** (*int*) – Channel number of intermediate features.
- **middle_blk_num** (*int*) – Number of middle blocks.
- **enc_blk_nums** (*List of int*) – Number of blocks for each encoder.
- **dec_blk_nums** (*L`ist of int*) – Number of blocks for each decoder.

class mmedit.models.editors.**NAFNet**(*img_channel=3*, *mid_channels=16*, *middle_blk_num=1*, *enc_blk_nums=[]*, *dec_blk_nums=[]*)

Bases: mmengine.model.BaseModule

NAFNet.

The original version of NAFNet in "Simple Baseline for Image Restoration".

Parameters

- **img_channels** (*int*) – Channel number of inputs.
- **mid_channels** (*int*) – Channel number of intermediate features.
- **middle_blk_num** (*int*) – Number of middle blocks.
- **enc_blk_nums** (*List of int*) – Number of blocks for each encoder.
- **dec_blk_nums** (*List of int*) – Number of blocks for each decoder.

forward(*inp*)

Forward function.

Parameters **inp** – input tensor image with (B, C, H, W) shape

check_image_size(*x*)

Check image size and pad images so that it has enough dimension do downsample.

Parameters **x** – input tensor image with (B, C, H, W) shape.

class mmedit.models.editors.**NAFNetLocal**(**args*, *train_size=(1, 3, 256, 256)*, *fast_imp=False*, ***kwargs*)

Bases: mmedit.models.editors.nafnet.naf_avgpool2d.Local_Base, *NAFNet*

The original version of NAFNetLocal in "Simple Baseline for Image Restoration".

NAFNetLocal uses local average pooling modules than NAFNet.

Parameters

- **img_channels** (*int*) – Channel number of inputs.
- **mid_channels** (*int*) – Channel number of intermediate features.
- **middle_blk_num** (*int*) – Number of middle blocks.
- **enc_blk_nums** (*List of int*) – Number of blocks for each encoder.
- **dec_blk_nums** (*List of int*) – Number of blocks for each decoder.

class mmedit.models.editors.**MaskConvModule**(**args*, ***kwargs*)

Bases: mmcv.cnn.ConvModule

Mask convolution module.

This is a simple wrapper for mask convolution like: 'partial conv'. Convolutions in this module always need a mask as extra input.

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.
- **conv_cfg** (*dict*) – Config dict for convolution layer.
- **norm_cfg** (*dict*) – Config dict for normalization layer.
- **act_cfg** (*dict*) – Config dict for activation layer, "relu" by default.
- **inplace** (*bool*) – Whether to use inplace mode for activation.
- **with_spectral_norm** (*bool*) – Whether use spectral norm in conv module.
- **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". Examples are ("conv", "norm", "act") and ("act", "conv", "norm").

supported_conv_list = ['PConv']

forward(*x*, *mask=None*, *activate=True*, *norm=True*, *return_mask=True*)

Forward function for partial conv2d.

Parameters

- **x** (*torch.Tensor*) – Tensor with shape of (n, c, h, w).
- **mask** (*torch.Tensor*) – Tensor with shape of (n, c, h, w) or (n, 1, h, w). If mask is not given, the function will work as standard conv2d. Default: None.
- **activate** (*bool*) – Whether use activation layer.
- **norm** (*bool*) – Whether use norm layer.
- **return_mask** (*bool*) – If True and mask is not None, the updated mask will be returned. Default: True.

Returns

Result Tensor or 2-tuple of

Tensor: Results after partial conv.

Tensor: Updated mask will be returned if mask is given and *return_mask* is True.

Return type Tensor or tuple

class mmedit.models.editors.**PartialConv2d**(**args*, *multi_channel=False*, *eps=1e-08*, ***kwargs*)

Bases: torch.nn.Conv2d

Implementation for partial convolution.

Image Inpainting for Irregular Holes Using Partial Convolutions [https://arxiv.org/abs/1804.07723]

Parameters

- **multi_channel** (*bool*) – If True, the mask is multi-channel. Otherwise, the mask is single-channel.

- **eps** (*float*) – Need to be changed for mixed precision training. For mixed precision training, you need change 1e-8 to 1e-6.

forward(*input*, *mask=None*, *return_mask=True*)

Forward function for partial conv2d.

Parameters

- **input** (*torch.Tensor*) – Tensor with shape of (n, c, h, w).
- **mask** (*torch.Tensor*) – Tensor with shape of (n, c, h, w) or (n, 1, h, w). If mask is not given, the function will work as standard conv2d. Default: None.
- **return_mask** (*bool*) – If True and mask is not None, the updated mask will be returned. Default: True.

Returns Results after partial conv. torch.Tensor : Updated mask will be returned if mask is given and `return_mask` is True.

Return type torch.Tensor

class mmedit.models.editors.**PConvDecoder**(*num_layers=7*, *interpolation='nearest'*, *conv_cfg=dict(type='PConv', multi_channel=True)*, *norm_cfg=dict(type='BN')*)

Bases: `mmengine.model.BaseModule`

Decoder with partial conv.

About the details for this architecture, pls see: Image Inpainting for Irregular Holes Using Partial Convolutions

Parameters

- **num_layers** (*int*) – The number of convolutional layers. Default: 7.
- **interpolation** (*str*) – The upsample mode. Default: 'nearest'.
- **conv_cfg** (*dict*) – Config for convolution module. Default: {'type': 'PConv', 'multi_channel': True}.
- **norm_cfg** (*dict*) – Config for norm layer. Default: {'type': 'BN'}.

forward(*input_dict*)

Forward Function.

Parameters **input_dict** (*dict | torch.Tensor*) – Input dict with middle features or torch.Tensor.

Returns Output tensor with shape of (n, c, h, w).

Return type torch.Tensor

class mmedit.models.editors.**PConvEncoder**(*in_channels=3*, *num_layers=7*, *conv_cfg=dict(type='PConv', multi_channel=True)*, *norm_cfg=dict(type='BN', requires_grad=True)*, *norm_eval=False*)

Bases: `mmengine.model.BaseModule`

Encoder with partial conv.

About the details for this architecture, pls see: Image Inpainting for Irregular Holes Using Partial Convolutions

Parameters

- **in_channels** (*int*) – The number of input channels. Default: 3.
- **num_layers** (*int*) – The number of convolutional layers. Default: 7.

- **conv_cfg** (*dict*) – Config for convolution module. Default: {'type': 'PConv', 'multi_channel': True}.
- **norm_cfg** (*dict*) – Config for norm layer. Default: {'type': 'BN'}.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effective on Batch Norm and its variants only. Default: False.

train(*mode=True*)

Set BatchNorm modules in the model to evaluation mode.

forward(*x*, *mask*)

Forward function for partial conv encoder.

Parameters

- **x** (*torch.Tensor*) – Masked image with shape (n, c, h, w).
- **mask** (*torch.Tensor*) – Mask tensor with shape (n, c, h, w).

Returns Contains the results and middle level features in this module. *hidden_feats* contain the middle feature maps and *hidden_masks* store updated masks.

Return type dict

class mmedit.models.editors.**PConvEncoderDecoder**(*encoder*, *decoder*)

Bases: mmengine.model.BaseModule

Encoder-Decoder with partial conv module.

Parameters

- **encoder** (*dict*) – Config of the encoder.
- **decoder** (*dict*) – Config of the decoder.

forward(*x*, *mask_in*)

Forward Function.

Parameters

- **x** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).
- **mask_in** (*torch.Tensor*) – Input tensor with shape of (n, c, h, w).

Returns Output tensor with shape of (n, c, h', w').

Return type torch.Tensor

class mmedit.models.editors.**PConvInpaintor**(*data_preprocessor: Union[dict, mmengine.config.Config]*, *encdec: dict*, *disc: Optional[dict] = None*, *loss_gan: Optional[dict] = None*, *loss_gp: Optional[dict] = None*, *loss_disc_shift: Optional[dict] = None*, *loss_composed_percep: Optional[dict] = None*, *loss_out_percep: bool = False*, *loss_l1_hole: Optional[dict] = None*, *loss_l1_valid: Optional[dict] = None*, *loss_tv: Optional[dict] = None*, *train_cfg: Optional[dict] = None*, *test_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = None*)

Bases: *mmedit.models.base_models.OneStageInpaintor*

Inpaintor for Partial Convolution method.

This inpaintor is implemented according to the paper: Image inpainting for irregular holes using partial convolutions

forward_test(*inputs*, *data_samples*)

Forward function for testing.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_samples** (*List[dict]*) – List of data sample dict.

Returns Contain output results and eval metrics (if have).

Return type dict

forward_tensor(*inputs*, *data_samples*)

Forward function in tensor mode.

Parameters

- **inputs** (*torch.Tensor*) – Input tensor.
- **data_sample** (*dict*) – Dict contains data sample.

Returns Dict contains output results.

Return type dict

train_step(*data: List[dict]*, *optim_wrapper*)

Train step function.

In this function, the inpaintor will finish the train step following the pipeline:

1. get fake res/image
2. optimize discriminator (if have)
3. optimize generator

If *self.train_cfg.disc_step > 1*, the train step will contain multiple iterations for optimizing discriminator with different input data and only one iteration for optimizing gerator after *disc_step* iterations for discriminator.

Parameters

- **data** (*List[dict]*) – Batch of data as input.
- **optim_wrapper** (*dict[torch.optim.Optimizer]*) – Dict with optimizers for generator and discriminator (if have).

Returns Dict with loss, information for logger, the number of samples and results for visualization.

Return type dict

class mmedit.models.editors.**ProgressiveGrowingGAN**(*generator*, *discriminator*, *data_preprocessor*, *nkimgs_per_scale*, *noise_size=None*, *interp_real=None*, *transition_kimgs: int = 600*, *prev_stage: int = 0*, *ema_config: Optional[Dict] = None*)

Bases: *mmedit.models.base_models.BaseGAN*

Progressive Growing Unconditional GAN.

In this GAN model, we implement progressive growing training schedule, which is proposed in Progressive Growing of GANs for improved Quality, Stability and Variation, ICLR 2018.

We highly recommend to use `GrowScaleImgDataset` for saving computational load in data pre-processing.

Notes for **using PGGAN**:

1. In official implementation, Tero uses gradient penalty with `norm_mode="HWC"`
2. We do not implement `minibatch_repeats` where has been used in official Tensorflow implementation.

Notes for resuming progressive growing GANs: Users should specify the `prev_stage` in `train_cfg`. Otherwise, the model is possible to reset the optimizer status, which will bring inferior performance. For example, if your model is resumed from the *256* stage, you should set `train_cfg=dict(prev_stage=256)`.

Parameters

- **generator** (*dict*) – Config for generator.
- **discriminator** (*dict*) – Config for discriminator.

forward(*inputs: mmedit.utils.typing.ForwardInputs*, *data_samples: Optional[list] = None*, *mode: Optional[str] = None*) → mmedit.utils.typing.SampleList

Sample images from noises by using the generator.

Parameters

- **batch_inputs** (*ForwardInputs*) – Dict containing the necessary information (e.g. noise, num_batches, mode) to generate image.
- **data_samples** (*Optional[list]*) – Data samples collated by `data_preprocessor`. Defaults to None.
- **mode** (*Optional[str]*) – *mode* is not used in *ProgressiveGrowingGAN*. Defaults to None.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

train_discriminator(*inputs: torch.Tensor*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*, *fake_data: torch.Tensor*, *real_data: torch.Tensor*) → Tuple[torch.Tensor, dict]

Get disc loss. PGGAN use WGAN-GP's loss and discriminator shift loss to train the discriminator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.
- **fake_data** (*Tensor*) – Generated images, used to calculate gradient penalty.
- **real_data** (*Tensor*) – Real images, used to calculate gradient penalty.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

train_generator(*inputs: torch.Tensor*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

gen_loss(*disc_pred_fake: torch.Tensor*) → Tuple[torch.Tensor, dict]

Generator loss for PGGAN. PGGAN use WGAN's loss to train the generator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **recon_imgs** (*Tensor*) – Reconstructive images.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

train_step(*data: dict*, *optim_wrapper: mmengine.optim.OptimWrapperDict*)

Train step function.

This function implements the standard training iteration for asynchronous adversarial training. Namely, in each iteration, we first update discriminator and then compute loss for generator with the newly updated discriminator.

As for distributed training, we use the `reducer` from ddp to synchronize the necessary params in current computational graph.

Parameters

- **data_batch** (*dict*) – Input data from dataloader.
- **optimizer** (*dict*) – Dict contains optimizer for generator and discriminator.
- **ddp_reducer** (`Reducer` | None, optional) – Reducer from ddp. It is used to prepare for `backward()` in ddp. Defaults to None.
- **running_status** (*dict | None, optional*) – Contains necessary basic information for training, e.g., iteration number. Defaults to None.

Returns Contains 'log_vars', 'num_samples', and 'results'.

Return type dict

class mmedit.models.editors.**Pix2Pix**(**args*, ***kwargs*)

Bases: *mmedit.models.base_models.BaseTranslationModel*

Pix2Pix model for paired image-to-image translation.

Ref: Image-to-Image Translation with Conditional Adversarial Networks

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

Forward function for testing.

Parameters

- **img** (*tensor*) – Input image tensor.
- **target_domain** (*str*) – Target domain of output image.
- **kwargs** (*dict*) – Other arguments.

Returns Forward results.

Return type dict

_get_disc_loss(*outputs*)

Get the loss of discriminator.

Parameters **outputs** (*dict*) – A dict of output.

Returns Loss and a dict of log of loss terms.

Return type Tuple

_get_gen_loss(*outputs*)

Get the loss of generator.

Parameters **outputs** (*dict*) – A dict of output.

Returns Loss and a dict of log of loss terms.

Return type Tuple

train_step(*data*, *optim_wrapper=None*)

Training step function.

Parameters

- **data_batch** (*dict*) – Dict of the input data batch.
- **optimizer** (*dict[torch.optim.Optimizer]*) – Dict of optimizers for the generator and discriminator.
- **ddp_reducer** (Reducer | None, optional) – Reducer from ddp. It is used to prepare for `backward()` in ddp. Defaults to None.
- **running_status** (*dict | None, optional*) – Contains necessary basic information for training, e.g., iteration number. Defaults to None.

Returns Dict of loss, information for logger, the number of samples and results for visualization.

Return type dict

test_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type List[EditDataSample]

val_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns Generated image or image dict.

Return type List[EditDataSample]

class mmedit.models.editors.**PlainDecoder**(*in_channels*, *init_cfg: Optional[dict] = None*)

Bases: mmengine.model.BaseModule

Simple decoder from Deep Image Matting.

Parameters

- **in_channels** (*int*) – Channel num of input features.
- **init_cfg** (*dict, optional*) – Initialization config dict. efaults to None.

init_weights()

Init weights for the module.

forward(*inputs*)

Forward function of PlainDecoder.

Parameters **inputs** (*dict*) – Output dictionary of the VGG encoder containing:

- out (Tensor): Output of the VGG encoder.
- max_idx_1 (Tensor): Index of the first maxpooling layer in the VGG encoder.
- max_idx_2 (Tensor): Index of the second maxpooling layer in the VGG encoder.
- max_idx_3 (Tensor): Index of the third maxpooling layer in the VGG encoder.
- max_idx_4 (Tensor): Index of the fourth maxpooling layer in the VGG encoder.
- max_idx_5 (Tensor): Index of the fifth maxpooling layer in the VGG encoder.

Returns Output tensor.

Return type Tensor

class mmedit.models.editors.**PlainRefiner**(*conv_channels=64*, *init_cfg=None*)

Bases: mmengine.model.BaseModule

Simple refiner from Deep Image Matting.

Parameters

- **conv_channels** (*int*) – Number of channels produced by the three main convolutional layer. Default: 64.
- **pretrained** (*str*) – Name of pretrained model. Default: None.

init_weights()

Init weights for the module.

forward(*x*, *raw_alpha*)

Forward function.

Parameters

- **x** (*Tensor*) – The input feature map of refiner.
- **raw_alpha** (*Tensor*) – The raw predicted alpha matte.

Returns The refined alpha matte.

Return type Tensor

class mmedit.models.editors.**RDNNet**(*in_channels*, *out_channels*, *mid_channels=64*, *num_blocks=16*, *upscale_factor=4*, *num_layers=8*, *channel_growth=64*)

Bases: mmengine.model.BaseModule

RDN model for single image super-resolution.

Paper: Residual Dense Network for Image Super-Resolution

Adapted from 'https://github.com/yjn870/RDN-pytorch.git' 'RDN-pytorch/blob/master/models.py' Copyright (c) 2021, JaeYun Yeo, under MIT License.

Most of the implementation follows the implementation in: 'https://github.com/sanghyun-son/EDSR-PyTorch.git' 'EDSR-PyTorch/blob/master/src/model/rdn.py' Copyright (c) 2017, sanghyun-son, under MIT license.

Parameters

- **in_channels** (*int*) – Channel number of inputs.
- **out_channels** (*int*) – Channel number of outputs.
- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64.
- **num_blocks** (*int*) – Block number in the trunk network. Default: 16.
- **upscale_factor** (*int*) – Upsampling factor. Support 2^n and 3. Default: 4.
- **num_layer** (*int*) – Layer number in the Residual Dense Block. Default: 8.
- **channel_growth** (*int*) – Channels growth in each layer of RDB. Default: 64.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**RealBasicVSR**(*generator*, *discriminator=None*, *gan_loss=None*, *pixel_loss=None*, *cleaning_loss=None*, *perceptual_loss=None*, *is_use_sharpened_gt_in_pixel=False*, *is_use_sharpened_gt_in_percep=False*, *is_use_sharpened_gt_in_gan=False*, *is_use_ema=False*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.editors.real_esrgan.RealESRGAN

RealBasicVSR model for real-world video super-resolution.

Ref: Investigating Tradeoffs in Real-World Video Super-Resolution, arXiv

Parameters

- **generator** (*dict*) – Config for the generator.
- **discriminator** (*dict, optional*) – Config for the discriminator. Default: None.
- **gan_loss** (*dict, optional*) – Config for the gan loss. Note that the loss weight in gan loss is only for the generator.
- **pixel_loss** (*dict, optional*) – Config for the pixel loss. Default: None.
- **cleaning_loss** (*dict, optional*) – Config for the image cleaning loss. Default: None.
- **perceptual_loss** (*dict, optional*) – Config for the perceptual loss. Default: None.
- **is_use_sharpened_gt_in_pixel** (*bool, optional*) – Whether to use the image sharpened by unsharp masking as the GT for pixel loss. Default: False.
- **is_use_sharpened_gt_in_percep** (*bool, optional*) – Whether to use the image sharpened by unsharp masking as the GT for perceptual loss. Default: False.
- **is_use_sharpened_gt_in_gan** (*bool, optional*) – Whether to use the image sharpened by unsharp masking as the GT for adversarial loss. Default: False.
- **train_cfg** (*dict*) – Config for training. Default: None. You may change the training of gan by setting: *disc_steps*: how many discriminator updates after one generate update; *disc_init_steps*: how many discriminator updates at the start of the training. These two keys are useful when training with WGAN.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for `BaseModule`. Default: None.
- **data_preprocessor** (*dict, optional*) – The pre-process config of `BaseDataPreprocessor`. Default: None.

extract_gt_data(*data_samples*)

extract gt data from data samples.

Parameters **data_samples** (*list*) – List of EditDataSample.

Returns Extract gt data.

Return type Tensor

g_step(*batch_outputs*, *batch_gt_data*)

G step of GAN: Calculate losses of generator.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tuple[Tensor]*) – Batch GT data.

Returns Dict of losses.

Return type dict

d_step_with_optim(*batch_outputs: torch.Tensor*, *batch_gt_data: torch.Tensor*, *optim_wrapper: mmengine.optim.OptimWrapperDict*)

D step with optim of GAN: Calculate losses of discriminator and run optim.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.

- **batch_gt_data** (*Tensor*) – Batch GT data.
- **optim_wrapper** (*OptimWrapperDict*) – Optim wrapper dict.

Returns Dict of parsed losses.

Return type dict

forward_train(*batch_inputs*, *data_samples=None*)

Forward Train.

Run forward of generator with `return_lqs=True`

Parameters

- **batch_inputs** (*Tensor*) – Batch inputs.
- **data_samples** (*List[EditDataSample]*) – Data samples of Editing. Default:None

Returns

Result of generator. (outputs, lqs)

Return type Tuple[Tensor]

class mmedit.models.editors.**RealBasicVSRNet**(*mid_channels=64*, *num_propagation_blocks=20*, *num_cleaning_blocks=20*, *dynamic_refine_thres=255*, *spynet_pretrained=None*, *is_fix_cleaning=False*, *is_sequential_cleaning=False*)

Bases: `mmengine.model.BaseModule`

RealBasicVSR network structure for real-world video super-resolution.

Support only x4 upsampling.

Paper: Investigating Tradeoffs in Real-World Video Super-Resolution, arXiv

Parameters

- **mid_channels** (*int, optional*) – Channel number of the intermediate features. Default: 64.
- **num_propagation_blocks** (*int, optional*) – Number of residual blocks in each propagation branch. Default: 20.
- **num_cleaning_blocks** (*int, optional*) – Number of residual blocks in the image cleaning module. Default: 20.
- **dynamic_refine_thres** (*int, optional*) – Stop cleaning the images when the residue is smaller than this value. Default: 255.
- **spynet_pretrained** (*str, optional*) – Pre-trained model path of SPyNet. Default: None.
- **is_fix_cleaning** (*bool, optional*) – Whether to fix the weights of the image cleaning module during training. Default: False.
- **is_sequential_cleaning** (*bool, optional*) – Whether to clean the images sequentially. This is used to save GPU memory, but the speed is slightly slower. Default: False.

forward(*lqs*, *return_lqs=False*)

Forward function for BasicVSR++.

Parameters

- **lqs** (*tensor*) – Input low quality (LQ) sequence with shape (n, t, c, h, w).
- **return_lqs** (*bool*) – Whether to return LQ sequence. Default: False.

Returns Output HR sequence.

Return type Tensor

class mmedit.models.editors.**RealESRGAN**(*generator*, *discriminator=None*, *gan_loss=None*, *pixel_loss=None*, *perceptual_loss=None*, *is_use_sharpened_gt_in_pixel=False*, *is_use_sharpened_gt_in_percep=False*, *is_use_sharpened_gt_in_gan=False*, *is_use_ema=True*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.editors.srgan.SRGAN

Real-ESRGAN model for single image super-resolution.

Ref: Real-ESRGAN: Training Real-World Blind Super-Resolution with Pure Synthetic Data, 2021.

Note: generator_ema is realized in EMA_HOOK

Parameters

- **generator** (*dict*) – Config for the generator.
- **discriminator** (*dict, optional*) – Config for the discriminator. Default: None.
- **gan_loss** (*dict, optional*) – Config for the gan loss. Note that the loss weight in gan loss is only for the generator.
- **pixel_loss** (*dict, optional*) – Config for the pixel loss. Default: None.
- **perceptual_loss** (*dict, optional*) – Config for the perceptual loss. Default: None.
- **is_use_sharpened_gt_in_pixel** (*bool, optional*) – Whether to use the image sharpened by unsharp masking as the GT for pixel loss. Default: False.
- **is_use_sharpened_gt_in_percep** (*bool, optional*) – Whether to use the image sharpened by unsharp masking as the GT for perceptual loss. Default: False.
- **is_use_sharpened_gt_in_gan** (*bool, optional*) – Whether to use the image sharpened by unsharp masking as the GT for adversarial loss. Default: False.
- **is_use_ema** (*bool, optional*) – When to apply exponential moving average on the network weights. Default: True.
- **train_cfg** (*dict*) – Config for training. Default: None. You may change the training of gan by setting: *disc_steps*: how many discriminator updates after one generate update; *disc_init_steps*: how many discriminator updates at the start of the training. These two keys are useful when training with WGAN.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule. Default: None.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor. Default: None.

forward_tensor(*inputs*, *data_samples=None*, *training=False*)

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.
- **training** (*bool*) – Whether is training. Default: False.

Returns result of simple forward.

Return type Tensor

g_step(*batch_outputs*, *batch_gt_data*)

G step of GAN: Calculate losses of generator.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tuple[Tensor]*) – Batch GT data.

Returns Dict of losses.

Return type dict

d_step_real(*batch_outputs*, *batch_gt_data: torch.Tensor*)

Real part of D step.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tuple[Tensor]*) – Batch GT data.

Returns Real part of gan_loss for discriminator.

Return type Tensor

d_step_fake(*batch_outputs*, *batch_gt_data*)

Fake part of D step.

Parameters

- **batch_outputs** (*Tensor*) – Output of generator.
- **batch_gt_data** (*Tuple[Tensor]*) – Batch GT data.

Returns Fake part of gan_loss for discriminator.

Return type Tensor

extract_gt_data(*data_samples*)

extract gt data from data samples.

Parameters **data_samples** (*list*) – List of EditDataSample.

Returns Extract gt data.

Return type Tensor

class mmedit.models.editors.**UNetDiscriminatorWithSpectralNorm**(*in_channels*, *mid_channels=64*, *skip_connection=True*)

Bases: mmengine.model.BaseModule

A U-Net discriminator with spectral normalization.

Parameters

- **in_channels** (*int*) – Channel number of the input.

- **mid_channels** (*int, optional*) – Channel number of the intermediate features. Default: 64.
- **skip_connection** (*bool, optional*) – Whether to use skip connection. Default: True.

forward(*img*)

Forward function.

Parameters **img** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**Restormer**(*inp_channels=3*, *out_channels=3*, *dim=48*, *num_blocks=[4, 6, 6, 8]*, *num_refinement_blocks=4*, *heads=[1, 2, 4, 8]*, *ffn_expansion_factor=2.66*, *bias=False*, *LayerNorm_type='WithBias'*, *dual_pixel_task=False*, *dual_keys=['imgL', 'imgR']*)

Bases: mmengine.model.BaseModule

Restormer A PyTorch impl of: *Restormer: Efficient Transformer for High- Resolution Image Restoration*. Ref repo: https://github.com/swz30/Restormer.

Parameters

- **inp_channels** (*int*) – Number of input image channels. Default: 3.
- **out_channels** (*int*) – Number of output image channels: 3.
- **dim** (*int*) – Number of feature dimension. Default: 48.
- **num_blocks** (*List(int)*) – Depth of each Transformer layer. Default: [4, 6, 6, 8].
- **num_refinement_blocks** (*int*) – Number of refinement blocks. Default: 4.
- **heads** (*List(int)*) – Number of attention heads in different layers. Default: 7.
- **ffn_expansion_factor** (*float*) – Ratio of feed forward network expansion. Default: 2.66.
- **bias** (*bool*) – The bias of convolution. Default: False
- **LayerNorm_type** (*str|optional*) – Select layer Normalization type. Optional: 'WithBias','BiasFree' Default: 'WithBias'.
- **dual_pixel_task** (*bool*) – True for dual-pixel defocus deblurring only. Also set inp_channels=6. Default: False.
- **dual_keys** (*List*) – Keys of dual images in inputs. Default: ['imgL', 'imgR'].

forward(*inp_img*)

Forward function.

Parameters **inp_img** (*Tensor*) – Input tensor with shape (B, C, H, W).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**SAGAN**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *noise_size: Optional[int] = 128*, *num_classes: Optional[int] = None*, *ema_config: Optional[Dict] = None*)

Bases: *mmedit.models.base_models.BaseConditionalGAN*

Impelmentation of *Self-Attention Generative Adversarial Networks*.

<https://arxiv.org/abs/1805.08318>`_ (SAGAN), Spectral Normalization for Generative Adversarial Networks (SNGAN), and cGANs with Projection Discriminator (Proj-GAN).

Detailed architecture can be found in `SNGANGenerator` and `ProjDiscriminator`

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or `GenDataPreprocessor`.
- **generator_steps** (*int*) – Number of times the generator was completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – Number of times the discriminator was completely updated before the generator is updated. Defaults to 1.
- **noise_size** (*Optional[int]*) – Size of the input noise vector. Default to 128.
- **num_classes** (*Optional[int]*) – The number classes you would like to generate. Defaults to None.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*) → Tuple[torch.Tensor, dict]

Get disc loss. SAGAN, SNGAN and Proj-GAN use hinge loss to train the discriminator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

gen_loss(*disc_pred_fake: torch.Tensor*) → Tuple[torch.Tensor, dict]

Get disc loss. SAGAN, SNGAN and Proj-GAN use hinge loss to train the generator.

Parameters **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.

- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

class mmedit.models.editors.**SinGAN**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *num_scales: Optional[int] = None*, *iters_per_scale: int = 2000*, *noise_weight_init: int = 0.1*, *lr_scheduler_args: Optional[dict] = None*, *test_pkl_data: Optional[str] = None*, *ema_confg: Optional[dict] = None*)

Bases: *mmedit.models.base_models.BaseGAN*

SinGAN.

This model implement the single image generative adversarial model proposed in: Singan: Learning a Generative Model from a Single Natural Image, ICCV'19.

Notes for training:

- This model should be trained with our dataset `SinGANDataset`.
- In training, the `total_iters` arguments is related to the number of scales in the image pyramid and `iters_per_scale` in the `train_cfg`. You should set it carefully in the training config file.

Notes for model architectures:

- The generator and discriminator need `num_scales` in initialization. However, this arguments is generated by `create_real_pyramid` function from the `singan_dataset.py`. The last element in the returned list (`stop_scale`) is the value for `num_scales`. Pay attention that this scale is counted from zero. Please see our tutorial for SinGAN to obtain more details or our standard config for reference.

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or `GenDataPreprocessor`.
- **generator_steps** (*int*) – The number of times the generator is completely updated before the discriminator is updated. Defaults to 1.

- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.
- **num_scales** (*int*) – The number of scales/stages in generator/ discriminator. Note that this number is counted from zero, which is the same as the original paper. Defaults to None.
- **iters_per_scale** (*int*) – The training iteration for each resolution scale. Defaults to 2000.
- **noise_weight_init** (*float*) – The initialize weight of fixed noise. Defaults to 0.1
- **lr_scheduler_args** (*Optional[dict]*) – Arguments for learning schedulers. Note that in SinGAN, we use MultiStepLR, which is the same as the original paper. If not passed, no learning schedule will be used. Defaults to None.
- **test_pkl_data** (*Optional[str]*) – The path of pickle file which contains fixed noise and noise weight. This is must for test. Defaults to None.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

load_test_pkl()

Load pickle for test.

_from_numpy(*data: Tuple[list, numpy.ndarray]*) → Tuple[torch.Tensor, List[torch.Tensor]]

Convert input numpy array or list of numpy array to Tensor or list of Tensor.

Parameters **data** (*Tuple[list, np.ndarray]*) – Input data to convert.

Returns Converted Tensor or list of tensor.

Return type Tuple[Tensor, List[Tensor]]

get_module(*model: torch.nn.Module*, *module_name: str*) → torch.nn.Module

Get an inner module from model.

Since we will wrapper DDP for some model, we have to judge whether the module can be indexed directly.

Parameters

- **model** (*nn.Module*) – This model may wrapped with DDP or not.
- **module_name** (*str*) – The name of specific module.

Returns Returned sub module.

Return type nn.Module

construct_fixed_noises()

Construct the fixed noises list used in SinGAN.

forward(*inputs: mmedit.utils.ForwardInputs*, *data_samples: Optional[list] = None*, *mode=None*) → List[*mmedit.structures.EditDataSample*]

Forward function for SinGAN. For SinGAN, *inputs* should be a dict contains 'num_batches', 'mode' and other input arguments for the generator.

Parameters

- **inputs** (*dict*) – Dict containing the necessary information (e.g., noise, num_batches, mode) to generate image.
- **data_samples** (*Optional[list]*) – Data samples collated by `data_preprocessor`. Defaults to None.

- **mode** (*Optional[str]*) – *mode* is not used in BaseConditionalGAN. Defaults to None.

gen_loss(*disc_pred_fake: torch.Tensor*, *recon_imgs: torch.Tensor*) → Tuple[torch.Tensor, dict]

Generator loss for SinGAN. SinGAN use WGAN's loss and MSE loss to train the generator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **recon_imgs** (*Tensor*) – Reconstructive images.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*, *fake_data: torch.Tensor*, *real_data: torch.Tensor*) → Tuple[torch.Tensor, dict]

Get disc loss. SAGAN, SNGAN and Proj-GAN use hinge loss to train the generator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.
- **fake_data** (*Tensor*) – Generated images, used to calculate gradient penalty.
- **real_data** (*Tensor*) – Real images, used to calculate gradient penalty.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_gan(*inputs_dict: dict*, *data_sample: List[*mmedit.structures.EditDataSample*]*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train GAN model. In the training of GAN models, generator and discriminator are updated alternatively. In MMGeneration's design, *self.train_step* is called with data input. Therefore we always update discriminator, whose updating is relay on real data, and then determine if the generator needs to be updated based on the current number of iterations. More details about whether to update generator can be found in `should_gen_update()`.

Parameters

- **data** (*dict*) – Data sampled from dataloader.
- **data_sample** (*List[EditDataSample]*) – List of data sample contains GT and meta information.
- **optim_wrapper** (*OptimWrapperDict*) – OptimWrapperDict instance contains OptimWrapper of generator and discriminator.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

train_step(*data: dict*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train step for SinGAN model. SinGAN is trained with multi-resolution images, and each resolution is trained for *:attr:self.iters_per_scale* times.

We initialize the weight and learning rate scheduler of the corresponding module at the start of each resolution's training. At the end of each resolution's training, we update the weight of the noise of current resolution by mse loss between reconstruced image and real image.

Parameters

- **data** (*dict*) – Data sampled from dataloader.
- **optim_wrapper** (*OptimWrapperDict*) – OptimWrapperDict instance contains OptimWrapper of generator and discriminator.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

test_step(*data: dict*) → mmedit.utils.SampleList

Gets the generated image of given data in test progress. Before generate images, we call *:meth:self.load_test_pkl* to load the fixed noise and current stage of the model from the pickle file.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

class mmedit.models.editors.**SRCNNNet**(*channels=(3, 64, 32, 3)*, *kernel_sizes=(9, 1, 5)*, *upscale_factor=4*)

Bases: `mmengine.model.BaseModule`

SRCNN network structure for image super resolution.

SRCNN has three conv layers. For each layer, we can define the *in_channels*, *out_channels* and *kernel_size*. The input image will first be upsampled with a bicubic upsampler, and then super-resolved in the HR spatial size.

Paper: Learning a Deep Convolutional Network for Image Super-Resolution.

Parameters

- **channels** (*tuple[int]*) – A tuple of channel numbers for each layer including channels of input and output . Default: (3, 64, 32, 3).
- **kernel_sizes** (*tuple[int]*) – A tuple of kernel sizes for each conv layer. Default: (9, 1, 5).
- **upscale_factor** (*int*) – Upsampling factor. Default: 4.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**SRGAN**(*generator*, *discriminator=None*, *gan_loss=None*, *pixel_loss=None*, *perceptual_loss=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: *mmedit.models.base_models.BaseEditModel*

SRGAN model for single image super-resolution.

Ref: Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network.

Parameters

- **generator** (*dict*) – Config for the generator.
- **discriminator** (*dict*) – Config for the discriminator. Default: None.
- **gan_loss** (*dict*) – Config for the gan loss. Note that the loss weight in gan loss is only for the generator.
- **pixel_loss** (*dict*) – Config for the pixel loss. Default: None.
- **perceptual_loss** (*dict*) – Config for the perceptual loss. Default: None.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule. Default: None.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor. Default: None.

forward_train(*inputs*, *data_samples=None*, ***kwargs*)

Forward training. Losses of training is calculated in train_step.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.

Returns Result of forward_tensor with training=True.

Return type Tensor

forward_tensor(*inputs*, *data_samples=None*, *training=False*)

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.
- **training** (*bool*) – Whether is training. Default: False.

Returns result of simple forward.

Return type Tensor

if_run_g()

Calculates whether need to run the generator step.

if_run_d()

Calculates whether need to run the discriminator step.

g_step(*batch_outputs: torch.Tensor*, *batch_gt_data: torch.Tensor*)

G step of GAN: Calculate losses of generator.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Dict of losses.

Return type dict

d_step_real(*batch_outputs*, *batch_gt_data: torch.Tensor*)

Real part of D step.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Real part of gan_loss for discriminator.

Return type Tensor

d_step_fake(*batch_outputs: torch.Tensor*, *batch_gt_data*)

Fake part of D step.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Fake part of gan_loss for discriminator.

Return type Tensor

g_step_with_optim(*batch_outputs: torch.Tensor*, *batch_gt_data: torch.Tensor*, *optim_wrapper: mmengine.optim.OptimWrapperDict*)

G step with optim of GAN: Calculate losses of generator and run optim.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.
- **optim_wrapper** (*OptimWrapperDict*) – Optim wrapper dict.

Returns Dict of parsed losses.

Return type dict

d_step_with_optim(*batch_outputs: torch.Tensor*, *batch_gt_data: torch.Tensor*, *optim_wrapper: mmengine.optim.OptimWrapperDict*)

D step with optim of GAN: Calculate losses of discriminator and run optim.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.
- **optim_wrapper** (*OptimWrapperDict*) – Optim wrapper dict.

Returns Dict of parsed losses.

Return type dict

extract_gt_data(*data_samples*)

extract gt data from data samples.

Parameters **data_samples** (*list*) – List of EditDataSample.

Returns Extract gt data.

Return type Tensor

train_step(*data: List[dict]*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train step of GAN-based method.

Parameters

- **data** (*List[dict]*) – Data sampled from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

class mmedit.models.editors.**ModifiedVGG**(*in_channels*, *mid_channels*)

Bases: `mmengine.model.BaseModule`

A modified VGG discriminator with input size 128 x 128.

It is used to train SRGAN and ESRGAN.

Parameters

- **in_channels** (*int*) – Channel number of inputs. Default: 3.
- **mid_channels** (*int*) – Channel number of base intermediate features. Default: 64.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**MSRResNet**(*in_channels*, *out_channels*, *mid_channels=64*, *num_blocks=16*, *upscale_factor=4*)

Bases: mmengine.model.BaseModule

Modified SRResNet.

A compacted version modified from SRResNet in "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network".

It uses residual blocks without BN, similar to EDSR. Currently, it supports x2, x3 and x4 upsampling scale factor.

Parameters

- **in_channels** (*int*) – Channel number of inputs.
- **out_channels** (*int*) – Channel number of outputs.
- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64.
- **num_blocks** (*int*) – Block number in the trunk network. Default: 16.
- **upscale_factor** (*int*) – Upsampling factor. Support x2, x3 and x4. Default: 4.

_supported_upscale_factors = [2, 3, 4]

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

init_weights()

Init weights for models.

class mmedit.models.editors.**StableDiffusion**(*diffusion_scheduler*, *unet*, *vae*, *requires_safety_checker=True*, *unet_sample_size=64*, *init_cfg=None*)

Bases: mmengine.model.BaseModel

class to run stable diffsuion pipeline.

Parameters

- **diffusion_scheduler** (*dict*) – Diffusion scheduler config.
- **unet_cfg** (*dict*) – Unet config.
- **vae_cfg** (*dict*) – Vae config.
- **pretrained_ckpt_path** (*dict*) – Pretrained ckpt path for submodels in stable diffusion.
- **requires_safety_checker** (*bool*) – whether to run safety checker after image generated.
- **unet_sample_size** (*int*) – sampel size for unet.

init_weights()

load pretrained ckpt for each submodel.

to(*torch_device: Optional[Union[str, torch.device]] = None*)

put submodels to torch device.

Parameters **torch_device** (*Optional[Union[str, torch.device]]*) – device to put, default to None.

Returns class instance itsself.

Return type self(*StableDiffusion*)

infer(*prompt: Union[str, List[str]]*, *height: Optional[int] = None*, *width: Optional[int] = None*, *num_inference_steps: int = 50*, *guidance_scale: float = 7.5*, *negative_prompt: Optional[Union[str, List[str]]] = None*, *num_images_per_prompt: Optional[int] = 1*, *eta: float = 0.0*, *generator: Optional[torch.Generator] = None*, *latents: Optional[torch.FloatTensor] = None*, *show_progress=True*, *seed=1*)

Function invoked when calling the pipeline for generation.

Parameters

- **prompt** (*str* or *List[str]*) – The prompt or prompts to guide the image generation.
- **(int** (*height*) – defaults to self.unet_sample_size * self.vae_scale_factor): The height in pixels of the generated image.
- **optional** – defaults to self.unet_sample_size * self.vae_scale_factor): The height in pixels of the generated image.

:param [defaults to self.unet_sample_size * self.vae_scale_factor):] The height in pixels of the generated image.

Parameters

- **(int** (*width*) – defaults to self.unet_sample_size * self.vae_scale_factor): The width in pixels of the generated image.
- **optional** – defaults to self.unet_sample_size * self.vae_scale_factor): The width in pixels of the generated image.

:param [defaults to self.unet_sample_size * self.vae_scale_factor):] The width in pixels of the generated image.

Parameters

- **num_inference_steps** (*int*, *optional*, defaults to 50) – The number of denoising steps. More denoising steps usually lead to a higher quality image at the expense of slower inference.
- **guidance_scale** (*float*, *optional*, defaults to 7.5) – Guidance scale as defined in [Classifier-Free Diffusion Guidance] (https://arxiv.org/abs/2207.12598).
- **negative_prompt** (*str* or *List[str]*, *optional*) – The prompt or prompts not to guide the image generation. Ignored when not using guidance (i.e., ignored if *guidance_scale* is less than *1*).
- **num_images_per_prompt** (*int*, *optional*, defaults to 1) – The number of images to generate per prompt.
- **eta** (*float*, *optional*, defaults to 0.0) – Corresponds to parameter eta () in the DDIM paper: https://arxiv.org/abs/2010.02502. Only applies to [*schedulers.DDIMScheduler*], will be ignored for others.

- **generator** (*torch.Generator*, *optional*) – A [torch generator] to make generation deterministic.
- **latents** (*torch.FloatTensor*, *optional*) – Pre-generated noisy latents, sampled from a Gaussian distribution, to be used as inputs for image generation. Can be used to tweak the same generation with different prompts. If not provided, a latents tensor will be generated by sampling using the supplied random *generator*.

Returns

['samples', 'nsfw_content_detected']: 'samples': image result samples 'nsfw_content_detected': nsfw content flags for image samples.

Return type dict

_encode_prompt(*prompt*, *device*, *num_images_per_prompt*, *do_classifier_free_guidance*, *negative_prompt*)

Encodes the prompt into text encoder hidden states.

Parameters

- **prompt** (*str or list(int)*) – prompt to be encoded.
- **device** – (torch.device): torch device.
- **num_images_per_prompt** (*int*) – number of images that should be generated per prompt.
- **do_classifier_free_guidance** (*bool*) – whether to use classifier free guidance or not.
- **negative_prompt** (*str or List[str]*) – The prompt or prompts not to guide the image generation. Ignored when not using guidance (i.e., ignored if *guidance_scale* is less than *1*).

Returns text embeddings generated by clip text encoder.

Return type text_embeddings (torch.Tensor)

run_safety_checker(*image*, *device*, *dtype*)

run safety checker to check whether image has nsfw content.

Parameters

- **image** (*numpy.ndarray*) – image generated by stable diffusion.
- **device** (*torch.device*) – device to run safety checker.
- **dtype** (*torch.dtype*) – float type to run.

Returns

black image if nsfw content detected else input image. has_nsfw_concept (list[bool]):

flag list to indicate nsfw content detected.

Return type image (numpy.ndarray)

decode_latents(*latents*)

use vae to decode latents.

Parameters **latents** (*torch.Tensor*) – latents to decode.

Returns image result.

Return type image (numpy.ndarray)

prepare_extra_step_kwargs(*generator*, *eta*)

prepare extra kwargs for the scheduler step.

Parameters

- **generator** (*torch.Generator*) – generator for random functions.
- **eta** (*float*) – eta () is only used with the DDIMScheduler, it will be ignored for other schedulers. eta corresponds to in DDIM paper: https://arxiv.org/abs/2010.02502 and should be between [0, 1]

Returns dict contains 'generator' and 'eta'

Return type extra_step_kwargs (dict)

check_inputs(*prompt*, *height*, *width*)

check whether inputs are in suitable format or not.

prepare_latents(*batch_size*, *num_channels_latents*, *height*, *width*, *dtype*, *device*, *generator*, *latents=None*)

prepare latents for diffusion to run in latent space.

Parameters

- **batch_size** (*int*) – batch size.
- **num_channels_latents** (*int*) – latent channel nums.
- **height** (*int*) – image height.
- **width** (*int*) – image width.
- **dtype** (*torch.dtype*) – float type.
- **device** (*torch.device*) – torch device.
- **generator** (*torch.Generator*) – generator for random functions, defaults to None.
- **latents** (*torch.Tensor*) – Pre-generated noisy latents, defaults to None.

Returns prepared latents.

Return type latents (torch.Tensor)

abstract forward(*inputs: torch.Tensor*, *data_samples: Optional[list] = None*, *mode: str = 'tensor'*) → Union[Dict[str, torch.Tensor], list]

forward is not implemented now.

class mmedit.models.editors.**StyleGAN1**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *style_channels: int = 512*, *nkimgs_per_scale: dict = {}*, *interp_real: Optional[dict] = None*, *transition_kimgs: int = 600*, *prev_stage: int = 0*, *ema_config: Optional[Dict] = None*)

Bases: mmedit.models.editors.pggan.ProgressiveGrowingGAN

Implementation of *A Style-Based Generator Architecture for Generative Adversarial Networks*.

<https://openaccess.thecvf.com/content_CVPR_2019/html/Karras_A_Style-Based_Generator_Architecture_for_Generative_Adversarial_Networks_CVPR_2019_paper.html>`_ # noqa (StyleGANv1). This class is inheriant from *ProgressiveGrowingGAN* to support progressive training.

Detailed architecture can be found in StyleGAN1Generator and StyleGAN1Discriminator

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or GenDataPreprocessor.
- **style_channels** (*int*) – The number of channels for style code. Defaults to 128.
- **nkimgs_per_scale** (*dict*) – The number of images need for each resolution's training. Defaults to *{}*.
- **intep_real** (*dict, optional*) – The config of interpolation method for real images. If not passed, bilinear interpolation with align_corners will be used. Defaults to None.
- **transition_kimgs** (*int, optional*) – The number of images during used to transit from the previous torgb layer to newer torgb layer. Defaults to 600.
- **prev_stage** (*int, optional*) – The resolution of previous stage. Used for resume training. Defaults to 0.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*, *fake_data: torch.Tensor*, *real_data: torch.Tensor*) → Tuple[torch.Tensor, dict]

Get disc loss. StyleGANv1 use non-saturating gan loss and R1 gradient penalty. loss to train the discriminator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.
- **fake_data** (*Tensor*) – Generated images, used to calculate gradient penalty.
- **real_data** (*Tensor*) – Real images, used to calculate gradient penalty.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

gen_loss(*disc_pred_fake: torch.Tensor*) → Tuple[torch.Tensor, dict]

Generator loss for PGGAN. PGGAN use WGAN's loss to train the generator.

Parameters **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

Returns Loss value and a dict of log variables.

Return type Tuple[Tensor, dict]

class mmedit.models.editors.**StyleGAN2**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *ema_config: Optional[Dict] = None*, *loss_config=dict()*)

Bases: *mmedit.models.base_models.BaseGAN*

Impelmentation of *Analyzing and Improving the Image Quality of Stylegan*. # noqa.

Paper link: https://openaccess.thecvf.com/content_CVPR_2020/html/Karras_Analyzing_and_Improving_the_Image_Quality_of_StyleGAN_CVPR_2020_paper.html. # noqa

StyleGAN2Generator and StyleGAN2Discriminator

Parameters

- **generator** (*ModelType*) – The config or model of the generator.
- **discriminator** (*Optional[ModelType]*) – The config or model of the discriminator. Defaults to None.
- **data_preprocessor** (*Optional[Union[dict, Config]]*) – The pre-process config or GenDataPreprocessor.
- **generator_steps** (*int*) – The number of times the generator is completely updated before the discriminator is updated. Defaults to 1.
- **discriminator_steps** (*int*) – The number of times the discriminator is completely updated before the generator is updated. Defaults to 1.
- **ema_config** (*Optional[Dict]*) – The config for generator's exponential moving average setting. Defaults to None.

disc_loss(*disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*, *real_imgs: torch.Tensor*) → Tuple

Get disc loss. StyleGANv2 use the non-saturating loss and R1 gradient penalty to train the discriminator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.
- **real_imgs** (*Tensor*) – Input real images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

gen_loss(*disc_pred_fake: torch.Tensor*, *batch_size: int*) → Tuple

Get gen loss. StyleGANv2 use the non-saturating loss and generator path regularization to train the generator.

Parameters

- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **batch_size** (*int*) – Batch size for generating fake images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A dict of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_step(*data: dict*, *optim_wrapper: mmengine.optim.OptimWrapperDict*) → Dict[str, torch.Tensor]

Train GAN model. In the training of GAN models, generator and discriminator are updated alternatively. In MMEditing's design, *self.train_step* is called with data input. Therefore we always update discriminator, whose updating is relay on real data, and then determine if the generator needs to be updated based on the current number of iterations. More details about whether to update generator can be found in `should_gen_update()`.

Parameters

- **data** (*dict*) – Data sampled from dataloader.
- **optim_wrapper** (*OptimWrapperDict*) – OptimWrapperDict instance contains OptimWrapper of generator and discriminator.

Returns A `dict` of tensor for logging.

Return type Dict[str, torch.Tensor]

class mmedit.models.editors.**StyleGAN3**(*generator: ModelType*, *discriminator: Optional[ModelType] = None*, *data_preprocessor: Optional[Union[dict, mmengine.Config]] = None*, *generator_steps: int = 1*, *discriminator_steps: int = 1*, *forward_kwargs: Optional[Dict] = None*, *ema_config: Optional[Dict] = None*, *loss_config=dict()*)

Bases: `mmedit.models.editors.stylegan2.StyleGAN2`

Impelmentation of *Alias-Free Generative Adversarial Networks*. # noqa.

Paper link: https://nvlabs-fi-cdn.nvidia.com/stylegan3/stylegan3-paper.pdf # noqa

Detailed architecture can be found in

`StyleGAN3Generator` and `StyleGAN2Discriminator`

test_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as *val_step()*.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

val_step(*data: dict*) → mmedit.utils.typing.SampleList

Gets the generated image of given data. Same as `val_step()`.

Parameters **data** (*dict*) – Data sampled from metric specific sampler. More detials in *Metrics* and *Evaluator*.

Returns A list of `EditDataSample` contain generated results.

Return type SampleList

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

sample_equivarience_pairs(*batch_size*, *sample_mode='ema'*, *eq_cfg=dict(compute_eqt_int=False, compute_eqt_frac=False, compute_eqr=False, translate_max=0.125, rotate_max=1)*, *sample_kwargs=dict()*)

class mmedit.models.editors.**StyleGAN3Generator**(*out_size*, *style_channels*, *img_channels*, *noise_size=512*, *rgb2bgr=False*, *pretrained=None*, *synthesis_cfg=dict(type='SynthesisNetwork')*, *mapping_cfg=dict(type='MappingNetwork')*)

Bases: `torch.nn.Module`

StyleGAN3 Generator.

In StyleGAN3, we make several changes to StyleGANv2's generator which include transformed fourier features, filtered nonlinearities and non-critical sampling, etc. More details can be found in: Alias-Free Generative Adversarial Networks NeurIPS'2021.

Ref: https://github.com/NVlabs/stylegan3

Parameters

- **out_size** (*int*) – The output size of the StyleGAN3 generator.
- **style_channels** (*int*) – The number of channels for style code.
- **img_channels** (*int*) – The number of output's channels.
- **noise_size** (*int, optional*) – Size of the input noise vector. Defaults to 512.
- **rgb2bgr** (*bool, optional*) – Whether to reformat the output channels with order *bgr*. We provide several pre-trained StyleGAN3 weights whose output channels order is *rgb*. You can set this argument to True to use the weights.
- **pretrained** (*str | dict, optional*) – Path for the pretrained model or dict containing information for pretained models whose necessary key is 'ckpt_path'. Besides, you can also provide 'prefix' to load the generator part from the whole state dict. Defaults to None.
- **synthesis_cfg** (*dict, optional*) – Config for synthesis network. Defaults to dict(type='SynthesisNetwork').
- **mapping_cfg** (*dict, optional*) – Config for mapping network. Defaults to dict(type='MappingNetwork').

_load_pretrained_model(*ckpt_path*, *prefix=''*, *map_location='cpu'*, *strict=True*)

forward(*noise*, *num_batches=0*, *input_is_latent=False*, *truncation=1*, *num_truncation_layer=None*, *update_emas=False*, *force_fp32=True*, *return_noise=False*, *return_latents=False*)

Forward Function for stylegan3.

Parameters

- **noise** (*torch.Tensor | callable | None*) – You can directly give a batch of noise through a `torch.Tensor` or offer a callable function to sample a batch of noise data. Otherwise, the `None` indicates to use the default noise sampler.
- **num_batches** (*int, optional*) – The number of batch size. Defaults to 0.
- **input_is_latent** (*bool, optional*) – If *True*, the input tensor is the latent tensor. Defaults to False.
- **truncation** (*float, optional*) – Truncation factor. Give value less than 1., the truncation trick will be adopted. Defaults to 1.
- **num_truncation_layer** (*int, optional*) – Number of layers use truncated latent. Defaults to None.
- **update_emas** (*bool, optional*) – Whether update moving average of mean latent. Defaults to False.
- **force_fp32** (*bool, optional*) – Force fp32 ignore the weights. Defaults to True.
- **return_noise** (*bool, optional*) – If True, `noise_batch` will be returned in a dict with `fake_img`. Defaults to False.
- **return_latents** (*bool, optional*) – If True, `latent` will be returned in a dict with `fake_img`. Defaults to False.

Returns Generated image tensor or dictionary containing more data.

Return type torch.Tensor | dict

get_mean_latent(*num_samples=4096*, ***kwargs*)

Get mean latent of W space in this generator.

Parameters **num_samples** (*int, optional*) – Number of sample times. Defaults to 4096.

Returns Mean latent of this generator.

Return type Tensor

`get_training_kwargs`(*phase*)

Get training kwargs. In StyleGANv3, we enable fp16, and update mangitude ema during training of discriminator. This function is used to pass related arguments.

Parameters **phase** (*str*) – Current training phase.

Returns Training kwargs.

Return type dict

`class mmedit.models.editors.SwinIRNet`(*img_size=64*, *patch_size=1*, *in_chans=3*, *embed_dim=96*, *depths=[6, 6, 6, 6]*, *num_heads=[6, 6, 6, 6]*, *window_size=7*, *mlp_ratio=4.0*, *qkv_bias=True*, *qk_scale=None*, *drop_rate=0.0*, *attn_drop_rate=0.0*, *drop_path_rate=0.1*, *norm_layer=nn.LayerNorm*, *ape=False*, *patch_norm=True*, *use_checkpoint=False*, *upscale=2*, *img_range=1.0*, *upsampler=''*, *resi_connection='1conv'*, ***kwargs*)

Bases: `mmengine.model.BaseModule`

SwinIR A PyTorch impl of: *SwinIR: Image Restoration Using Swin Transformer*, based on Swin Transformer. Ref repo: https://github.com/JingyunLiang/SwinIR

Parameters

- **img_size** (*int | tuple(int)*) – Input image size. Default 64
- **patch_size** (*int | tuple(int)*) – Patch size. Default: 1
- **in_chans** (*int*) – Number of input image channels. Default: 3
- **embed_dim** (*int*) – Patch embedding dimension. Default: 96
- **depths** (*tuple(int)*) – Depth of each Swin Transformer layer. Default: [6, 6, 6, 6]
- **num_heads** (*tuple(int)*) – Number of attention heads in different layers. Default: [6, 6, 6, 6]
- **window_size** (*int*) – Window size. Default: 7
- **mlp_ratio** (*float*) – Ratio of mlp hidden dim to embedding dim. Default: 4
- **qkv_bias** (*bool*) – If True, add a learnable bias to query, key, value. Default: True
- **qk_scale** (*float*) – Override default qk scale of head_dim ** -0.5 if set. Default: None
- **drop_rate** (*float*) – Dropout rate. Default: 0
- **attn_drop_rate** (*float*) – Attention dropout rate. Default: 0
- **drop_path_rate** (*float*) – Stochastic depth rate. Default: 0.1
- **norm_layer** (*nn.Module*) – Normalization layer. Default: nn.LayerNorm.
- **ape** (*bool*) – If True, add absolute position embedding to the patch embedding. Default: False
- **patch_norm** (*bool*) – If True, add normalization after patch embedding. Default: True
- **use_checkpoint** (*bool*) – Whether to use checkpointing to save memory. Default: False
- **upscale** (*int*) – Upscale factor. 2/3/4/8 for image SR, 1 for denoising and compress artifact reduction. Default: 2

- **img_range** (*float*) – Image range. 1. or 255. Default: 1.0
- **upsampler** (*string, optional*) – The reconstruction module. 'pixelshuffle' / 'pixelshuffledirect' /'nearest+conv'/None. Default: ''
- **resi_connection** (*string*) – The convolutional block before residual connection. '1conv'/'3conv'. Default: '1conv'

_init_weights(*m*)

no_weight_decay()

no_weight_decay_keywords()

check_image_size(*x*)

Check image size and pad images so that it has enough dimension do window size.

Parameters **x** – input tensor image with (B, C, H, W) shape.

forward_features(*x*)

Forward function of Deep Feature Extraction.

Parameters **x** (*Tensor*) – Input tensor with shape (B, C, H, W).

Returns Forward results.

Return type Tensor

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (B, C, H, W).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**TDAN**(*generator*, *pixel_loss*, *lq_pixel_loss*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.BaseEditModel

TDAN model for video super-resolution.

Paper: TDAN: Temporally-Deformable Alignment Network for Video Super- Resolution, CVPR, 2020

Parameters

- **generator** (*dict*) – Config for the generator structure.
- **pixel_loss** (*dict*) – Config for pixel-wise loss.
- **lq_pixel_loss** (*dict*) – Config for pixel-wise loss for the LQ images.
- **train_cfg** (*dict*) – Config for training. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor.

forward_train(*inputs*, *data_samples=None*, ***kwargs*)

Forward training. Returns dict of losses of training.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.

Returns Dict of losses.

Return type dict

forward_tensor(*inputs*, *data_samples=None*, *training=False*, ***kwargs*)

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.
- **training** (*bool*) – Whether is training. Default: False.

Returns

results of forward inference and forward train.

Return type (Tensor | List[Tensor])

class mmedit.models.editors.**TDANNet**(*in_channels=3*, *mid_channels=64*, *out_channels=3*, *num_blocks_before_align=5*, *num_blocks_after_align=10*)

Bases: mmengine.model.BaseModule

TDAN network structure for video super-resolution.

Support only x4 upsampling.

Paper: TDAN: Temporally-Deformable Alignment Network for Video Super- Resolution, CVPR, 2020

Parameters

- **in_channels** (*int*) – Number of channels of the input image. Default: 3.
- **mid_channels** (*int*) – Number of channels of the intermediate features. Default: 64.
- **out_channels** (*int*) – Number of channels of the output image. Default: 3.
- **num_blocks_before_align** (*int*) – Number of residual blocks before temporal alignment. Default: 5.
- **num_blocks_after_align** (*int*) – Number of residual blocks after temporal alignment. Default: 10.

forward(*lrs*)

Forward function for TDANNet.

Parameters **lrs** (*Tensor*) – Input LR sequence with shape (n, t, c, h, w).

Returns Output HR image with shape (n, c, 4h, 4w) and aligned LR images with shape (n, t, c, h, w).

Return type tuple[Tensor]

class mmedit.models.editors.**TOFlowVFINet**(*rgb_mean=[0.485, 0.456, 0.406]*, *rgb_std=[0.229, 0.224, 0.225]*, *flow_cfg=dict(norm_cfg=None, pretrained=None)*, *init_cfg=None*)

Bases: mmengine.model.BaseModule

PyTorch implementation of TOFlow for video frame interpolation.

Paper: Xue et al., Video Enhancement with Task-Oriented Flow, IJCV 2018 Code reference:

1. https://github.com/anchen1011/toflow
2. https://github.com/Coldog2333/pytoflow

Parameters

- **rgb_mean** (*list[float]*) – Image mean in RGB orders. Default: [0.485, 0.456, 0.406]
- **rgb_std** (*list[float]*) – Image std in RGB orders. Default: [0.229, 0.224, 0.225]
- **flow_cfg** (*dict*) – Config of SPyNet. Default: dict(norm_cfg=None, pretrained=None)
- **init_cfg** (*dict, optional*) – Initialization config dict. Default: None.

forward(*imgs*)

Parameters **imgs** – Input frames with shape of (b, 2, 3, h, w).

Returns Interpolated frame with shape of (b, 3, h, w).

Return type Tensor

class mmedit.models.editors.**TOFlowVSRNet**(*adapt_official_weights=False*)

Bases: mmengine.model.BaseModule

PyTorch implementation of TOFlow.

In TOFlow, the LR frames are pre-upsampled and have the same size with the GT frames.

Paper: Xue et al., Video Enhancement with Task-Oriented Flow, IJCV 2018 Code reference:

1. https://github.com/anchen1011/toflow
2. https://github.com/Coldog2333/pytoflow

Parameters **adapt_official_weights** (*bool*) – Whether to adapt the weights translated from the official implementation. Set to false if you want to train from scratch. Default: False

forward(*lrs*)

Parameters **lrs** – Input lr frames: (b, 7, 3, h, w).

Returns SR frame: (b, 3, h, w).

Return type Tensor

class mmedit.models.editors.**ToFResBlock**

Bases: torch.nn.Module

ResNet architecture.

Three-layers ResNet/ResBlock

forward(*frames*)

Parameters **frames** (*Tensor*) – Tensor with shape of (b, 2, 3, h, w).

Returns Interpolated frame with shape of (b, 3, h, w).

Return type Tensor

class mmedit.models.editors.LTE(*requires_grad=True*, *pixel_range=1.0*, *load_pretrained_vgg=True*)

Bases: mmengine.model.BaseModule

Learnable Texture Extractor.

Based on pretrained VGG19. Generate features in 3 levels.

Parameters

- **requires_grad** (*bool*) – Require grad or not. Default: True.
- **pixel_range** (*float*) – Pixel range of geature. Default: 1.
- **load_pretrained_vgg** (*bool*) – Load pretrained VGG from torchvision. Default: True. Train: must load pretrained VGG. Eval: needn't load pretrained VGG, because we will load pretrained LTE.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, 3, h, w).

Returns

Forward results in 3 levels. x_level3: Forward results in level 3 (n, 256, h/4, w/4). x_level2: Forward results in level 2 (n, 128, h/2, w/2). x_level1: Forward results in level 1 (n, 64, h, w).

Return type Tuple[Tensor]

class mmedit.models.editors.TTSR(*generator*, *extractor*, *transformer*, *pixel_loss*, *discriminator=None*, *perceptual_loss=None*, *transferal_perceptual_loss=None*, *gan_loss=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *data_preprocessor=None*)

Bases: mmedit.models.editors.srgan.SRGAN

TTSR model for Reference-based Image Super-Resolution.

Paper: Learning Texture Transformer Network for Image Super-Resolution.

Parameters

- **generator** (*dict*) – Config for the generator.
- **extractor** (*dict*) – Config for the extractor.
- **transformer** (*dict*) – Config for the transformer.
- **pixel_loss** (*dict*) – Config for the pixel loss.
- **discriminator** (*dict*) – Config for the discriminator. Default: None.
- **perceptual_loss** (*dict*) – Config for the perceptual loss. Default: None.
- **transferal_perceptual_loss** (*dict*) – Config for the transferal perceptual loss. Default: None.
- **gan_loss** (*dict*) – Config for the GAN loss. Default: None

- **train_cfg** (*dict*) – Config for train. Default: None.
- **test_cfg** (*dict*) – Config for testing. Default: None.
- **init_cfg** (*dict, optional*) – The weight initialized config for BaseModule. Default: None.
- **data_preprocessor** (*dict, optional*) – The pre-process config of BaseDataPreprocessor. Default: None.

forward_tensor(*inputs*, *data_samples=None*, *training=False*)

Forward tensor. Returns result of simple forward.

Parameters

- **inputs** (*torch.Tensor*) – batch input tensor collated by data_preprocessor.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by data_preprocessor.
- **training** (*bool*) – Whether is training. Default: False.

Returns

results of forward inference and forward train.

Return type (Tensor | Tuple[List[Tensor]])

if_run_g()

Calculates whether need to run the generator step.

if_run_d()

Calculates whether need to run the discriminator step.

g_step(*batch_outputs*, *batch_gt_data:* mmedit.structures.EditDataSample)

G step of GAN: Calculate losses of generator.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.

Returns Dict of losses.

Return type dict

g_step_with_optim(*batch_outputs: torch.Tensor*, *batch_gt_data: torch.Tensor*, *optim_wrapper: mmengine.optim.OptimWrapperDict*)

G step with optim of GAN: Calculate losses of generator and run optim.

Parameters

- **batch_outputs** (*Tensor*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.
- **optim_wrapper** (*OptimWrapperDict*) – Optim wrapper dict.

Returns Dict of parsed losses.

Return type dict

d_step_with_optim(*batch_outputs*, *batch_gt_data*, *optim_wrapper*)

D step with optim of GAN: Calculate losses of discriminator and run optim.

Parameters

- **batch_outputs** (*Tuple[Tensor]*) – Batch output of generator.
- **batch_gt_data** (*Tensor*) – Batch GT data.
- **optim_wrapper** (*OptimWrapper*) – Optim wrapper of discriminator.

Returns Dict of parsed losses.

Return type dict

class mmedit.models.editors.**SearchTransformer**

Bases: torch.nn.Module

Search texture reference by transformer.

Include relevance embedding, hard-attention and soft-attention.

gather(*inputs*, *dim*, *index*)

Hard Attention. Gathers values along an axis specified by dim.

Parameters

- **inputs** (*Tensor*) – The source tensor. (N, C*k*k, H*W)
- **dim** (*int*) – The axis along which to index.
- **index** (*Tensor*) – The indices of elements to gather. (N, H*W)

results: outputs (Tensor): The result tensor. (N, C*k*k, H*W)

forward(*img_lq*, *ref_lq*, *refs*)

Texture transformer.

Q = LTE(img_lq) K = LTE(ref_lq) V = LTE(ref), from V_level_n to V_level_1

Relevance embedding aims to embed the relevance between the LQ and Ref image by estimating the similarity between Q and K.

Hard-Attention: Only transfer features from the most relevant position in V for each query.

Soft-Attention: synthesize features from the transferred GT texture features T and the LQ features F from the backbone.

Parameters

- **extractor** (*All args are features come from*) – These features contain 3 levels. When upscale_factor=4, the size ratio of these features is level3:level2:level1 = 1:2:4.
- **img_lq** (*Tensor*) – Tensor of 4x bicubic-upsampled lq image. (N, C, H, W)
- **ref_lq** (*Tensor*) – Tensor of ref_lq. ref_lq is obtained by applying bicubic down-sampling and up-sampling with factor 4x on ref. (N, C, H, W)
- **refs** (*Tuple[Tensor]*) – Tuple of ref tensors. [(N, C, H, W), (N, C/2, 2H, 2W), ...]

Returns

tuple contains: soft_attention (Tensor): Soft-Attention tensor. (N, 1, H, W)

textures (Tuple[Tensor]): Transferred GT textures. [(N, C, H, W), (N, C/2, 2H, 2W), ...]

Return type tuple

class mmedit.models.editors.**TTSRDiscriminator**(*in_channels=3*, *in_size=160*)

Bases: mmengine.model.BaseModule

A discriminator for TTSR.

Parameters

- **in_channels** (*int*) – Channel number of inputs. Default: 3.
- **in_size** (*int*) – Size of input image. Default: 160.

forward(*x*)

Forward function.

Parameters **x** (*Tensor*) – Input tensor with shape (n, c, h, w).

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**TTSRNet**(*in_channels*, *out_channels*, *mid_channels=64*, *texture_channels=64*, *num_blocks=(16, 16, 8, 4)*, *res_scale=1.0*)

Bases: mmengine.model.BaseModule

TTSR network structure (main-net) for reference-based super-resolution.

Paper: Learning Texture Transformer Network for Image Super-Resolution

Adapted from 'https://github.com/researchmm/TTSR.git' 'https://github.com/researchmm/TTSR' Copyright permission at 'https://github.com/researchmm/TTSR/issues/38'.

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels in the output image
- **mid_channels** (*int*) – Channel number of intermediate features. Default: 64
- **texture_channels** (*int*) – Number of texture channels. Default: 64.
- **num_blocks** (*tuple[int]*) – Block numbers in the trunk network. Default: (16, 16, 8, 4)
- **res_scale** (*float*) – Used to scale the residual in residual block. Default: 1.

forward(*x*, *soft_attention*, *textures*)

Forward function.

Parameters

- **x** (*Tensor*) – Input tensor with shape (n, c, h, w).
- **soft_attention** (*Tensor*) – Soft-Attention tensor with shape (n, 1, h, w).
- **textures** (*Tuple[Tensor]*) – Transferred HR texture tensors. [(N, C, H, W), (N, C/2, 2H, 2W), ...]

Returns Forward results.

Return type Tensor

class mmedit.models.editors.**WGANGP**(**args*, ***kwargs*)

Bases: *mmedit.models.base_models.BaseGAN*

Impelmentation of *Improved Training of Wasserstein GANs*.

Paper link: https://arxiv.org/pdf/1704.00028

Detailed architecture can be found in `WGANGPGenerator` and `WGANGPDiscriminator`

disc_loss(*real_data: torch.Tensor*, *fake_data: torch.Tensor*, *disc_pred_fake: torch.Tensor*, *disc_pred_real: torch.Tensor*) → Tuple

Get disc loss. WGAN-GP use the wgan loss and gradient penalty to train the discriminator.

Parameters

- **real_data** (*Tensor*) – Real input data.
- **fake_data** (*Tensor*) – Fake input data.
- **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.
- **disc_pred_real** (*Tensor*) – Discriminator's prediction of the real images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

gen_loss(*disc_pred_fake: torch.Tensor*) → Tuple

Get gen loss. DCGAN use the wgan loss to train the generator.

Parameters **disc_pred_fake** (*Tensor*) – Discriminator's prediction of the fake images.

Returns Loss value and a dict of log variables.

Return type tuple[Tensor, dict]

train_discriminator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train discriminator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader.
- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

train_generator(*inputs: dict*, *data_samples: List[*mmedit.structures.EditDataSample*]*, *optimizer_wrapper: mmengine.optim.OptimWrapper*) → Dict[str, torch.Tensor]

Train generator.

Parameters

- **inputs** (*dict*) – Inputs from dataloader.
- **data_samples** (*List[EditDataSample]*) – Data samples from dataloader. Do not used in generator's training.

- **optim_wrapper** (*OptimWrapper*) – OptimWrapper instance used to update model parameters.

Returns A `dict` of tensor for logging.

Return type Dict[str, Tensor]

1.53 mmedit.utils

1.53.1 Package Contents

Functions

modify_args()	Modify args of argparse.ArgumentParser.
get_box_info(pred_bbox, original_shape, final_size)	**param pred_bbox** The bounding box for the instance
reorder_image(img[, input_order])	Reorder images to 'HWC' order.
tensor2img(tensor[, out_type, min_max])	Convert torch Tensors into image numpy arrays.
to_numpy(img[, dtype])	Convert data into numpy arrays of dtype.
download_from_url(url[, dest_path, dest_dir, hash_prefix])	Download object at the given URL to a local path.
print_colored_log(msg[, level, color])	Print colored log with default logger.
get_sampler(sample_kwargs, runner)	Get a sampler to loop input data.
register_all_modules(→ None)	Register all modules in mmedit into the registries.
try_import(→ Optional[types.ModuleType])	Try to import a module.
add_gaussian_noise(img, mu, sigma)	Add Gaussian Noise on the input image.
adjust_gamma(image[, gamma, gain])	Performs Gamma Correction on the input image.
bbox2mask(img_shape, bbox[, dtype])	Generate mask in np.ndarray from bbox.
brush_stroke_mask(img_shape[, num_vertices, ...])	Generate free-form mask.
get_irregular_mask(img_shape[, area_ratio_range])	Get irregular mask with the constraints in mask ratio.
make_coord(shape[, ranges, flatten])	Make coordinates at grid centers.
random_bbox(img_shape, max_bbox_shape[, ...])	Generate a random bbox for the mask on a given image.
random_choose_unknown(unknown, crop_size)	Randomly choose an unknown start (top-left) point for a given crop_size.

Attributes

MMEDIT_CACHE_DIR
ConfigType
ForwardInputs
LabelVar
NoiseVar
SampleList

mmedit.utils.**modify_args**()

Modify args of argparse.ArgumentParser.

mmedit.utils.**get_box_info**(*pred_bbox*, *original_shape*, *final_size*)

Parameters

- **pred_bbox** – The bounding box for the instance
- **original_shape** – Original image shape
- **final_size** – Size of the final output

Returns [L_pad, R_pad, T_pad, B_pad, rh, rw]

Return type List

mmedit.utils.**reorder_image**(*img*, *input_order='HWC'*)

Reorder images to 'HWC' order.

If the input_order is (h, w), return (h, w, 1); If the input_order is (c, h, w), return (h, w, c); If the input_order is (h, w, c), return as it is.

Parameters

- **img** (*np.ndarray*) – Input image.
- **input_order** (*str*) – Whether the input order is 'HWC' or 'CHW'. If the input image shape is (h, w), input_order will not have effects. Default: 'HWC'.

Returns Reordered image.

Return type np.ndarray

mmedit.utils.**tensor2img**(*tensor*, *out_type=np.uint8*, *min_max=(0, 1)*)

Convert torch Tensors into image numpy arrays.

After clamping to (min, max), image values will be normalized to [0, 1].

For different tensor shapes, this function will have different behaviors:

1. **4D mini-batch Tensor of shape (N x 3/1 x H x W):** Use *make_grid* to stitch images in the batch dimension, and then convert it to numpy array.
2. **3D Tensor of shape (3/1 x H x W) and 2D Tensor of shape (H x W):** Directly change to numpy array.

Note that the image channel in input tensors should be RGB order. This function will convert it to cv2 convention, i.e., (H x W x C) with BGR order.

Parameters

- **tensor** (*Tensor | list[Tensor]*) – Input tensors.
- **out_type** (*numpy type*) – Output types. If `np.uint8`, transform outputs to uint8 type with range [0, 255]; otherwise, float type with range [0, 1]. Default: `np.uint8`.
- **min_max** (*tuple*) – min and max values for clamp.

Returns 3D ndarray of shape (H x W x C) or 2D ndarray of shape (H x W).

Return type (Tensor | list[Tensor])

`mmedit.utils.`**to_numpy**(*img*, *dtype=np.float64*)

Convert data into numpy arrays of dtype.

Parameters

- **img** (*Tensor | np.ndarray*) – Input data.
- **dtype** (*np.dtype*) – Set the data type of the output. Default: np.float64

Returns Converted numpy arrays data.

Return type img (np.ndarray)

`mmedit.utils.`**MMEDIT_CACHE_DIR**

`mmedit.utils.`**download_from_url**(*url*, *dest_path=None*, *dest_dir=MMEDIT_CACHE_DIR*, *hash_prefix=None*)

Download object at the given URL to a local path.

Parameters

- **url** (*str*) – URL of the object to download.
- **dest_path** (*str*) – Path where object will be saved.
- **dest_dir** (*str*) – The directory of the destination. Defaults to `'~/.cache/openmmlab/mmgen/'`.
- **hash_prefix** (*string, optional*) – If not None, the SHA256 downloaded file should start with *hash_prefix*. Default: None.

Returns path for the downloaded file.

Return type str

`mmedit.utils.`**print_colored_log**(*msg*, *level=logging.INFO*, *color='magenta'*)

Print colored log with default logger.

Parameters

- **msg** (*str*) – Message to log.
- **level** (*int*) – The root logger level. Note that only the process of rank 0 is affected, while other processes will set the level to "Error" and be silent most of the time.Log level, default to 'info'.
- **color** (*str, optional*) – Color 'magenta'.

mmedit.utils.**get_sampler**(*sample_kwargs: dict*, *runner: Optional[mmengine.runner.Runner]*)

Get a sampler to loop input data.

Parameters

- **sample_kwargs** (*dict*) – _description_
- **runner** (*Optional[Runner]*) – _description_

Returns _description_

Return type _type_

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

Register all modules in mmedit into the registries.

Parameters **init_default_scope** (*bool*) – Whether initialize the mmedit default scope. When *init_default_scope=True*, the global default scope will be set to *mmedit*, and all registries will build modules from mmedit's registry node. To understand more about the registry, please refer to https://github.com/open-mmlab/mmengine/blob/main/docs/en/tutorials/registry.md Defaults to True.

mmedit.utils.**try_import**(*name: str*) → Optional[types.ModuleType]

Try to import a module.

Parameters **name** (*str*) – Specifies what module to import in absolute or relative terms (e.g. either pkg.mod or ..mod).

Returns If importing successfully, returns the imported module, otherwise returns None.

Return type ModuleType or None

mmedit.utils.**add_gaussian_noise**(*img: numpy.ndarray*, *mu*, *sigma*)

Add Gaussian Noise on the input image.

Parameters

- **img** (*np.ndarray*) – Input image.
- **mu** (*float*) – The mu value of the Gaussian function.
- **sigma** (*float*) – The sigma value of the Gaussian function.

Returns Gaussian noisy output image.

Return type noisy_img (np.ndarray)

mmedit.utils.**adjust_gamma**(*image*, *gamma=1*, *gain=1*)

Performs Gamma Correction on the input image.

This function is adopted from skimage: https://github.com/scikit-image/scikit-image/blob/7e4840bd9439d1dfb6beaf549998452c99f97fdd/skimage/exposure/ exposure.py#L439-L494

Also known as Power Law Transform. This function transforms the input image pixelwise according to the equation `O = I**gamma` after scaling each pixel to the range 0 to 1.

Parameters

- **image** (*np.ndarray*) – Input image.
- **gamma** (*float, optional*) – Non negative real number. Defaults to 1.
- **gain** (*float, optional*) – The constant multiplier. Defaults to 1.

Returns Gamma corrected output image.

Return type np.ndarray

mmedit.utils.**bbox2mask**(*img_shape*, *bbox*, *dtype='uint8'*)

Generate mask in np.ndarray from bbox.

The returned mask has the shape of (h, w, 1). '1' indicates the hole and '0' indicates the valid regions.

We prefer to use *uint8* as the data type of masks, which may be different from other codes in the community.

Parameters

- **img_shape** (*tuple[int]*) – The size of the image.
- **bbox** (*tuple[int]*) – Configuration tuple, (top, left, height, width)
- **np.dtype** (*str*) – Indicate the data type of returned masks. Default: 'uint8'

Returns Mask in the shape of (h, w, 1).

Return type mask (np.ndarray)

mmedit.utils.**brush_stroke_mask**(*img_shape*, *num_vertices=(4, 12)*, *mean_angle=2 * math.pi / 5*, *angle_range=2 * math.pi / 15*, *brush_width=(12, 40)*, *max_loops=4*, *dtype='uint8'*)

Generate free-form mask.

The method of generating free-form mask is in the following paper: Free-Form Image Inpainting with Gated Convolution.

When you set the config of this type of mask. You may note the usage of *np.random.randint* and the range of *np.random.randint* is [left, right).

We prefer to use *uint8* as the data type of masks, which may be different from other codes in the community.

TODO: Rewrite the implementation of this function.

Parameters

- **img_shape** (*tuple[int]*) – Size of the image.
- **num_vertices** (*int | tuple[int]*) – Min and max number of vertices. If only give an integer, we will fix the number of vertices. Default: (4, 12).
- **mean_angle** (*float*) – Mean value of the angle in each vertex. The angle is measured in radians. Default: 2 * math.pi / 5.
- **angle_range** (*float*) – Range of the random angle. Default: 2 * math.pi / 15.
- **brush_width** (*int | tuple[int]*) – (min_width, max_width). If only give an integer, we will fix the width of brush. Default: (12, 40).
- **max_loops** (*int*) – The max number of for loops of drawing strokes. Default: 4.
- **np.dtype** (*str*) – Indicate the data type of returned masks. Default: 'uint8'.

Returns Mask in the shape of (h, w, 1).

Return type mask (np.ndarray)

mmedit.utils.**get_irregular_mask**(*img_shape*, *area_ratio_range=(0.15, 0.5)*, ***kwargs*)

Get irregular mask with the constraints in mask ratio.

Parameters

- **img_shape** (*tuple[int]*) – Size of the image.
- **area_ratio_range** (*tuple(float)*) – Contain the minimum and maximum area

- **Default** (*ratio.*) – (0.15, 0.5).

Returns Mask in the shape of (h, w, 1).

Return type mask (np.ndarray)

mmedit.utils.**make_coord**(*shape*, *ranges=None*, *flatten=True*)

Make coordinates at grid centers.

Parameters

- **shape** (*tuple*) – shape of image.
- **ranges** (*tuple*) – range of coordinate value. Default: None.
- **flatten** (*bool*) – flatten to (n, 2) or Not. Default: True.

Returns coordinates.

Return type coord (Tensor)

mmedit.utils.**random_bbox**(*img_shape*, *max_bbox_shape*, *max_bbox_delta=40*, *min_margin=20*)

Generate a random bbox for the mask on a given image.

In our implementation, the max value cannot be obtained since we use *np.random.randint*. And this may be different with other standard scripts in the community.

Parameters

- **img_shape** (*tuple[int]*) – The size of a image, in the form of (h, w).
- **max_bbox_shape** (*int | tuple[int]*) – Maximum shape of the mask box, in the form of (h, w). If it is an integer, the mask box will be square.
- **max_bbox_delta** (*int | tuple[int]*) – Maximum delta of the mask box, in the form of (delta_h, delta_w). If it is an integer, delta_h and delta_w will be the same. Mask shape will be randomly sampled from the range of *max_bbox_shape - max_bbox_delta* and *max_bbox_shape*. Default: (40, 40).
- **min_margin** (*int | tuple[int]*) – The minimum margin size from the edges of mask box to the image boarder, in the form of (margin_h, margin_w). If it is an integer, margin_h and margin_w will be the same. Default: (20, 20).

Returns The generated box, (top, left, h, w).

Return type tuple[int]

mmedit.utils.**random_choose_unknown**(*unknown*, *crop_size*)

Randomly choose an unknown start (top-left) point for a given crop_size.

Parameters

- **unknown** (*np.ndarray*) – The binary unknown mask.
- **crop_size** (*tuple[int]*) – The given crop size.

Returns The top-left point of the chosen bbox.

Return type tuple[int]

mmedit.utils.**ConfigType**

mmedit.utils.**ForwardInputs**

mmedit.utils.**LabelVar**

mmedit.utils.**NoiseVar**

mmedit.utils.**SampleList**

1.54 Overview

This section introduce the following contents in terms of migration from MMEditing 0.x

- *New dependencies*
- *Overall structures*

1.54.1 New dependencies

MMEdit 1.x depends on some new packages, you can prepare a new clean environment and install again according to the *install tutorial*. Or install the below packages manually.

1. MMEngine: MMEngine is the core the OpenMMLab 2.0 architecture, and we splited many compentents unrelated to computer vision from MMCV to MMEngine.
2. MMCV: The computer vision package of OpenMMLab. This is not a new dependency, but you need to upgrade it to above 2.0.0rc0 version.
3. rich: A terminal formatting package, and we use it to beautify some outputs in the terminal.

1.54.2 Overall structures

We refactor overall structures in MMEdit 1.x as following.

- The `core` in the old versions of MMEdit is split into `engine`, `evaluation`, `structures`, and `visualization`
- The `pipelines` of `datasets` in the old versions of MMEdit is refactored to `transforms`
- The `models` in MMedit 1.x is refactored to five parts: `base_models`, `data_preprocessors`, `editors`, `layers` and `losses`.

1.54.3 Other config settings

We rename config file to new template: `{model_settings}_{module_setting}_{training_setting}_{datasets_info}`.

More details of config are shown in *config guides*.

1.55 Migration of Runtime Settings

We update runtime settings in MMEdit 1.x. Important modifications are as following.

- The `checkpoint_config` is moved to `default_hooks.checkpoint` and the `log_config` is moved to `default_hooks.logger`. And we move many hooks settings from the script code to the `default_hooks` field in the runtime configuration.
- The `resume_from` is removed. And we use `resume` to replace it.
  - If resume=True and load_from is not None, resume training from the checkpoint in load_from.
  - If resume=True and load_from is None, try to resume from the latest checkpoint in the work directory.

  - If resume=False and load_from is not None, only load the checkpoint, not resume training.
  - If resume=False and load_from is None, do not load nor resume.

- The `dist_params` field is a sub field of `env_cfg` now. And there are some new configurations in the `env_cfg`.
- The `workflow` related functionalities are removed.
- New field `visualizer`: The visualizer is a new design. We use a visualizer instance in the runner to handle results & log visualization and save to different backends, like Local, TensorBoard and Wandb.
- New field `default_scope`: The start point to search module for all registries.

```
checkpoint_config = dict(  # Config to set the checkpoint hook, Refer to https://github.
↪com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/checkpoint.py for implementation.
    interval=5000,  # The save interval is 5000 iterations
    save_optimizer=True,  # Also save optimizers
    by_epoch=False)  # Count by iterations
log_config = dict(  # Config to register logger hook
    interval=100,  # Interval to print the log
    hooks=[
        dict(type='TextLoggerHook', by_epoch=False),  # The logger used to record the␣
↪training process
        dict(type='TensorboardLoggerHook'),  # The Tensorboard logger is also supported
    ])
visual_config = None  # Visual config, we do not use it.
# runtime settings
dist_params = dict(backend='nccl')  # Parameters to setup distributed training, the port␣
↪can also be set
log_level = 'INFO'  # The level of logging
load_from = None # load models as a pre-trained model from a given path. This will not␣
↪resume training
resume_from = None # Resume checkpoints from a given path, the training will be resumed␣
↪from the iteration when the checkpoint's is saved
workflow = [('train', 1)]  # Workflow for runner. [('train', 1)] means there is only one␣
↪workflow and the workflow named 'train' is executed once. Keep this unchanged when␣
↪training current matting models
```

```
default_hooks = dict(  # Used to build default hooks
    checkpoint=dict(  # Config to set the checkpoint hook
        type='CheckpointHook',
        interval=5000,  # The save interval is 5000 iterations
        save_optimizer=True,
        by_epoch=False,  # Count by iterations
        out_dir=save_dir,
    ),
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=100),  # Config to register logger hook
    param_scheduler=dict(type='ParamSchedulerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)
default_scope = 'mmedit' # Used to set registries location
env_cfg = dict(  # Parameters to setup distributed training, the port can also be set
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=4),
    dist_cfg=dict(backend='nccl'),
```

(continues on next page)

(continued from previous page)

```
)
log_level = 'INFO'  # The level of logging
log_processor = dict(type='LogProcessor', window_size=100, by_epoch=False)  # Used to␣
↪build log processor
load_from = None  # load models as a pre-trained model from a given path. This will not␣
↪resume training.
resume = False  # Resume checkpoints from a given path, the training will be resumed␣
↪from the epoch when the checkpoint's is saved.
```

1.56 Migration of Model Settings

We update model settings in MMEdit 1.x. Important modifications are as following.

- Remove `pretrained` fields.
- Add `train_cfg` and `test_cfg` fields in model settings.
- Add `data_preprocessor` fields. Normalization and color space transforms operations are moved from datasets transforms pipelines to data_preprocessor. We will introduce data_preprocessor later.

```
model = dict(
    type='BasicRestorer',  # Name of the model
    generator=dict(  # Config of the generator
        type='EDSR',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pretrained=None,
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean'))  # Config for␣
↪pixel loss model training and testing settings
```

```
model = dict(
    type='BaseEditModel',  # Name of the model
    generator=dict(  # Config of the generator
        type='EDSRNet',  # Type of the generator
        in_channels=3,  # Channel number of inputs
        out_channels=3,  # Channel number of outputs
        mid_channels=64,  # Channel number of intermediate features
        num_blocks=16,  # Block number in the trunk network
        upscale_factor=scale, # Upsampling factor
        res_scale=1,  # Used to scale the residual in residual block
        rgb_mean=(0.4488, 0.4371, 0.4040),  # Image mean in RGB orders
        rgb_std=(1.0, 1.0, 1.0)),  # Image std in RGB orders
    pixel_loss=dict(type='L1Loss', loss_weight=1.0, reduction='mean')  # Config for␣
↪pixel loss
    train_cfg=dict(),  # Config of training model.
```

(continues on next page)

(continued from previous page)

```
    test_cfg=dict(),  # Config of testing model.
    data_preprocessor=dict(  # The Config to build data preprocessor
        type='EditDataPreprocessor', mean=[0., 0., 0.], std=[255., 255.,
                                                             255.]))
```

We refactor models in MMEdit 1.x. Important modifications are as following.

- The `models` in MMedit 1.x is refactored to five parts: `base_models`, `data_preprocessors`, `editors`, `layers` and `losses`.
- Add `data_preprocessor` module in `models`. Normalization and color space transforms operations are moved from datasets transforms pipelines to data_preprocessor. The data out from the data pipeline is transformed by this module and then fed into the model.

More details of models are shown in *model guides*.

1.57 Migration of Evaluation and Testing Settings

We update evaluation settings in MMEdit 1.x. Important modifications are as following.

- The evaluation field is split to `val_evaluator` and `test_evaluator`. The `interval` is moved to `train_cfg.val_interval`.
- The metrics to evaluation are moved from `test_cfg` to `val_evaluator` and `test_evaluator`.

```
train_cfg = None  # Training config
test_cfg = dict(  # Test config
    metrics=['PSNR'],  # Metrics used during testing
    crop_border=scale)  # Crop border during evaluation

evaluation = dict(  # The config to build the evaluation hook
    interval=5000,  # Evaluation interval
    save_image=True,  # Save images during evaluation
    gpu_collect=True)  # Use gpu collect
```

```
val_evaluator = [
    dict(type='PSNR', crop_border=scale),  # The name of metrics to evaluate
]
test_evaluator = val_evaluator

train_cfg = dict(
    type='IterBasedTrainLoop', max_iters=300000, val_interval=5000)  # Config of train␣
↪loop type
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type
```

We have merged MMGeneration 1.x into MMEditing. Here is migration of Evaluation and Testing Settings about MMGeneration.

The evaluation field is splited to `val_evaluator` and `test_evaluator`. And it won't support `interval` and `save_best` arguments. The `interval` is moved to `train_cfg.val_interval`, see *the schedule settings* and the `save_best` is moved to `default_hooks.checkpoint.save_best`.

```
evaluation = dict(
    type='GenerativeEvalHook',
    interval=10000,
    metrics=[
        dict(
            type='FID',
            num_images=50000,
            bgr2rgb=True,
            inception_args=dict(type='StyleGAN')),
        dict(type='IS', num_images=50000)
    ],
    best_metric=['fid', 'is'],
    sample_kwargs=dict(sample_model='ema'))
```

```
val_evaluator = dict(
    type='GenEvaluator',
    metrics=[
        dict(
            type='FID',
            prefix='FID-Full-50k',
            fake_nums=50000,
            inception_style='StyleGAN',
            sample_model='orig')
        dict(
            type='IS',
            prefix='IS-50k',
            fake_nums=50000)])
# set best config
default_hooks = dict(
    checkpoint=dict(
        type='CheckpointHook',
        interval=10000,
        by_epoch=False,
        less_keys=['FID-Full-50k/fid'],
        greater_keys=['IS-50k/is'],
        save_optimizer=True,
        save_best=['FID-Full-50k/fid', 'IS-50k/is'],
        rule=['less', 'greater']))
test_evaluator = val_evaluator
```

To evaluate and test the model correctly, we need to set specific loop in `val_cfg` and `test_cfg`.

```
total_iters = 1000000

runner = dict(
    type='DynamicIterBasedRunner',
    is_dynamic_ddp=False,
    pass_training_status=True)
```

```
train_cfg = dict(
    by_epoch=False,  # use iteration based training
    max_iters=1000000,  # max training iteration
```

(continues on next page)

(continued from previous page)

```
    val_begin=1,
    val_interval=10000)  # evaluation interval
val_cfg = dict(type='GenValLoop')  # specific loop in validation
test_cfg = dict(type='GenTestLoop')  # specific loop in testing
```

1.58 Migration of Schedule Settings

We update schedule settings in MMEdit 1.x. Important modifications are as following.

- Now we use `optim_wrapper` field to specify all configuration about the optimization process. And the `optimizer` is a sub field of `optim_wrapper` now.
- The `lr_config` field is removed and we use new `param_scheduler` to replace it.
- The `total_iters` field is moved to `train_cfg` as `max_iters`, `val_cfg` and `test_cfg`, which configure the loop in training, validation and test.

```
optimizers = dict(generator=dict(type='Adam', lr=1e-4, betas=(0.9, 0.999)))  # Config␣
↪used to build optimizer, support all the optimizers in PyTorch whose arguments are␣
↪also the same as those in PyTorch
total_iters = 300000 # Total training iters
lr_config = dict( # Learning rate scheduler config used to register LrUpdater hook
    policy='Step', by_epoch=False, step=[200000], gamma=0.5)  # The policy of scheduler
```

```
optim_wrapper = dict(
    dict(
        type='OptimWrapper',
        optimizer=dict(type='Adam', lr=1e-4),
    )
)  # Config used to build optimizer, support all the optimizers in PyTorch whose␣
↪arguments are also the same as those in PyTorch.
param_scheduler = dict(  # Config of learning policy
    type='MultiStepLR', by_epoch=False, milestones=[200000], gamma=0.5)  # The policy of␣
↪scheduler
train_cfg = dict(
    type='IterBasedTrainLoop', max_iters=300000, val_interval=5000)  # Config of train␣
↪loop type
val_cfg = dict(type='ValLoop')  # The name of validation loop type
test_cfg = dict(type='TestLoop')  # The name of test loop type
```

More details of schedule settings are shown in MMEngine Documents.

1.59 Migration of Data Settings

This section introduces the migration of data settings:

1.59.1 Data pipelines

We update data pipelines settings in MMEdit 1.x. Important modifications are as following.

- Remove normalization and color space transforms operations. They are moved from datasets transforms pipelines to data_preprocessor.
- The original formatting transforms pipelines `Collect` and `ToTensor` are combined as `PackEditInputs`. More details of data pipelines are shown in *transform guides*.

```
train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
        key='lq',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
        key='gt',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(type='RescaleToZeroOne', keys=['lq', 'gt']),  # Rescale images from [0, 255] to␣
↪[0, 1]
    dict(type='Normalize',  # Augmentation pipeline that normalize the input images
        keys=['lq', 'gt'],  # Images to be normalized
        mean=[0, 0, 0],  # Mean values
        std=[1, 1, 1],  # Standard variance
        to_rgb=True),  # Change to RGB channel
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='Collect',  # Pipeline that decides which keys in the data should be␣
↪passed to the model
        keys=['lq', 'gt'],  # Keys to pass to the model
        meta_keys=['lq_path', 'gt_path']), # Meta information keys. In training, meta␣
↪information is not needed
```

(continues on next page)

(continued from previous page)

```
    dict(type='ToTensor',  # Convert images to tensor
        keys=['lq', 'gt'])  # Images to be converted to Tensor
]
test_pipeline = [  # Test pipeline
    dict(
        type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
        key='lq',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(
        type='LoadImageFromFile',  # Load images from files
        io_backend='disk',  # io backend
        key='gt',  # Keys in results to find corresponding path
        flag='unchanged'),  # flag for reading images
    dict(type='RescaleToZeroOne', keys=['lq', 'gt']),  # Rescale images from [0, 255] to␣
↪[0, 1]
    dict(
        type='Normalize',  # Augmentation pipeline that normalize the input images
        keys=['lq', 'gt'],  # Images to be normalized
        mean=[0, 0, 0],  # Mean values
        std=[1, 1, 1],  # Standard variance
        to_rgb=True),  # Change to RGB channel
    dict(type='Collect',  # Pipeline that decides which keys in the data should be␣
↪passed to the model
        keys=['lq', 'gt'],  # Keys to pass to the model
        meta_keys=['lq_path', 'gt_path']),  # Meta information keys
    dict(type='ToTensor',  # Convert images to tensor
        keys=['lq', 'gt'])  # Images to be converted to Tensor
]
```

```
train_pipeline = [  # Training data processing pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='SetValues', dictionary=dict(scale=scale)),  # Set value to destination␣
↪keys
    dict(type='PairedRandomCrop', gt_patch_size=96),  # Paired random crop
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='horizontal'),  # Flip direction
    dict(type='Flip',  # Flip images
        keys=['lq', 'gt'],  # Images to be flipped
        flip_ratio=0.5,  # Flip ratio
        direction='vertical'),  # Flip direction
```

(continues on next page)

(continued from previous page)

```
    dict(type='RandomTransposeHW',  # Random transpose h and w for images
        keys=['lq', 'gt'],  # Images to be transposed
        transpose_ratio=0.5  # Transpose ratio
        ),
    dict(type='PackEditInputs')  # The config of collecting data from current pipeline
]
test_pipeline = [  # Test pipeline
    dict(type='LoadImageFromFile',  # Load images from files
        key='img',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='LoadImageFromFile',  # Load images from files
        key='gt',  # Keys in results to find corresponding path
        color_type='color',  # Color type of image
        channel_order='rgb',  # Channel order of image
        imdecode_backend='cv2'),  # decode backend
    dict(type='PackEditInputs')  # The config of collecting data from current pipeline
]
```

1.59.2 Dataloader

We update dataloader settings in MMEdit 1.x. Important modifications are as following.

- The original `data` field is split to `train_dataloader`, `val_dataloader` and `test_dataloader`. This allows us to configure them in fine-grained. For example, you can specify different sampler and batch size during training and test.
- The `samples_per_gpu` is renamed to `batch_size`.
- The `workers_per_gpu` is renamed to `num_workers`.

```
data = dict(
    # train
    samples_per_gpu=16,  # Batch size of a single GPU
    workers_per_gpu=4,  # Worker to pre-fetch data for each single GPU
    drop_last=True,  # Use drop_last in data_loader
    train=dict(  # Train dataset config
        type='RepeatDataset',  # Repeated dataset for iter-based model
        times=1000,  # Repeated times for RepeatDataset
        dataset=dict(
            type=train_dataset_type,  # Type of dataset
            lq_folder='data/DIV2K/DIV2K_train_LR_bicubic/X2_sub',  # Path for lq folder
            gt_folder='data/DIV2K/DIV2K_train_HR_sub',  # Path for gt folder
            ann_file='data/DIV2K/meta_info_DIV2K800sub_GT.txt',  # Path for annotation␣
↪file
            pipeline=train_pipeline,  # See above for train_pipeline
            scale=scale)),  # Scale factor for upsampling
    # val
    val_samples_per_gpu=1,  # Batch size of a single GPU for validation
    val_workers_per_gpu=4,  # Worker to pre-fetch data for each single GPU for validation
    val=dict(
```

(continues on next page)

(continued from previous page)

```
        type=val_dataset_type,  # Type of dataset
        lq_folder='data/val_set5/Set5_bicLRx2',  # Path for lq folder
        gt_folder='data/val_set5/Set5_mod12',  # Path for gt folder
        pipeline=test_pipeline,  # See above for test_pipeline
        scale=scale,  # Scale factor for upsampling
        filename_tmpl='{}'),  # filename template
    # test
    test=dict(
        type=val_dataset_type,  # Type of dataset
        lq_folder='data/val_set5/Set5_bicLRx2',  # Path for lq folder
        gt_folder='data/val_set5/Set5_mod12',  # Path for gt folder
        pipeline=test_pipeline,  # See above for test_pipeline
        scale=scale,  # Scale factor for upsampling
        filename_tmpl='{}'))  # filename template
```

```
dataset_type = 'BasicImageDataset'  # The type of dataset
data_root = 'data'  # Root path of data
train_dataloader = dict(
    batch_size=16,
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    sampler=dict(type='InfiniteSampler', shuffle=True),  # The type of data sampler
    dataset=dict(  # Train dataset config
        type=dataset_type,  # Type of dataset
        ann_file='meta_info_DIV2K800sub_GT.txt',  # Path of annotation file
        metainfo=dict(dataset_type='div2k', task_name='sisr'),
        data_root=data_root + '/DIV2K',  # Root path of data
        data_prefix=dict(  # Prefix of image path
            img='DIV2K_train_LR_bicubic/X2_sub', gt='DIV2K_train_HR_sub'),
        filename_tmpl=dict(img='{}', gt='{}'),  # Filename template
        pipeline=train_pipeline))
val_dataloader = dict(
    batch_size=1,
    num_workers=4,  # The number of workers to pre-fetch data for each single GPU
    persistent_workers=False,  # Whether maintain the workers Dataset instances alive
    drop_last=False,  # Whether drop the last incomplete batch
    sampler=dict(type='DefaultSampler', shuffle=False),  # The type of data sampler
    dataset=dict(  # Validation dataset config
        type=dataset_type,  # Type of dataset
        metainfo=dict(dataset_type='set5', task_name='sisr'),
        data_root=data_root + '/Set5',  # Root path of data
        data_prefix=dict(img='LRbicx2', gt='GTmod12'),  # Prefix of image path
        pipeline=test_pipeline))
test_dataloader = val_dataloader
```

1.60 Migration of Distributed Training Settings

We have merged MMGeneration 1.x into MMEditing. Here is migration of Distributed Training Settings about MMGeneration.

In 0.x version, MMGeneration uses `DDPWrapper` and `DynamicRunner` to train static and dynamic model (e.g., PGGAN and StyleGANv2) respectively. In 1.x version, we use `MMSeparateDistributedDataParallel` provided by MMEngine to implement distributed training.

The configuration differences are shown below:

```
# Use DDPWrapper
use_ddp_wrapper = True
find_unused_parameters = False

runner = dict(
    type='DynamicIterBasedRunner',
    is_dynamic_ddp=False)
```

```
model_wrapper_cfg = dict(
    type='MMSeparateDistributedDataParallel',
    broadcast_buffers=False,
    find_unused_parameters=False)
```

```
use_ddp_wrapper = False
find_unused_parameters = False

# Use DynamicRunner
runner = dict(
    type='DynamicIterBasedRunner',
    is_dynamic_ddp=True)
```

```
model_wrapper_cfg = dict(
    type='MMSeparateDistributedDataParallel',
    broadcast_buffers=False,
    find_unused_parameters=True) # set `find_unused_parameters` for dynamic models
```

1.61 Migration of Optimizers

We have merged MMGeneration 1.x into MMEditing. Here is migration of Optimizers about MMGeneration.

In version 0.x, MMGeneration uses PyTorch's native Optimizer, which only provides general parameter optimization. In version 1.x, we use `OptimizerWrapper` provided by MMEngine.

Compared to PyTorch's `Optimizer`, `OptimizerWrapper` supports the following features:

- `OptimizerWrapper.update_params` implement `zero_grad`, `backward` and `step` in a single function.
- Support gradient accumulation automatically.
- Provide a context manager named `OptimizerWrapper.optim_context` to warp the forward process. `optim_context` can automatically call `torch.no_sync` according to current number of updating iteration. In AMP (auto mixed precision) training, `autocast` is called in `optim_context` as well.

For GAN models, generator and discriminator use different optimizer and training schedule. To ensure that the GAN model's function signature of `train_step` is consistent with other models, we use `OptimWrapperDict`, inherited from `OptimizerWrapper`, to wrap the optimizer of the generator and discriminator. To automate this process MMGeneration implement `GenOptimWrapperContructor`. And you should specify this constructor in your config is you want to train GAN model.

The config for the 0.x and 1.x versions are shown below:

```
optimizer = dict(
    generator=dict(type='Adam', lr=0.0001, betas=(0.0, 0.999), eps=1e-6),
    discriminator=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-6))
```

```
optim_wrapper = dict(
    # Use constructor implemented by MMGeneration
    constructor='GenOptimWrapperConstructor',
    generator=dict(optimizer=dict(type='Adam', lr=0.0002, betas=(0.0, 0.999), eps=1e-6)),
    discriminator=dict(
        optimizer=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-6)))
```

Note that, in the 1.x, MMGeneration uses `OptimWrapper` to realize gradient accumulation. This make the config of `discriminator_steps` (training trick for updating the generator once after multiple updates of the discriminator) and gradient accumulation different between 0.x and 1.x version.

- In 0.x version, we use `disc_steps`, `gen_steps` and `batch_accumulation_steps` in configs. `disc_steps` and `batch_accumulation_steps` are counted by the number of calls of `train_step` (is also the number of data reads from the dataloader). Therefore the number of consecutive updates of the discriminator is `disc_steps // batch_accumulation_steps`. And for generators, `gen_steps` is the number of times the generator actually updates continuously.
- In 1.x version, we use `discriminator_steps`, `generator_steps` and `accumulative_counts` in configs. `discriminator_steps` and `generator_steps` are the number of consecutive updates to itself before updating other modules.

Take config of BigGAN-128 as example.

```
model = dict(
    type='BasiccGAN',
    generator=dict(
        type='BigGANGenerator',
        output_scale=128,
        noise_size=120,
        num_classes=1000,
        base_channels=96,
        shared_dim=128,
        with_shared_embedding=True,
        sn_eps=1e-6,
        init_type='ortho',
        act_cfg=dict(type='ReLU', inplace=True),
        split_noise=True,
        auto_sync_bn=False),
    discriminator=dict(
        type='BigGANDiscriminator',
        input_scale=128,
        num_classes=1000,
        base_channels=96,
```

(continues on next page)

(continued from previous page)

```
        sn_eps=1e-6,
        init_type='ortho',
        act_cfg=dict(type='ReLU', inplace=True),
        with_spectral_norm=True),
    gan_loss=dict(type='GANLoss', gan_type='hinge'))

# continuous update discriminator for `disc_steps // batch_accumulation_steps = 8 // 8 =␣
↪1` times
# continuous update generator for `gen_steps = 1` times
# generators and discriminators perform `batch_accumulation_steps = 8` times gradient␣
↪accumulations before each update
train_cfg = dict(
    disc_steps=8, gen_steps=1, batch_accumulation_steps=8, use_ema=True)
```

```
model = dict(
    type='BigGAN',
    num_classes=1000,
    data_preprocessor=dict(type='GANDataPreprocessor'),
    generator=dict(
        type='BigGANGenerator',
        output_scale=128,
        noise_size=120,
        num_classes=1000,
        base_channels=96,
        shared_dim=128,
        with_shared_embedding=True,
        sn_eps=1e-6,
        init_type='ortho',
        act_cfg=dict(type='ReLU', inplace=True),
        split_noise=True,
        auto_sync_bn=False),
    discriminator=dict(
        type='BigGANDiscriminator',
        input_scale=128,
        num_classes=1000,
        base_channels=96,
        sn_eps=1e-6,
        init_type='ortho',
        act_cfg=dict(type='ReLU', inplace=True),
        with_spectral_norm=True),
    # continuous update discriminator for `discriminator_steps = 1` times
    # continuous update generator for `generator_steps = 1` times
    generator_steps=1,
    discriminator_steps=1)

optim_wrapper = dict(
    constructor='GenOptimWrapperConstructor',
    generator=dict(
        # generator perform `accumulative_counts = 8` times gradient accumulations before␣
↪each update
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0001, betas=(0.0, 0.999), eps=1e-6)),
```

(continues on next page)

(continued from previous page)

```
    discriminator=dict(
        # discriminator perform `accumulative_counts = 8` times gradient accumulations␣
↪before each update
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-6)))
```

1.62 Migration of Visualization

In 0.x, MMEditing use `VisualizationHook` to visualize results in training process. In 1.x version, we unify the function of those hooks into `BasicVisualizationHook` / `GenVisualizationHook`. Additionally, follow the design of MMEngine, we implement `ConcatImageVisualizer` / `GenVisualizer` and a group of `VisBackend` to draw and save the visualization results.

```
visual_config = dict(
    type='VisualizationHook',
    output_dir='visual',
    interval=1000,
    res_name_list=['gt_img', 'masked_img', 'fake_res', 'fake_img'],
)
```

```
vis_backends = [dict(type='LocalVisBackend')]
visualizer = dict(
    type='ConcatImageVisualizer',
    vis_backends=vis_backends,
    fn_key='gt_path',
    img_keys=['gt_img', 'input', 'pred_img'],
    bgr2rgb=True)
custom_hooks = [dict(type='BasicVisualizationHook', interval=1)]
```

To learn more about the visualization function, please refers to *this tutorial*.

1.63 Migration of AMP Training

In 0.x, MMEditing do not support AMP training for the entire forward process. Instead, users must use `auto_fp16` decorator to warp the specific submodule and convert the parameter of submodule to fp16. This allows for fine-grained control of the model parameters, but is more cumbersome to use. In addition, users need to handle operations such as scaling of the loss function during the training process by themselves.

In 1.x version, MMEditing use `AmpOptimWrapper` provided by MMEngine. In `AmpOptimWrapper.update_params`, gradient scaling and `GradScaler` updating is automatically performed. And in `optim_context` context manager, `auto_cast` is applied to the entire forward process.

Specifically, the difference between the 0.x and 1.x is as follows:

```
# config
runner = dict(fp16_loss_scaler=dict(init_scale=512))
```

```
# code
import torch.nn as nn
```

(continues on next page)

(continued from previous page)

```
from mmedit.models.builder import build_model
from mmedit.core.runners.fp16_utils import auto_fp16


class DemoModule(nn.Module):
    def __init__(self, cfg):
        self.net = build_model(cfg)

    @auto_fp16
    def forward(self, x):
        return self.net(x)

class DemoModel(nn.Module):

    def __init__(self, cfg):
        super().__init__(self)
        self.demo_network = DemoModule(cfg)

    def train_step(self,
                   data_batch,
                   optimizer,
                   ddp_reducer=None,
                   loss_scaler=None,
                   use_apex_amp=False,
                   running_status=None):
        # get data from data_batch
        inputs = data_batch['img']
        output = self.demo_network(inputs)

        optimizer.zero_grad()
        loss, log_vars = self.get_loss(data_dict_)

        if ddp_reducer is not None:
            ddp_reducer.prepare_for_backward(_find_tensors(loss_disc))

        if loss_scaler:
            # add support for fp16
            loss_scaler.scale(loss_disc).backward()
        elif use_apex_amp:
            from apex import amp
            with amp.scale_loss(loss_disc, optimizer,
                    loss_id=0) as scaled_loss_disc:
                scaled_loss_disc.backward()
        else:
            loss_disc.backward()

        if loss_scaler:
            loss_scaler.unscale_(optimizer)
            loss_scaler.step(optimizer)
        else:
            optimizer.step()
```

```
# config
optim_wrapper = dict(
    constructor='OptimWrapperConstructor',
    generator=dict(
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0001, betas=(0.0, 0.999), eps=1e-06),
        type='AmpOptimWrapper',  # use amp wrapper
        loss_scale='dynamic'),
    discriminator=dict(
        accumulative_counts=8,
        optimizer=dict(type='Adam', lr=0.0004, betas=(0.0, 0.999), eps=1e-06),
        type='AmpOptimWrapper',  # use amp wrapper
        loss_scale='dynamic'))
```

```
# code
import torch.nn as nn
from mmedit.registry import MODULES
from mmengine.model import BaseModel


class DemoModule(nn.Module):
    def __init__(self, cfg):
        self.net = MODULES.build(cfg)

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

class DemoModel(BaseModel):
    def __init__(self, cfg):
        super().__init__(self)
        self.demo_network = DemoModule(cfg)

    def train_step(self, data, optim_wrapper):
        # get data from data_batch
        data = self.data_preprocessor(data, True)
        inputs = data['inputs']

        with optim_wrapper.optim_context(self.discriminator):
            output = self.demo_network(inputs)
        loss_dict = self.get_loss(output)
        # use parse_loss provide by `BaseModel`
        loss, log_vars = self.parse_loss(loss_dict)
        optimizer_wrapper.update_params(loss)

        return log_vars
```

To avoid user modifications to the configuration file, MMEditing provides the `--amp` option in `train.py`, which allows the user to start AMP training without modifying the configuration file. Users can start AMP training by following command:

```
bash tools/dist_train.sh CONFIG GPUS --amp

# for slurm users
```

(continues on next page)

(continued from previous page)

```
bash tools/slurm_train.sh PARTITION JOB_NAME CONFIG WORK_DIR --amp
```

1.64 English

1.65

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

R

S

T

U

V

W