mmflow

MMFlow Author

Nov 14, 2022
# LEARN THE BASICS

1 Learn the Basics .......................................................... 1
2 Prerequisites ................................................................. 3
3 Installation ................................................................. 5
4 Get Started ................................................................. 9
5 Model Zoo Statistics ...................................................... 13
6 Tutorial 0: Learn about Configs ..................................... 15
7 Tutorial 1: Inference with existing models ..................... 21
8 Tutorial 2: Finetuning Models ....................................... 25
9 Tutorial 3: Custom Data Pipelines ............................... 29
10 Tutorial 4: Adding New Modules ................................. 33
11 Tutorial 5: Customize Runtime Settings ....................... 37
12 Conventions .............................................................. 45
13 Changelog ................................................................. 49
14 Frequently Asked Questions ........................................ 55
15 mmflow.apis .............................................................. 57
16 mmflow.core .............................................................. 59
17 mmflow.datasets ........................................................ 65
18 mmflow.models .......................................................... 83
19 Indices and tables ....................................................... 85
Python Module Index ..................................................... 87
Index .............................................................................. 89
LEARN THE BASICS

This chapter introduces you to the basic conception of optical flow, and the framework of MMFlow, and provides links to detailed tutorials about MMFlow.

1.1 What is Optical flow estimation

Optical flow is a 2D velocity field, representing the apparent 2D image motion of pixels from the reference image to the target image [1]. The task can be defined as follows: Given two images img1, img2 \( RH \times W \times 3 \), the flow field \( U \in RH \times W \times 2 \) describes the horizontal and vertical image motion between img1 and img2 [2]. Here is an example for visualized flow map from Sintel dataset [3-4]. The character in origin images moves left, so the motion raises the optical flow, and referring to the color wheel whose color represents the direction on the right, the left flow can be rendered as blue.

Note that optical flow only focuses on images, and is not relative to the projection of the 3D motion of points in the scene onto the image plane.

One may ask, “What about the motion of a smooth surface like a smooth rotating sphere?”

If the surface of the sphere is untextured then there will be no apparent motion on the image plane and hence no optical flow [2]. It illustrates that the motion field [5], corresponding to the motion of points in the scene, is not always the same as the optical flow field. However, for most applications of optical flow, it is the motion field that is required and, typically, the world has enough structure so that optical flow provides a good approximation to the motion field [2]. As long as the optical flow field provides a reasonable approximation, it can be considered as a strong hint of sequential frames and is used in a variety of situations, e.g., action recognition, autonomous driving, and video editing [6].

The metrics to compare the performance of the optical flow methods are \( EPE \), EndPoint Error over the complete frames, and \( Fl-all \), percentage of outliers averaged over all pixels, that inliers are defined as \( EPE < 3 \) pixels or < 5%.

The mainstream benchmark datasets are Sintel for dense optical flow and KITTI [7-9] for sparse optical flow.

1.2 What is MMFlow

MMFlow is the first toolbox that provides a framework for unified implementation and evaluation of optical flow methods, and below is its whole framework:

MMFlow consists of 4 main parts, datasets, models, core and apis.

- datasets is for datasets loading and data augmentation. In this part, we support various datasets for supervised optical flow algorithms, useful data augmentation transforms in pipelines for pre-processing image pairs and flow data (including its auxiliary data), and samplers for data loading in samplers.
• **models** is the most vital part containing models of learning-based optical flow. As you can see, we implement each model as a flow estimator and decompose it into two components encoder and decoder. The loss functions for flow models training are in this module as well.

• **core** provides evaluation tools and customized hooks for model training.

• **apis**, provides high-level APIs for models training, testing, and inference.

### 1.3 How to Use this Guide

Here is a detailed step-by-step guide to learn more about MMFlow:

1. For installation instructions, please see *install*.
2. *get_started* is for the basic usage of MMFlow.
3. Refer to the below tutorials to dive deeper:
   - config
   - model inference
   - fine tuning
   - data pipeline
   - add new modules
   - customized runtime

### 1.4 References

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

MMFlow works on Linux, Windows and macOS. It requires Python 3.6+, CUDA 9.2+ and PyTorch 1.5+.

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

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

**Step 1.** Create a conda environment and activate it.

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

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

On GPU platforms:

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

On CPU platforms:

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

INSTALLATION

We recommend that users follow our best practices to install MMFlow. However, the whole process is highly customiz-
able. See Customize Installation section for more information.

3.1 Best Practices

Step 0. Install MMCV using MIM.

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

Step 1. Install MMFlow.

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

```bash
git clone https://github.com/open-mmlab/mmflow.git
cd mmflow
pip install -v -e .
```

# "-v" means verbose, or more output
# "-e" means installing a project in editable mode,
# thus any local modifications made to the code will take effect without reinstallation.

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

```bash
pip install mmflow
```

3.2 Verify the installation

To verify whether MMFlow is installed correctly, we provide some sample codes to run an inference demo.

Step 1. We need to download config and checkpoint files.

```bash
mim download mmflow --config pwcnet_ft_4x1_300k_sintel_final_384x768
```

The downloading will take several seconds or more, depending on your network environment. When it is done, you will find two files `pwcnet_ft_4x1_300k_sintel_final_384x768.py` and `pwcnet_ft_4x1_300k_sintel_final_384x768.pth` in your current folder.

Step 2. Verify the inference demo.

Option (a). If you install mmflow from source, just run the following command.
Output will be saved in the directory results including a rendered flow map flow.png and flow file flow.flo
Option (b). If you install mmflow with pip, open you python interpreter and copy&paste the following codes.

```python
from mmflow.apis import inference_model, init_model

config_file = 'pwcnet_ft_4x1_300k_sintel_final_384x768.py'
checkpoint_file = 'pwcnet_ft_4x1_300k_sintel_final_384x768.pth'
device = 'cuda:0'

# init a model
model = init_model(config_file, checkpoint_file, device=device)

# inference the demo image
inference_model(model, 'demo/frame_0001.png', 'demo/frame_0002.png')
```

You will see a array printed, which is the flow data.

### 3.3 Customize Installation

#### 3.3.1 CUDA versions

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

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

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

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

#### 3.3.2 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.

```bash
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.10/index._html
```
3.3.3 Install on CPU-only platforms

MMFlow can be built for CPU only environment. In CPU mode you can train (requires MMCV version >= 1.4.4), test or inference a model.

However some functionalities are gone in this mode:

- Correlation

If you try to train/test/inference a model containing above ops, an error will be raised. The following table lists affected algorithms.

3.3.4 Install on Google Colab

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

**Step 1.** Install MMCV using MIM.

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

**Step 2.** Install MMFlow from the source.

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

**Step 3.** Verification.

```
import mmflow
print(mmflow.__version__)
# Example output: 0.4.1
```

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

3.3.5 Using MMFlow with Docker

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

```
# build an image with PyTorch 1.6, CUDA 10.1
# If you prefer other versions, just modified the Dockerfile
docker build -t mmflow docker/
```

Run it with

```
docker run --gpus all -- shm-size=8g -it -v {DATA_DIR}:/mmflow/data mmflow
```
3.4 Trouble shooting

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

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

- **Get Started**
  - Prepare datasets
  - **Inference with Pre-trained Models**
    - Run a demo
    - Test a dataset
  - Train a model
  - Tutorials

### 4.1 Prepare datasets

It is recommended to symlink the dataset root to `$MMFlow/data`. Please follow the corresponding guidelines for data preparation.

- FlyingChairs
- FlyingThings3d_subset
- FlyingThings3d
- Sintel
- KITTI2015
- KITTI2012
- FlyingChairsOcc
- ChairsSDHom
- HD1K
4.2 Inference with Pre-trained Models

We provide testing scripts to evaluate a whole dataset (Sintel, KITTI2015, etc.), and provide some high-level APIs and scripts to estimate flow for images or a video easily.

4.2.1 Run a demo

We provide scripts to run demos. Here is an example to predict the optical flow between two adjacent frames.

1. image demo

```python
demo/demo/image_demo.py ${IMAGE1} ${IMAGE2} ${CONFIG_FILE} ${CHECKPOINT_FILE} $ →{OUTPUT_DIR} \n   [--out_prefix] ${OUTPUT_PREFIX} [--device] ${DEVICE}
```

Optional arguments:
- `--out_prefix`: The prefix for the output results including flow file and visualized flow map.
- `--device`: Device used for inference.

Example:
Assume that you have already downloaded the checkpoints to the directory checkpoints/, and output will be saved in the directory raft_demo.

```bash
demo/demo/image_demo.py demo/frame_0001.png demo/frame_0002.png \n   configs/raft/raft_8x2_100k_mixed_368x768.py \n   checkpoints/raft_8x2_100k_mixed_368x768.pth raft_demo
```

2. video demo

```python
demo/demo/video_demo.py ${VIDEO} ${CONFIG_FILE} ${CHECKPOINT_FILE} ${OUTPUT_FILE} \n   →\n   [--gt] ${GROUND_TRUTH} [--device] ${DEVICE}
```

Optional arguments:
- `--gt`: The video file of ground truth for input video. If specified, the ground truth will be concatenated predicted result as a comparison.
- `--device`: Device used for inference.

Example:
Assume that you have already downloaded the checkpoints to the directory checkpoints/, and output will be save as raft_demo.mp4.

```bash
demo/demo/video_demo.py demo/demo.mp4 \n   configs/raft/raft_8x2_100k_mixed_368x768.py \n   checkpoints/raft_8x2_100k_mixed_368x768.pth \n   raft_demo.mp4 --gt demo/demo_gt.mp4
```
4.2.2 Test a dataset

You can use the following commands to test a dataset, and more information is in tutorials/1_inference.

```plaintext
# single-gpu testing
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments]
```

Optional arguments:

- `--out_dir`: Directory to save the output results. If not specified, the flow files will not be saved.
- `--fuse-conv-bn`: Whether to fuse conv and bn, this will slightly increase the inference speed.
- `--show_dir`: Directory to save the visualized flow maps. If not specified, the flow maps will not be saved.
- `--eval`: Evaluation metrics, e.g., “EPE”.
- `--cfg-option`: Override some settings in the used config, the key-value pair in xxx=yyy format will be merged into config file. For example, ‘`--cfg-option model.encoder.in_channels=6`’.

Examples:

Assume that you have already downloaded the checkpoints to the directory checkpoints/.

Test PWC-Net on Sintel clean and final sub-datasets without saving predicted flow files and evaluating the EPE.

```plaintext
python tools/test.py configs/pwcnet/pwcnet_ft_4x1_300k_sintel_384x768.py checkpoints/pwcnet_8x1_sfine_sintel_384x768.pth --eval EPE
```

4.3 Train a model

You can use the train script to launch training task with a single GPU, and more information in tutorials/2_finetune

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

Optional arguments:

- `--work-dir`: Override the working directory specified in the config file.
- `--load-from`: The checkpoint file to load weights from.
- `--resume-from`: Resume from a previous checkpoint file.
- `--no-validate`: Whether not to evaluate the checkpoint during training.
- `--seed`: Seed id for random state in python, numpy and pytorch to generate random numbers.
- `--deterministic`: If specified, it will set deterministic options for CUDNN backend.
- `--cfg-options`: Override some settings in the used config, the key-value pair in xxx=yyy format will be merged into config file. For example, ‘`--cfg-option model.encoder.in_channels=6`’.

Difference between `resume-from` and `load-from`: `resume-from` loads both the model weights and optimizer status, and the epoch/iter is also inherited from the specified checkpoint. It is usually used for resuming the training process that is interrupted accidentally. `load-from` only loads the model weights and the training epoch/iter starts from 0. It is usually used for finetuning.

Here is an example to train PWC-Net.

```plaintext
python tools/train.py configs/pwcnet/pwcnet_ft_4x1_300k_sintel_384x768.py --work-dir work_dir/pwcnet
```
4.4 Tutorials

We provide some tutorials for users:

• learn about configs
• inference model
• finetune model
• customize data pipelines
• add new modules
• customize runtime settings.
CHAPTER
FIVE

MODEL ZOO STATISTICS

- Number of papers: 9
- Number of checkpoints: 62 ckpts
  - FlowNet: Learning Optical Flow with Convolutional Networks (5 ckpts)
  - FlowNet 2.0: Evolution of Optical Flow Estimation with Deep Networks (7 ckpts)
  - LiteFlowNet: A Lightweight Convolutional Neural Network for Optical Flow Estimation (9 ckpts)
  - A Lightweight Optical Flow CNN-Revisiting Data Fidelity and Regularization (8 ckpts)
  - Iterative Residual Refinement for Joint Optical Flow and Occlusion Estimation (5 ckpts)
  - MaskFlownet: Asymmetric Feature Matching with Learnable Occlusion Mask (4 ckpts)
  - RAFT: Recurrent All-Pairs Field Transforms for Optical Flow (5 ckpts)
  - GMA: Learning to Estimate Hidden Motions with Global Motion Aggregation (13 ckpts)
CHAPTER SIX

TUTORIAL 0: LEARN ABOUT CONFIGS

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

6.1 Config File Structure

There are 4 basic component types under `config/_base_`, datasets, models, schedules, default_runtime. Many methods could be easily constructed with one of each like PWC-Net. The configs that are composed by 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 PWC-Net, user may first inherit the basic PWC-Net structure by specifying `_base_ = ../pwcnet/pwcnet_slong_8x1_flyingchairs_384x448.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 mmcv for detailed documentation.

6.2 Config File Naming Convention

We follow the below style to name config files. Contributors are advised to follow the same style.

```
{model}_{schedule}_{gpu x batch_per_gpu}_{training datasets}_{input_size}.py
```

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

- `{model}`: model type like `pwcnet`, `flownets`, etc.
- `{schedule}`: training schedule. Following FlowNet2’s convention, we use `slong`, `sfine` and `sshort`, or number of iteration like `150k` `150k(iterations)``.
- `{gpu x batch_per_gpu}`: GPUs and samples per GPU, like `8x1`.
- `{training datasets}`: training dataset like `flyingchairs`, `flyingthings3d_subset`, `flyingthings3d`.
- `{input_size}`: the size of training images.
6.3 Config System

To help the users have a basic idea of a complete config and the modules in MMFlow, we make brief comments on the config of PWC-Net trained on FlyingChairs with slong schedule. For more detailed usage and the corresponding alternative for each module, please refer to the API documentation and the tutorial in MMDetection.

```python
_base_ = ['../_base_/models/pwcnet.py', '../_base_/datasets/flyingchairs_384x448.py', '../_base_/schedules/schedule_s_long.py', '../_base_/default_runtime.py']

# base config file which we build new config file on.
_base_/models/pwc_net.py is a basic model cfg file for PWC-Net.

model = dict(
    type='PWCNet',  # The algorithm name
    encoder=dict(  # Encoder module config
        type='PWCNetEncoder',  # The name of encoder in PWC-Net.
        in_channels=3,  # The input channels
        # The type of this sub-module, if net_type is Basic, the the number of
        # convolution layers of each level is 3,
        # if net_type is Small, the the number of convolution layers of each level is 2.
        net_type='Basic',
        pyramid_levels=[
            'level1', 'level2', 'level3', 'level4', 'level5', 'level6'
        ],  # The list of feature pyramid levels that are the keys for output dict.
        out_channels=(16, 32, 64, 96, 128, 196),  # List of numbers of output channels
        strides=(2, 2, 2, 2, 2),  # List of strides of each pyramid level.
        dilations=(1, 1, 1, 1, 1),  # List of dilation of each pyramid level.
        act_cfg=dict(type='LeakyReLU', negative_slope=0.1)),  # Config dict for each
        # activation layer in ConvModule.
        decoder=dict(  # Decoder module config.
            type='PWCNetDecoder',  # The name of flow decoder in PWC-Net.
            in_channels=dict(  # Input
                level6=81, level5=213, level4=181, level3=149, level2=117),
            # channels of basic dense block.
            flow_div=20.,  # The constant divisor to scale the ground truth value.
            corr_cfg=dict(type='Correlation', max_displacement=4, padding=0),
            warp_cfg=dict(type='Warp'),
            act_cfg=dict(type='LeakyReLU', negative_slope=0.1),
            scaled=False,  # Whether to use scaled correlation by the number of elements.
            post_processor=dict(type='ContextNet', in_channels=565),  # The configuration
            # for post processor.
            flow_loss=dict(  # The loss function configuration.
                type='MultiLevelEPE',
                p=2,
                reduction='sum',
                weights={  # The weights for different levels of flow.
                    'level2': 0.005,
                    'level3': 0.01,
                    'level4': 0.02,
                    'level5': 0.08,
                }
            )
        )
)
```

(continues on next page)
'level6': 0.32
},

# model training and testing settings
train_cfg = dict(),
test_cfg = dict(),
init_cfg = dict(
    type='Kaiming',
    nonlinearity='leaky_relu',
    layer=['Conv2d', 'ConvTranspose2d'],
    mode='fan_in',
    bias=0)

in _base_/datasets/flyingchairs_384x448.py

dataset_type = 'FlyingChairs'  # Dataset name
data_root = 'data/FlyingChairs/data'  # Root path of dataset

img_norm_cfg = dict(mean=[0., 0., 0.], std=[255., 255., 255], to_rgb=False)  # Image normalization config to normalize the input images

train_pipeline = [
    dict(type='LoadImageFromFile'),  # load images
    dict(type='LoadAnnotations'),  # load flow data
    dict(type='ColorJitter',  # Randomly change the brightness, contrast, saturation and hue of an image.
        brightness=0.5,  # How much to jitter brightness.
        contrast=0.5,  # How much to jitter contrast.
        saturation=0.5,  # How much to jitter saturation.
        hue=0.5),  # How much to jitter hue.
    dict(type='RandomGamma', gamma_range=(0.7, 1.5)),  # Randomly gamma correction on images.
    dict(type='Normalize', **img_norm_cfg),  # Normalization config, the values are from img_norm_cfg
    dict(type='GaussianNoise', sigma_range=(0, 0.04), clamp_range=(0., 1.)),  # Add Gaussian noise and a sigma uniformly sampled from [0, 0.04];
    dict(type='RandomFlip', prob=0.5, direction='horizontal'),  # Random horizontal flip
    dict(type='RandomFlip', prob=0.5, direction='vertical'),  # Random vertical flip
    # Random affine transformation of images
    # Keys of global_transform and relative_transform should be the subset of ('translates', 'zoom', 'shear', 'rotate'). And also, each key and its corresponding values has to satisfy the following rules:
    # - translates: the translation ratios along x axis and y axis. Defaults to(0., 0.).
    # - zoom: the min and max zoom ratios. Defaults to (1.0, 1.0).
    # - shear: the min and max shear ratios. Defaults to (1.0, 1.0).
    # - rotate: the min and max rotate degree. Defaults to (0., 0.).
    dict(type='RandomAffine',
        global_transform=dict(
            translates=(0.05, 0.05),
            zoom=(1.0, 1.5),
            shear=(0.86, 1.16),
        )
    )

(continues on next page)
rotate=(-10., 10.),
relative_transform=dict(
    translates=(0.00375, 0.00375),
    zoom=(0.985, 1.015),
    shear=(1.0, 1.0),
    rotate=(-1.0, 1.0)
)),

dict(type='RandomCrop', crop_size=(384, 448)),  # Random crop the image and flow as (384, 448)
dict(type='DefaultFormatBundle'),  # It simplifies the pipeline of formatting common fields, including "img1", "img2" and "flow_gt".
dict(type='Collect', keys=['imgs'],  # Collect data from the loader relevant to the specific task.
    meta_keys=('img_fields', 'ann_fields', 'filename1', 'filename2',
        'ori_filename1', 'ori_filename2', 'filename_flow',
        'ori_filename_flow', 'ori_shape', 'img_shape',
        'img_norm_cfg')),
]

test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations'),
    dict(type='InputResize', exponent=4),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='TestFormatBundle'),  # It simplifies the pipeline of formatting common fields, including "img1" and "img2".
    dict(type='Collect',
        keys=['imgs'],  # Collect data from the loader relevant to the specific task.
        meta_keys=('flow_gt', 'filename', 'filename2', 'ori_filename',
            'ori_filename2', 'ori_shape', 'img_shape', 'img_norm_cfg',
            'scale_factor', 'pad_shape'))  # 'flow_gt' in img_meta is works for online evaluation.
]
data = dict(
    train_dataloader=dict(
        samples_per_gpu=1,  # Batch size of a single GPU
        workers_per_gpu=5,  # Worker to pre-fetch data for each single GPU
        drop_last=True),  # Drops the last non-full batch
    val_dataloader=dict(
        samples_per_gpu=1,  # Batch size of a single GPU
        workers_per_gpu=2,  # Worker to pre-fetch data for each single GPU
        shuffle=False),  # Whether shuffle dataset.
    test_dataloader=dict(
        samples_per_gpu=1,  # Batch size of a single GPU
        workers_per_gpu=2,  # Worker to pre-fetch data for each single GPU
        )
shuffle=False, # Whether shuffle dataset.

train=dict( # Train dataset config
type=dataset_type,
pipe=ne=5un_train_pipeline,
data_root=data_root,
split_file='data/FlyingChairs_release/FlyingChairs_train_val.txt', # train-
validation split file
),

val=dict(
type=dataset_type,
pipeline=test_pipeline,
data_root=data_root,
test_mode=True),

test=dict(
type=dataset_type,
pipeline=test_pipeline,
data_root=data_root,
test_mode=True)
)

in _base_/schedules/schedule_s_long.py

# optimizer
optimizer = dict(
type='Adam', lr=0.0001, weight_decay=0.0004, betas=(0.9, 0.999))

optimizer_config = dict(grad_clip=None)

# learning policy
lr_config = dict(
policy='step',
by_epoch=False,
gamma=0.5,
step=[400000, 600000, 800000, 1000000])

runner = dict(type='IterBasedRunner', max_iters=1200000)

checkpoint_config = dict(by_epoch=False, interval=100000)

evaluation = dict(interval=100000, metric='EPE')

in _base_/default_runtime.py

log_config = dict( # config to register logger hook
interval=50, # Interval to print the log
hooks=[

dict(type='TextLoggerHook'),

dict(type='TensorboardLoggerHook')
], # The logger used to record the training process.
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.
6.4 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-option model.encoder.in_channels=6.
- Update keys inside a list of configs.
  Some config dicts are composed as a list in your config. For example, the training pipeline data.train.pipeline is normally a list e.g. [dict(type='LoadImageFromFile'), ...]. If you want to change 'LoadImageFromFile' to 'LoadImageFromWebcam' in the pipeline, you may specify --cfg-options data.train.pipeline.0.type=LoadImageFromWebcam.
- Update values of list/tuples.
  If the value to be updated is a list or a tuple. For example, the config file normally sets workflow=[('train', 1)]. If you want to change this key, you may specify --cfg-options workflow="[('train',1),('val',1)]". Note that the quotation mark ” is necessary to support list/tuple data types, and that NO white space is allowed inside the quotation marks in the specified value.

6.5 FAQ

6.5.1 Ignore some fields in the base configs

Sometimes, you may set _delete_=True to ignore some of fields in base configs. You may refer to mmcv for simple illustration.

You may have a careful look at this tutorial for better understanding of this feature.

6.5.2 Use intermediate variables in configs

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. An intuitive example can be found in this tutorial.
TUTORIAL 1: INFEERENCE WITH EXISTING MODELS

MMFlow provides pre-trained models for flow estimation in Model Zoo, and supports multiple standard datasets, including FlyingChairs, Sintel, etc. This note will show how to perform common tasks on these existing models and standard datasets, including:

- Use existing models to inference on given images.
- Test existing models on standard datasets.

7.1 Inference on given images

MMFlow provides high-level Python APIs for inference on images. Here is an example of building the model and inference on given images.

```python
from mmflow.apis import init_model, inference_model
from mmflow.datasets import visualize_flow, write_flow
import mmcv

# Specify the path to model config and checkpoint file
config_file = 'configs/pwcnet/pwcnet_8x1_slong_flyingchairs_384x448.py'
checkpoint_file = 'checkpoints/pwcnet_8x1_slong_flyingchairs_384x448.pth'

# build the model from a config file and a checkpoint file
model = init_model(config_file, checkpoint_file, device='cuda:0')

# test image pair, and save the results
img1='demo/frame_0001.png'
img2='demo/frame_0002.png'
result = inference_model(model, img1, img2)

# save the optical flow file
write_flow(result, flow_file='flow.flo')

# save the visualized flow map
flow_map = visualize_flow(result, save_file='flow_map.png')
```

An image demo can be found in demo/image_demo.py.
7.2 Evaluate existing models on standard datasets

7.2.1 Test existing models

We provide testing scripts for evaluating an existing model on the whole dataset. The following testing environments are supported:

- single GPU
- CPU
- single node multiple GPUs
- multiple nodes

Choose the proper script to perform testing depending on the testing environment.

```python
# single-gpu testing
python tools/test.py \\
  ${CONFIG_FILE} \\
  ${CHECKPOINT_FILE} \\
  [--eval ${EVAL_METRICS}] \\
  [--out-dir ${OUTPUT_DIRECTORY}] \\
  [--show-dir ${VISUALIZATION_DIRECTORY}]

# CPU: disable GPUs and run single-gpu testing script
export CUDA_VISIBLE_DEVICES=-1
python tools/test.py \\
  ${CONFIG_FILE} \\
  ${CHECKPOINT_FILE} \\
  [--out ${RESULT_FILE}] \\
  [--eval ${EVAL_METRICS}] \\
  [--show]

# multi-gpu testing
bash tools/dist_test.sh \\
  ${CONFIG_FILE} \\
  ${CHECKPOINT_FILE} \\
  ${GPU_NUM} \\
  [--eval ${EVAL_METRICS}] \\
  [--out-dir ${OUTPUT_DIRECTORY}] 
```

`tools/dist_test.sh` also supports multi-node testing, but relies on PyTorch’s launch utility.

`Slurm` is a good job scheduling system for computing clusters. On a cluster managed by Slurm, you can use `slurm_test.sh` to spawn testing jobs. It supports both single-node and multi-node testing.

```bash
[GPUS=${GPUS}] ./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} \\
  ${CONFIG_FILE} ${CHECKPOINT_FILE} \\
  [--eval ${EVAL_METRICS}] \\
  [--out-dir ${OUTPUT_DIRECTORY}]
```

Optional arguments:

- `--eval`: Evaluation metrics, e.g., “EPE”.
- `--fuse-conv-bn`: Whether to fuse conv and bn, this will slightly increase the inference speed.
• **--out-dir**: If specified, predicted optical flow will be saved in this directory.

• **--show-dir**: If specified, the visualized optical flow map will be saved in this directory.

• **--cfg-options**: If specified, the key-value pair optional cfg will be merged into config file. For example, `--cfg-option model.encoder.in_channels=6`.

Below is the optional arguments for multi-gpu test:

• **--gpu_collect**: If specified, recognition results will be collected using gpu communication. Otherwise, it will save the results on different gpus to TMPDIR and collect them by the rank 0 worker.

• **--tmpdir**: Temporary directory used for collecting results from multiple workers, available when --gpu_collect is not specified.

• **--launcher**: Items for distributed job initialization launcher. Allowed choices are none, pytorch, slurm, mpi. Especially, if set to none, it will test in a non-distributed mode.

• **--local_rank**: ID for local rank. If not specified, it will be set to 0.

Examples:

Assume that you have already downloaded the checkpoints to the directory checkpoints/, and test PWC-Net on Sintel clean and final sub-datasets without save predicted flow files and evaluate the EPE.

```
python tools/test.py configs/pwc_net_8x1_sfine_sintel_384x768.py \
    checkpoints/pwcnet_8x1_sfine_sintel_384x768.pth --eval EPE
```

We recommend using single gpu and setting batch_size=1 to evaluate models, as it must ensure that the number of dataset samples can be divisible by batch size, so even if working on slurm, we will use one gpu to test. Assume our partition is Test and job name is test_pwc, so here is the example:

```
GPUS=1 GPUS_PER_NODE=1 CPUS_PER_TASK=2 ./tools/slurm_test.sh Test test_pwc \
    configs/pwc_net_8x1_sfine_sintel_384x768.py \
    checkpoints/pwcnet_8x1_sfine_sintel_384x768.pth --eval EPE
```
Flow estimators pre-trained on the FlyingChairs and FlyingThings3d can serve as a good pre-trained model for other datasets. This tutorial provides instruction for users to use the models provided in the Model Zoo for other datasets to obtain better performance. MMFlow also provides out-of-the-box tools for training models. This section will show how to train predefined models on standard datasets.

### 8.1 Modify training schedule

The fine-tuning hyper-parameters vary from the default schedule. It usually requires smaller learning rate and less training iterations.

```python
# optimizer
optimizer = dict(type='Adam', lr=1e-5, weight_decay=0.0004, betas=(0.9, 0.999))
optimizer_config = dict(grad_clip=None)

# learning policy
lr_config = dict(
    policy='step',
    by_epoch=False,
    gamma=0.5,
    step=[
        45000, 65000, 85000, 95000, 97500, 100000, 110000, 120000, 130000,
        140000
    ]
)
runner = dict(type='IterBasedRunner', max_iters=150000)
checkpoint_config = dict(by_epoch=False, interval=10000)
evaluation = dict(interval=10000, metric='EPE')
```

### 8.2 Use pre-trained model

Users can load a pre-trained model by setting the load_from field of the config to the model’s path or link. The users might need to download the model weights before training to avoid the download time during training.

```python
# use the pre-trained model for the whole PWC-Net
load_from = 'https://download.openmmlab.com/mmflow/pwcnet/pwcnet_8x1_sfine_flyingthings3d_subset_384x768.pth'  # model path can be found in model zoo
```
8.3 Training on a single GPU

We provide tools/train.py to launch training jobs on a single GPU. The basic usage is as follows.

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

During training, log files and checkpoints will be saved to the working directory, which is specified by `work_dir` in the config file or via CLI argument `--work-dir`.

This tool accepts several optional arguments, including:

- `--work-dir ${WORK_DIR}`: Override the working directory.
- `--resume-from ${CHECKPOINT_FILE}`: Resume from a previous checkpoint file.
- `--cfg-option`: Override some settings in the used config, the key-value pair in xxx=yyy format will be merged into config file. For example, ‘--cfg-option model.encoder.in_channels=6’.

Note:
Difference between `resume-from` and `load-from`:

`resume-from` loads both the model weights and optimizer status, and the iteration is also inherited from the specified checkpoint. It is usually used for resuming the training process that is interrupted accidentally. `load-from` only loads the model weights and the training iteration starts from 0. It is usually used for finetuning.

8.3.1 Training on CPU

The process of training on the CPU is consistent with single GPU training. We just need to disable GPUs before the training process.

```bash
export CUDA_VISIBLE_DEVICES=-1
```

And then run the script above.

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

8.4 Training on multiple GPUs

MMFlow implements distributed training with `MMDistributedDataParallel`.

We provide tools/dist_train.sh to launch training on multiple GPUs. The basic usage is as follows.

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

Optional arguments remain the same as stated above and has additional arguments to specify the number of GPUs.
8.4.1 Launch multiple jobs on a single machine

If you would like to launch multiple jobs on a single machine, e.g., 2 jobs of 4-GPU training on a machine with 8 GPUs, you need to specify different ports (29500 by default) for each job to avoid communication conflict.

If you use `dist_train.sh` to launch training jobs, you can set the port in commands.

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

8.5 Training on multiple nodes

MMFlow relies on `torch.distributed` package for distributed training. Thus, as a basic usage, one can launch distributed training via PyTorch’s `launch` utility.

8.5.1 Train with multiple machines

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

On the first machine:

```
NNODES=2 NODE_RANK=0 PORT=${MASTER_PORT} MASTER_ADDR=${MASTER_ADDR} sh tools/dist_train.sh

sh ${CONFIG_FILE} ${GPUS}
```

On the second machine:

```
NNODES=2 NODE_RANK=1 PORT=${MASTER_PORT} MASTER_ADDR=${MASTER_ADDR} sh tools/dist_train.sh

sh ${CONFIG_FILE} ${GPUS}
```

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

8.5.2 Manage jobs with Slurm

Slurm is a good job scheduling system for computing clusters. On a cluster managed by Slurm, you can use `slurm_train.sh` to spawn training jobs. It supports both single-node and multi-node training.

The basic usage is as follows.

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

Below is an example of using 8 GPUs to train PWC-Net on a Slurm partition named `dev`, and set the work-dir to some shared file systems.

```
GPUS=8 sh tools/slurm_train.sh dev pwc_chairs configs/pwcnet/pwcnet_8x1_slong_flyingchairs_384x448.py work_dir/pwc_chairs
```

You can check the source code to review full arguments and environment variables.

When using Slurm, the port option need to be set in one of the following ways:

1. Set the port through `--cfg-options`. This is more recommended since it does not change the original configs.
2. Modify the config files to set different communication ports.

In config1.py, set

```python
dist_params = dict(backend='nccl', port=29500)
```

In config2.py, set

```python
dist_params = dict(backend='nccl', port=29501)
```

Then you can launch two jobs with config1.py and config2.py.

```bash
GPUS=4 GPUS_PER_NODE=4 sh tools/slurm_train.sh $(PARTITION) $(JOB_NAME) config1.py $(WORK_DIR) --cfg-options 'dist_params.port=29500'
GPUS=4 GPUS_PER_NODE=4 sh tools/slurm_train.sh $(PARTITION) $(JOB_NAME) config2.py $(WORK_DIR) --cfg-options 'dist_params.port=29501'
```


9.1 Design of Data pipelines

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. Since the data flow estimation may not be the same size, we introduce a new DataContainer type in MMCV to help collect and distribute data of different size. See here for more details.

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, formatting.

Here is a pipeline example for PWC-Net

```python
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations'),
    dict(type='ColorJitter', brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5),
    dict(type='RandomGamma', gamma_range=(0.7, 1.5)),
    dict(type='Normalize', mean=[0., 0., 0.], std=[255., 255., 255.], to_rgb=False),
    dict(type='GaussianNoise', sigma_range=(0, 0.04), clamp_range=(0., 1.)),
    dict(type='RandomFlip', prob=0.5, direction='horizontal'),
    dict(type='RandomFlip', prob=0.5, direction='vertical'),
    dict(type='RandomAffine',
        global_transform=dict(
            translates=(0.05, 0.05),
            zoom=(1.0, 1.5),
            shear=(0.86, 1.16),
            rotate=(-10., 10.)
        ),
        relative_transform=dict(
            translates=(0.00375, 0.00375),
            zoom=(0.985, 1.015),
            shear=(1.0, 1.0),
            rotate=(-1.0, 1.0)
        ),
    dict(type='RandomCrop', crop_size=(384, 448)),
    dict(type='DefaultFormatBundle'),
    dict(

    )
]
```

(continues on next page)
For each operation, we list the related dict fields that are added/updated/removed.

### 9.1.1 Data loading

LoadImageFromFile
- add: img1, img2, filename1, filename2, img_shape, ori_shape, pad_shape, scale_factor, img_norm_cfg

LoadAnnotations
- add: flow_gt, filename_flow

### 9.1.2 Pre-processing

ColorJitter
- update: img1, img2

RandomGamma
- update: img1, img2

Normalize
- update: img1, img2, img_norm_cfg

GaussianNoise
- update: img1, img2

RandomFlip
- update: img1, img2, flow_gt

RandomAffine
- update: img1, img2, flow_gt

RandomCrop
- update: img1, img2, flow_gt, img_shape
9.1.3 Formatting

DefaultFormatBundle

- update: img1, img2, flow_gt

Collect

- add: img_meta (the keys of img_meta is specified by meta_keys)
- remove: all other keys except for those specified by keys

9.2 Extend and use custom pipelines

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

```python
from mmflow.datasets import PIPELINES

@PIPELINES.register_module()
class MyTransform:
    def __call__(self, results):
        results['dummy'] = True
        return results
```

2. Import the new class.

```python
from .my_pipeline import MyTransform
```

3. Use it in config files.

```python
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations'),
    dict(type='ColorJitter', brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5),
    dict(type='RandomGamma', gamma_range=(0.7, 1.5)),
    dict(type='Normalize', mean=[0., 0., 0.], std=[255., 255., 255.], to_rgb=False),
    dict(type='GaussianNoise', sigma_range=(0, 0.04), clamp_range=(0., 1.)),
    dict(type='RandomFlip', prob=0.5, direction='horizontal'),
    dict(type='RandomFlip', prob=0.5, direction='vertical'),
    dict(type='RandomAffine',
        global_transform=dict(  
            translates=(0.05, 0.05),
            zoom=(1.0, 1.5),
            shear=(0.86, 1.16),
            rotate=(-10., 10.)
        ),
        relative_transform=dict(  
            translates=(0.00375, 0.00375),
            zoom=(0.985, 1.015),
            shear=(1.0, 1.0),
            rotate=(-1.0, 1.0)
        )
    ),
```

(continues on next page)
dict(type=
(RandomCrop', crop_size=(384, 448)),
dict(type='MyTransform'),
dict(type='DefaultFormatBundle'),
dict(
    type='Collect',
    keys=['imgs', 'flow_gt'],
    meta_keys=('img_fields', 'ann_fields', 'filename1', 'filename2',
               'ori_filename1', 'ori_filename2', 'filename_flow',
               'ori_filename_flow', 'ori_shape', 'img_shape',
               'img_norm_cfg'))]
MMFlow decomposes a flow estimation method `flow_estimator` into encoder and decoder. This tutorial is for how to add new components.

### 10.1 Add a new encoder

1. Create a new file `mmflow/models/encoders/my_model.py`.

   ```python
   from mmcv.runner import BaseModule
   from ..builder import ENCODERS
   @ENCODERS.register_module()
   class MyModel(BaseModule):
       def __init__(self, arg1, arg2):
           pass
       def forward(self, x):
           # should return a tuple
           pass
       def init_weights(self, pretrained=None):
           pass
   
   2. Import the module in `mmflow/models/encoders/__init__.py`.

      ```python
      from .my_model import MyModel
      ```

### 10.2 Add a new decoder

1. Create a new file `mmflow/models/decoders/my_decoder.py`.

    You can write a new head inherit from `BaseModule` from MMCV, and overwrite `forward(self, x)`, `forward_train` and `forward_test` methods. We have a unified interface for `weights initialization` in MMCV, you can use `init_cfg` to specify the initialization function and arguments, or overwrite `init_weights` if you prefer customized initialization.
from ..builder import DECODERS

@DECODERS.register_module()
class MyDecoder(BaseModule):
    def __init__(self, arg1, arg2):
        pass

    def forward(self, *args):
        pass

    # optional
    def init_weights(self):
        pass

    def forward_train(self, *args, flow_gt):
        flow_pred = self.forward(*args)
        return self.losses(flow_pred, flow_gt)

    def forward_test(self, *args, img_metas):
        flow_pred = self.forward(*args)
        return self.get_flow(flow_pred, img_metas)

losses is the loss function to compute the losses between the model output and target, get_flow is implemented in BaseDecoder to restore the flow shape as the original shape of input images.

1. Import the module in mmflow/models/decoders/__init__.py

from .my_decoder import MyDecoder

### 10.3 Add a new flow_estimator

1. Create a new file mmflow/models/flow_estimators/my_estimator.py

You can write a new flow estimator inherit from FlowEstimator like PWC-Net, and implement forward_train and forward_test

from ..builder import FLOW_ESTIMATORS
from .base import FlowEstimator

@FLOW_ESTIMATORS.register_module()
class MyEstimator(FlowEstimator):
    def __init__(self, arg1, arg2):
        pass

    def forward_train(self, *args, imgs):
        pass

    def forward_test(self, *args, img_metas):
        pass

(continues on next page)
def forward_test(self, imgs):
    pass

2. Import the module in mmflow/models/flow_estimator/__init__.py

from .my_estimator import MyEstimator

3. Use it in your config file.

we set the module type as MyEstimator.

model = dict(
    type='MyEstimator',
    encoder=dict(
        type='MyModel',
        arg1=xxx,
        arg2=xxx),
    decoder=dict(
        type='MyDecoder',
        arg1=xxx,
        arg2=xxx))

10.4 Add new loss

Assume you want to add a new loss as MyLoss, for flow estimation. To add a new loss function, the users need implement it in mmflow/models/losses/my_loss.py.

import torch
import torch.nn as nn

from mmflow.models import LOSSES

def my_loss(pred, target):
    pass

@LOSSES.register_module()
class MyLoss(nn.Module):
    def __init__(self, arg1):
        super(MyLoss, self).__init__()

        def forward(self, output, target):
            return my_loss(output, target)

Then the users need to add it in the mmflow/models/losses/__init__.py.

from .my_loss import MyLoss, my_loss

To use it, modify the flow_loss field in the model.
flow_loss=dict(type='MyLoss', use_target_weight=False)
In this tutorial, we will introduce some methods about how to customize optimization methods, training schedules, workflow and hooks when running your own settings for the project.

### 11.1 Customize Optimization Methods

#### 11.1.1 Customize optimizer supported by PyTorch

We already support to use all the optimizers implemented by PyTorch, and the only modification is to change the optimizer field of config files. For example, if you want to use Adam, the modification could be as the following.

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

To modify the learning rate of the model, the users only need to modify the lr in the config of optimizer. The users can directly set arguments following the API doc of PyTorch.

For example, if you want to use Adam with the setting like `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)` in PyTorch, the modification could be set as the following.

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

#### 11.1.2 Customize self-implemented optimizer

1. Define a new optimizer

A customized optimizer could be defined as following.

Assume you want to add an optimizer named MyOptimizer, which has arguments a, b, and c. You need to create a new directory named mmflow/core/optimizer. And then implement the new optimizer in a file, e.g., in mmflow/core/optimizer/my_optimizer.py:

```python
from ..builder import OPTIMIZERS
from torch.optim import Optimizer

@OPTIMIZERS.register_module()
class MyOptimizer(Optimizer):
```

(continues on next page)
2. Add the optimizer to registry

To find the above module defined above, this module should be imported into the main namespace at first. There are two ways to achieve it.

- Modify `mmflow/core/optimizer/__init__.py` to import it.
  The newly defined module should be imported in `mmflow/core/optimizer/__init__.py` so that the registry will find the new module and add it:

```python
from .my_optimizer import MyOptimizer
```

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

`custom_imports` can import module manually as long as the module can be located in `PYTHONPATH`, without modifying source code

```python
custom_imports = dict(imports=['mmflow.core.optimizer.my_optimizer'], allow_failed_imports=False)
```

The module `mmflow.core.optimizer.my_optimizer` will be imported at the beginning of the program and the class `MyOptimizer` is then automatically registered. Note that only the package containing the class `MyOptimizer` should be imported. `mmflow.core.optimizer.my_optimizer.MyOptimizer` cannot be imported directly.

3. Specify the optimizer in the config file

Then you can use `MyOptimizer` in `optimizer` field of config files. In the configs, the optimizers are defined by the field `optimizer` like the following:

```python
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
```

To use your own optimizer, the field can be changed to

```python
optimizer = dict(type='MyOptimizer', a=a_value, b=b_value, c=c_value)
```

11.1.3 Customize optimizer constructor

Some models may have some parameter-specific settings for optimization, e.g. weight decay for BatchNorm layers. The users can do those fine-grained parameter tuning through customizing optimizer constructor.

```python
from mmcv.utils import build_from_cfg
from mmcv.runner.optimizer import OPTIMIZER_BUILDERS, OPTIMIZERS
from mmflow.utils import get_root_logger
from .my_optimizer import MyOptimizer
```
@OPTIMIZER_BUILDERS.register_module()
class MyOptimizerConstructor:
    
def __init__(self, optimizer_cfg, paramwise_cfg=None):
        pass
    
def __call__(self, model):
        return my_optimizer

The default optimizer constructor is implemented here, which could also serve as a template for the new optimizer constructor.

11.1.4 Additional settings

Tricks not implemented by the optimizer should be implemented through optimizer constructor (e.g., set parameter-wise learning rates) or hooks. We list some common settings that could stabilize the training or accelerate the training. Feel free to create PR, issue for more settings.

- **Use gradient clip to stabilize training**: Some models need gradient clip to clip the gradients to stabilize the training process. An example is as below:

```python
optimizer_config = dict(grad_clip=dict(max_norm=35, norm_type=2))
```

- **Use momentum schedule to accelerate model convergence**: We support momentum scheduler to modify model’s momentum according to learning rate, which could make the model converge in a faster way. Momentum scheduler is usually used with LR scheduler, for example, the following config is used in 3D detection to accelerate convergence. For more details, please refer to the implementation of CyclicLrUpdater and CyclicMomentumUpdater.

```python
lr_config = dict(
    policy='cyclic',
    target_ratio=(10, 1e-4),
    cyclic_times=1,
    step_ratio_up=0.4,
)
momentum_config = dict(
    policy='cyclic',
    target_ratio=(0.85 / 0.95, 1),
    cyclic_times=1,
    step_ratio_up=0.4,
)```
11.2 Customize Training Schedules

we use step learning rate with default value in config files, this calls StepLRHook in MMCV. We support many other learning rate schedule here, such as CosineAnnealing and Poly schedule. Here are some examples

- Poly schedule:

```python
lr_config = dict(policy='poly', power=0.9, min_lr=1e-4, by_epoch=False)
```

- CosineAnnealing schedule:

```python
lr_config = dict(
    policy='CosineAnnealing',
    warmup='linear',
    warmup_iters=1000,
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

11.3 Customize Workflow

Workflow is a list of (phase, epochs) to specify the running order and epochs. By default it is set to be

```python
workflow = [('train', 1)]
```

which means running 1 epoch for training. Sometimes user may want to check some metrics (e.g. loss, accuracy) about the model on the validate set. In such case, we can set the workflow as

```python
[('train', 1), ('val', 1)]
```

so that 1 epoch for training and 1 epoch for validation will be run iteratively.

Note:

1. The parameters of model will not be updated during val epoch.
2. Keyword total_epochs in the config only controls the number of training epochs and will not affect the validation workflow.
3. Workflows [('train', 1), ('val', 1)] and [('train', 1)] will not change the behavior of EpochEvalHook because EpochEvalHook is called by after_train_epoch and validation workflow only affect hooks that are called through after_val_epoch. Therefore, the only difference between [('train', 1), ('val', 1)] and [('train', 1)] is that the runner will calculate losses on the validation set after each training epoch.
11.4 Customize Hooks

11.4.1 Customize self-implemented hooks

1. Implement a new hook

Here we give an example of creating a new hook in mmflow and using it in training.

```python
from mmcv.runner import HOOKS, Hook

@HOOKS.register_module()
class MyHook(Hook):
    def __init__(self, a, b):
        pass

    def before_run(self, runner):
        pass

    def after_run(self, runner):
        pass

    def before_epoch(self, runner):
        pass

    def after_epoch(self, runner):
        pass

    def before_iter(self, runner):
        pass

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

Depending on the functionality of the hook, the users need to specify what the hook will do at each stage of the training in `before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`.

2. Register the new hook

Then we need to make `MyHook` imported. Assuming the file is in `mmflow/core/hooks/my_hook.py` there are two ways to do that:

- Modify `mmflow/core/hooks/__init__.py` to import it.

  The newly defined module should be imported in `mmflow/core/hooks/__init__.py` so that the registry will find the new module and add it:

  ```python
  from .my_hook import MyHook
  ```

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

  ```python
  custom_imports = dict(imports=['mmflow.core.hooks.my_hook'], allow_failed_imports=False)
  ```
3. Modify the config

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

You can also set the priority of the hook by adding key `priority` to 'NORMAL' or 'HIGHEST' as below

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

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

11.4.2 Use hooks implemented in MMCV

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

```python
mmcv_hooks = [
    dict(type='MMCVHook', a=a_value, b=b_value, priority='NORMAL')
]
```

11.4.3 Modify default runtime hooks

There are some common hooks that are not registered through `custom_hooks` but has been registered by default when importing MMCV, they are

- `log_config`
- `checkpoint_config`
- `evaluation`
- `lr_config`
- `optimizer_config`
- `momentum_config`

In those hooks, only the logger hook has the `VERY_LOW` priority, others’ priority are `NORMAL`. The above-mentioned tutorials already cover how to modify `optimizer_config`, `momentum_config`, and `lr_config`. Here we reveals how what we can do with `log_config`, `checkpoint_config`, and `evaluation`.

**Checkpoint config**

The MMCV runner will use `checkpoint_config` to initialize `CheckpointHook`.

```python
checkpoint_config = dict(interval=1)
```

The users could set `max_keep_ckpts` to only save only small number of checkpoints or decide whether to store state dict of optimizer by `save_optimizer`. More details of the arguments are [here](#).
Log config

The log_config wraps multiple logger hooks and enables to set intervals. Now MMCV supports WandbLoggerHook, MlflowLoggerHook, and TensorboardLoggerHook. The detail usages can be found in the doc.

```python
log_config = dict(
    interval=50,
    hooks=[
        dict(type='TextLoggerHook'),
        dict(type='TensorboardLoggerHook')
    ]
)
```

Evaluation config

The config of evaluation will be used to initialize the EvalHook. Except for the key interval, other arguments such as metric will be passed to the online_evaluation()

```python
evaluation = dict(interval=50000, metric='EPE')
```
Please check the following conventions if you would like to modify MMFlow as your own project.

### 12.1 Optical flow visualization

In MMFlow, we render the optical flow following this color wheel from Middlebury flow dataset. Smaller vectors are lighter and color represents the direction.

### 12.2 Return Values

In MMFlow, a dict containing losses will be returned by `model(**data, test_mode=False)`, and a list containing a batch of inference results will be returned by `model(**data, test_mode=True)`. As some methods will predict flow with different direction or occlusion mask, the item type of inference results is `Dict[str=ndarray]`.

For example in PWCNetDecoder,

```python
@DECODERS.register_module()
class PWCNetDecoder(BaseDecoder):
    def forward_test(self, feat1: Dict[str, Tensor], feat2: Dict[str, Tensor], H: int, W: int, img_metas: Optional[Sequence[dict]] = None) -> Sequence[Dict[str, ndarray]]:
        """Forward function when model testing."

        Args:
            feat1 (Dict[str, Tensor]): The feature pyramid from the first image.
            feat2 (Dict[str, Tensor]): The feature pyramid from the second image.
            H (int): The height of images after data augmentation.
            W (int): The width of images after data augmentation.
            img_metas (Sequence[dict], optional): meta data of image to revert the flow to original ground truth size. Defaults to None.

        Returns:
```

(continues on next page)
Sequence[Dict[str, ndarray]]: The batch of predicted optical flow with the same size of images before augmentation.

```python
flow_pred = self.forward(feat1, feat2)
flow_result = flow_pred[self.end_level]
# resize flow to the size of images after augmentation.
flow_result = F.interpolate(
    flow_result, size=(H, W), mode='bilinear', align_corners=False)
# reshape [2, H, W] to [H, W, 2]
flow_result = flow_result.permute(0, 2, 3, 1).cpu().data.numpy() * self.flow_div

# unravel batch dim,
flow_result = list(flow_result)
flow_result = [dict(flow=f) for f in flow_result]
return self.get_flow(flow_result, img_metas=img_metas)
```

def forward_train(self,
    feat1: Dict[str, Tensor],
    feat2: Dict[str, Tensor],
    flow_gt: Tensor,
    valid: Optional[Tensor] = None) -> Dict[str, Tensor]:
    """Forward function when model training.

    Args:
    feat1 (Dict[str, Tensor]): The feature pyramid from the first image.
    feat2 (Dict[str, Tensor]): The feature pyramid from the second image.
    flow_gt (Tensor): The ground truth of optical flow from image1 to image2.
    valid (Tensor, optional): The valid mask of optical flow ground truth. Defaults to None.

    Returns:
    Dict[str, Tensor]: The dict of losses.
    """
    flow_pred = self.forward(feat1, feat2)
    return self.losses(flow_pred, flow_gt, valid=valid)

def losses(self,
    flow_pred: Dict[str, Tensor],
    flow_gt: Tensor,
    valid: Optional[Tensor] = None) -> Dict[str, Tensor]:
    """Compute optical flow loss.

    Args:
    flow_pred (Dict[str, Tensor]): multi-level predicted optical flow.
    """
flow_gt (Tensor): The ground truth of optical flow.
valid (Tensor, optional): The valid mask. Defaults to None.

Returns:
Dict[str, Tensor]: The dict of losses.

loss = dict()
loss['loss_flow'] = self.flow_loss(flow_pred, flow_gt, valid)
return loss
13.1 v0.5.1 (07/29/2022)

13.1.1 Improvements

• Set the maximum version of MMCV to 1.7.0 (167)
• Update the qq_group_qrcode image in resources (166)

13.1.2 New Contributors

• @Weepingchestnut made their first contribution in https://github.com/open-mmlab/mmflow/pull/166

13.2 v0.5.0 (07/01/2022)

13.2.1 Highlight

• Add config and pre-trained model for FlowNet2 on FlyingChairs (163)

13.2.2 Documentation

• Add a template for PR (160)
• Fix config file error in metafile (151)
• Fix broken URL in metafile (157)
• Fix broken URLs for issue reporting in README (147)
13.2.3 Improvements

- Add mim to extras_require in setup.py (154)
- Fix mdformat version to support python3.6 and remove ruby install (153)
- Add test_mim.yml for testing commands of mim in CI (158)

13.3 v0.4.2 (05/31/2022)

13.3.1 Bug Fixes

- Inference bug for sparse flow map (133)
- H and W input images must be divisible by 2**6 (136)

13.3.2 Documentation

- Configure Myst-parser to parse anchor tag (129)
- Replace markdownlint with mdformat for avoiding installing ruby (130)
- Rewrite install and README by (139, 140, 141, 144, 145)

13.4 v0.4.1 (04/29/2022)

13.4.1 Feature

- Loading flow annotation from file client (#116)
- Support overall dastaloader settings (#117)
- Generate ann_file for flyingchairs (121)

13.4.2 Improvements

- Add GPG keys in CI (127)

13.4.3 Bug Fixes

- The config and weights are not corresponding in the metafile.yml (#118)
- Replace recommonmark with myst_parser (#120)
13.4.4 Documentation

- Add zh-cn doc 0_config_.md (#126)

13.5 New Contributors

- @HiitXinyiii made their first contribution in https://github.com/open-mmlab/mmflow/pull/118
- @SheffieldCao made their first contribution in https://github.com/open-mmlab/mmflow/pull/126

13.6 v0.4.0(04/01/2022)

13.6.1 Highlights

- Support occlusion estimation methods including flow forward-backward consistency, range map of the backward flow, and flow forward-backward abstract difference

13.6.2 Features

- Support three occlusion estimation methods (#106)
- Support different seeds on different ranks when distributed training (#104)

13.6.3 Improvements

- Revise collect_env for win platform (#112)
- Add script and documentation for multi-machine distributed training (#107)

13.7 v0.3.0(03/04/2022)

13.7.1 Highlights

- Officially support CPU Train/Inference
- Add census loss, SSIM loss and smoothness loss
- Officially support model inference in windows platform
- Update nan files in Flyingthings3d_subset dataset
13.7.2 Features

• Add census loss (#100)
• Add smoothness loss function (#97)
• Add SSIM loss function (#96)

13.7.3 Bug Fixes

• Update nan files in Flyingthings3d_subset (#94)
• Add pretrained pwcnet-model when train PWCNet+ (#99)
• Fix bug in non-distributed multi-gpu training/testing (#85)
• Fix writing flow map bug in test (#83)

13.7.4 Improvements

• Add win-ci (#92)
• Update the installation of MMCV (#89)
• Upgrade isort in pre-commit hook (#87)
• Support CPU Train/Inference (#86)
• Add multi-processes script (#79)
• Deprecate the support for “python setup.py test” (#73)

13.7.5 Documentation

• Fix broken URLs in GMA README (#93)
• Fix date format in readme (#90)
• Reorganizing OpenMMLab projects in readme (#98)
• Fix README files of algorithms (#84)
• Add url of OpenMMLab and platform in README (76)

13.8 v0.2.0(01/07/2022)

13.8.1 Highlights

• Support GMA: Learning to Estimate Hidden Motions with Global Motion Aggregation (ICCV 2021) (#32)
• Fix the bug of wrong refine iter in RAFT, and update RAFT model checkpoint after the bug fixing (#62, #68)
• Support resuming from the latest checkpoint automatically (#71)
13.8.2 Features

- Add `scale_as_level` for multi-level flow loss (#58)
- Add `scale_mode` for correlation block (#56)
- Add `upsample_cfg` in IRR-PWC decoder (#53)

13.8.3 Bug Fixes

- Resized input image must be dividable by $2^6$ (#65)
- Fix RAFT wrong refine iter after evaluation (#62)

13.8.4 Improvements

- Add `persistent_workers=True` in `val_dataloader` (#63)
- Revise `env_info` key (#46)
- Add digital version (#43)
- Try to create a symbolic link on windows (#37)
- Set a random seed when the user does not set a seed (#27)

13.8.5 Refactors

- Refactor utils in models (#50)

13.8.6 Documentation

- Refactor documentation (#14)
- Fix script bug in `FlyingChairs` dataset prepare (#21)
- Fix broken links in model_zoo (#60)
- Update metafile (#39, #41, #49)
- Update documentation (#28, #35, #36, #47, #48, #70)
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.

14.1 Installation

The compatible MMFlow and MMCV versions are as below. Please install the correct version of MMCV to avoid installation issues.

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`.
CHAPTER
FIFTEEN

MMFLOW.APIS
### 16.1 Evaluation

```python
class mmflow.core.evaluation.DistEvalHook:
        # Distributed evaluation hook.

        Parameters
        - **dataloader** (DataLoader) – A PyTorch dataloader.
        - **interval** (int) – Evaluation interval (by epochs). Default: 1.
        - **tmpdir** (str | None) – Temporary directory to save the results of all processes. Default: None.
        - **gpu_collect** (bool) – Whether to use gpu or cpu to collect results. Default: False.
        - **by_epoch** (bool) – Determine perform evaluation by epoch or by iteration. If set to True, it will perform by epoch. Otherwise, by iteration. Default: False.
        - **dataset_name** (str, list, optional) – The name of the dataset this evaluation hook will doing in.
        - **eval_kwargs** (any) – Evaluation arguments fed into the evaluate function of the dataset.

    def evaluate(self, runner: mmcv.runner.iter_based_runner.IterBasedRunner):
        # Evaluation function to call online evaluate function.

class mmflow.core.evaluation.EvalHook:
    def __init__(self, dataloader: torch.utils.data.dataloader.DataLoader, interval: int = 1, by_epoch: bool = False, dataset_name: Optional[Union[str, Sequence[str]]] = None, **eval_kwargs: Any):
        # Evaluation hook.

        Parameters
        - **dataloader** (DataLoader) – A PyTorch dataloader.
        - **interval** (int) – Evaluation interval (by epochs). Default: 1.
        - **by_epoch** (bool) – Determine perform evaluation by epoch or by iteration. If set to True, it will perform by epoch. Otherwise, by iteration. Default: False.
        - **dataset_name** (str, list, optional) – The name of the dataset this evaluation hook will doing in.
```

Distributed evaluation hook.

Evaluation hook.
• **eval_kwargs** *(any)* – Evaluation arguments fed into the evaluate function of the dataset.

    **after_train_epoch**(runner: mmcv.runner.iter_based_runner.IterBasedRunner) → None
    After train epoch.

    **after_train_iter**(runner: mmcv.runner.iter_based_runner.IterBasedRunner) → None
    After train iteration.

    **evaluate**(runner: mmcv.runner.iter_based_runner.IterBasedRunner) → None
    Evaluation function to call online evaluate function.

    **mmflow.core.evaluation.end_point_error**(flow_pred: Sequence[numpy.ndarray], flow_gt: Sequence[numpy.ndarray], valid_gt: Sequence[numpy.ndarray]) → float
    Calculate end point errors between prediction and ground truth.

    **Parameters**
    - **flow_pred** *(list)* – output list of flow map from flow_estimator shape(H, W, 2).
    - **flow_gt** *(list)* – ground truth list of flow map shape(H, W, 2).
    - **valid_gt** *(list)* – the list of valid mask for ground truth with the shape (H, W).

    **Returns** end point error for output.

    **Return type** float

    **mmflow.core.evaluation.end_point_error_map**(flow_pred: numpy.ndarray, flow_gt: numpy.ndarray) → numpy.ndarray
    Calculate end point error map.

    **Parameters**
    - **flow_pred** *(ndarray)* – The predicted optical flow with the shape (H, W, 2).
    - **flow_gt** *(ndarray)* – The ground truth of optical flow with the shape (H, W, 2).

    **Returns** End point error map with the shape (H, W).

    **Return type** ndarray

    **mmflow.core.evaluation.eval_metrics**(results: Sequence[numpy.ndarray], flow_gt: Sequence[numpy.ndarray], valid_gt: Sequence[numpy.ndarray], metrics: Union[Sequence[str], str] = ['EPE']) → Dict[str, float]
    Calculate evaluation metrics.

    **Parameters**
    - **results** *(list)* – list of predicted flow maps.
    - **flow_gt** *(list)* – list of ground truth flow maps
    - **metrics** *(list, str)* – metrics to be evaluated. Defaults to [‘EPE’], end-point error.

    **Returns** metrics and their values.

    **Return type** dict
Evaluate model with multiple gpus online.

This function will not save the flow. Namely, there do not exist any IO operations in this function. Thus, in general, online mode will achieve a faster evaluation. However, using this function, the img_metadata must include the ground truth e.g. flow_gt or flow_fw_gt and flow_bw_gt.

**Parameters**

- **model** (*nn.Module*) – The optical flow estimator model.
- **data_loader** (*DataLoader*) – The test dataloader.
- **metric** (*str, list*) – Metrics to be evaluated. Default: ‘EPE’.
- **tmpdir** (*str*) – Path of directory to save the temporary results from different gpus under cpu mode.
- **gpu_collect** (*bool*) – Option to use either gpu or cpu to collect results.

**Returns** The evaluation result.

**Return type** *dict*

Evaluate model online.

**Parameters**

- **model** (*nn.Module*) – The optical flow estimator model.
- **data_loader** (*DataLoader*) – The test dataloader.
- **metric** (*str, list*) – Metrics to be evaluated. Default: ‘EPE’.
- **kwargs** (*any*) – Evaluation arguments fed into the evaluate function of the dataset.

**Returns** The evaluation result.

**Return type** *dict*

Calculate percentage of optical flow outliers for KITTI dataset.

**Parameters**

- **flow_pred** (*list*) – output list of flow map from flow_estimator shape(H, W, 2).
- **flow_gt** (*list*) – ground truth list of flow map shape(H, W, 2).
- **valid_gt** (*list*) – the list of valid mask for ground truth with the shape (H, W).

**Returns** optical flow outliers for output.

**Return type** *float*

Evaluate model with single gpu online.

This function will not save the flow. Namely, there do not exist any IO operations in this function. Thus, in general, online mode will achieve a faster evaluation. However, using this function, the img_metas must include the ground truth e.g. flow_gt or flow_fw_gt and flow_bw_gt.

Parameters

- **model** (nn.Module) – The optical flow estimator model.
- **data_loader** (DataLoader) – The test dataloader.
- **metric** (str, list) – Metrics to be evaluated. Default: ‘EPE’.

Returns

The evaluation result.

Return type
dict

16.2 hooks

class mmflow.core.hooks.LiteFlowNetStageLoadHook(src_level: str, dst_level: str)
Stage loading hook for LiteFlowNet.

This hook works for loading weights at the previous stage to the additional stage in this training.

Parameters

- **src_level** (str) – The source level to be loaded.
- **dst_level** (str) – The level that will load the weights.

before_run(runner: mmcv.runner.iter_based_runner.IterBasedRunner) → None
Before running function of Hook.

Parameters

- **runner** (IterBasedRunner) – The runner for this training. This hook only has be tested in IterBasedRunner.

class mmflow.core.hooks.MultiStageLrUpdaterHook(milestone_lrs: Sequence[float], milestone_iters: Sequence[int], steps: Sequence[Sequence[int]], gammas: Sequence[float], **kwargs: Any)
Multi-Stage Learning Rate Hook.

Parameters

- **milestone_lrs** (Sequence[float]) – The base LR for multi-stages.
- **milestone_iters** (Sequence[int]) – The first iterations in different stages.
- **steps** (Sequence[Sequence[int]]) – The steps to decay the LR in stages.
- **gammas** (Sequence[float]) – The list of decay LR ratios.
- **kwargs** (any) – The arguments of LrUpdaterHook.

get_lr(runner: mmcv.runner.iter_based_runner.IterBasedRunner, base_lr: float) → float
Get current LR.

Parameters
- `runner` (*IterBasedRunner*) – The runner to control the training workflow.
- `base_lr` (*float*) – The base LR in training workflow.

Returns: The current LR.

Return type: float
17.1 datasets

class mmflow.datasets.ChairsSDHom(*args: Any, **kwargs: Any)
    ChairsSDHom dataset.
    load_data_info() → None
        load data information.

class mmflow.datasets.Collect(keys: collections.abc.Sequence, meta_keys: collections.abc.Sequence =
    ('filename1', 'filename2', 'ori_filename1', 'ori_filename2', 'filename_flow',
    'ori_filename_flow', 'ori_shape', 'img_shape', 'pad_shape', 'scale_factor',
    'flip', 'flip_direction', 'img_norm_cfg'))
    Collect data from the loader relevant to the specific task.
    This is usually the last stage of the data loader pipeline. Typically keys is set to some subset of “img”, “flow_gt”.
    The “img_meta” item is always populated. The contents of the “img_meta” dictionary depends on “meta_keys”.
    By default this includes:
    • “img_shape”: shape of the image input to the network as a tuple (h, w, c). Note that images may be zero padded on the bottom/right if the batch tensor is larger than this shape.
    • “scale_factor”: a float indicating the preprocessing scale
    • “flip”: a boolean indicating if image flip transform was used
    • “filename1”: path to the image1 file
    • “filename2”: path to the image2 file
    • “ori_filename1”: image1 file name
    • “ori_filename2”: image2 file name
    • “ori_shape”: original shape of the image as a tuple (h, w, c)
    • “pad_shape”: image shape after padding
    • “img_norm_cfg”: a dict of normalization information:
        – mean - per channel mean subtraction
        – std - per channel std divisor
        – to_rgb - bool indicating if bgr was converted to rgb

Parameters
    • keys (Sequence[str]) – Keys of results to be collected in data.
• **meta_keys** *(Sequence[str], optional)* – Meta keys to be converted to mmcv. DataContainer and collected in data[img_metas]. Default: ('filename1', 'filename2', 'ori_filename1', 'ori_filename2', 'ori_shape', 'img_shape', 'pad_shape', 'scale_factor', 'flip', 'flip_direction', 'img_norm_cfg')

```python
class mmflow.datasets.ColorJitter(asymmetric_prob=0.0, brightness=0.0, contrast=0.0, saturation=0.0, hue=0.0)
```

Randomly change the brightness, contrast, saturation and hue of an image. :param asymmetric_prob: the probability to do color jitter for two images asymmetrically.

**Parameters**

- **brightness** *(float, tuple)* – How much to jitter brightness. brightness_factor is chosen uniformly from \[\max(0, 1 - \text{brightness}), 1 + \text{brightness}\] or the given \([\text{min}, \text{max}]\). Should be non negative numbers.
- **contrast** *(float, tuple)* – How much to jitter contrast. contrast_factor is chosen uniformly from \([\max(0, 1 - \text{contrast}), 1 + \text{contrast}]\) or the given \([\text{min}, \text{max}]\). Should be non negative numbers.
- **saturation** *(float, tuple)* – How much to jitter saturation. saturation_factor is chosen uniformly from \([\max(0, 1 - \text{saturation}), 1 + \text{saturation}]\) or the given \([\text{min}, \text{max}]\). Should be non negative numbers.
- **hue** *(float, tuple)* – How much to jitter hue. hue_factor is chosen uniformly from \([-\text{hue}, \text{hue}]\) or the given \([\text{min}, \text{max}]\). Should have \(0 \leq \text{hue} \leq 0.5\) or \(-0.5 \leq \text{min} \leq \text{max} \leq 0.5\).

```python
class mmflow.datasets.Compose(transforms: Sequence)
```

Compose multiple transforms sequentially.

**Parameters** transforms *(Sequence[dict | callable])* – Sequence of transform object or config dict to be composed.

```python
class mmflow.datasets.ConcatDataset(datasets: Sequence[torch.utils.data.dataset.Dataset], separate_eval: bool = True)
```

A wrapper of concatenated dataset.

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

**Parameters**

- **datasets** *(list[Dataset])* – A list of datasets.
- **separate_eval** *(bool)* – Whether to evaluate the results separately if it is used as validation dataset. Defaults to True.

```python
def evaluate(results: list, logger: Optional[Union[str, logging.Logger]] = None, **kwargs: Any)
```

Evaluate the results.

**Parameters**

- **results** *(list[dict] | tuple)* – Testing results of the dataset.
- **logger** *(logging.Logger | str | None)* – Logger used for printing related information during evaluation. Default: None.

**Returns** float: AP results of the total dataset or each separate dataset if `self.separate_eval=True`.

**Return type** dict[str]
class mmflow.datasets.DefaultFormatBundle
    Default formatting bundle.

    It simplifies the pipeline of formatting common fields, including “img” and “flow_gt”. These fields are formatted as follows.

    • img1: (1)transpose, (2)to tensor, (3)to DataContainer (stack=True)
    • img2: (1)transpose, (2)to tensor, (3)to DataContainer (stack=True)
    • flow_gt: (1)transpose, (2)to tensor, (3)to DataContainer (stack=True)

class mmflow.datasets.DistributedSampler(dataset: torch.utils.data.dataset.Dataset, num_replicas: Optional[int] = None, rank: Optional[int] = None, shuffle: bool = True, seed=0)
    
    DistributedSampler inheriting from torch.utils.data.DistributedSampler.

    This distributed sampler is compatible Pytorch==1.5, as there is no seed argument in Pytorch==1.5.

    Parameters

    • datasets (Dataset) – the dataset will be loaded.
    • num_replicas (int, optional) – Number of processes participating in distributed training. By default, world_size is retrieved from the current distributed group.
    • rank (int, optional) – Rank of the current process within num_replicas. By default, rank is retrieved from the current distributed group.
    • shuffle (bool) – If True (default), sampler will shuffle the indices.
    • seed (int) – random seed used to shuffle the sampler if shuffle=True. This number should be identical across all processes in the distributed group. Default: 0.

class mmflow.datasets.Erase(prob: float, bounds: Sequence = [50, 100], max_num: int = 3)
    
    Erase transform from RAFT is randomly erasing rectangular regions in img2 to simulate occlusions.

    Parameters

    • prob (float) – the probability for erase transform.
    • bounds (list, tuple) – the bounds for erase regions (bound_x, bound_y).
    • max_num (int) – the max number of erase regions.

    Returns revised results, ‘img2’ and ‘erase_num’ are added into results.

    Return type dict

class mmflow.datasets.FlyingChairs(*args, split_file: str, **kwargs)
    
    FlyingChairs dataset.

    Parameters split_file (str) – File name of train-validation split file for FlyingChairs.

    load_ann_info(filename: Sequence[str], filename_key: str) → None
    
    Load information of optical flow.

    This function splits the dataset into two subsets, training subset and testing subset.

    Parameters

    • filename (list) – ordered list of abstract file path of annotation.
    • filename_key (str) – the annotation e.g. ‘flow’.

    load_data_info() → None
    
    Load data information, including file path of image1, image2 and optical flow.
load_img_info(img1_filename: Sequence[str], img2_filename: Sequence[str]) → None
Load information of image1 and image2.

Parameters

• img1_filename (list) – ordered list of abstract file path of img1.
• img2_filename (list) – ordered list of abstract file path of img2.

class mmflow.datasets.FlyingChairsOcc(*args, **kwargs)
FlyingChairsOcc dataset.

load_ann_info(filename, filename_key)
Load information of optical flow.
This function splits the dataset into two subsets, training subset and testing subset.

Parameters

• filename (list) – ordered list of abstract file path of annotation.
• filename_key (str) – the annotation key for FlyingChairsOcc dataset ‘flow_fw’, ‘flow_bw’, ‘occ_fw’, and ‘occ_bw’.

load_data_info()
Load data information, including file path of image1, image2 and optical flow.

load_img_info(img1_filename, img2_filename)
Load information of image1 and image2.

Parameters

• img1_filename (list) – ordered list of abstract file path of img1.
• img2_filename (list) – ordered list of abstract file path of img2.

FlyingThings3D subset dataset.

Parameters

• direction (str) – Direction of flow, has 4 options ‘forward’, ‘backward’, ‘bidirection’ and ['forward', 'backward']. Default: ['forward', 'backward'].
• scene (list, str) – Scene in FlyingThings3D dataset, default: ‘left’. This default value is for RAFT, as FlyingThings3D is so large and not often used, and only RAFT use the ‘left’ data in it.
• pass_style (str) – Pass style for FlyingThings3D dataset, and it has 2 options ['clean', 'final']. Default: ‘clean’.

load_data_info() → None
Load data information, including file path of image1, image2 and optical flow.

class mmflow.datasets.FlyingThings3DSubset(*args, direction: Union[str, Sequence[str]] = ['forward', 'backward'], scene: Optional[Union[str, Sequence[str]]] = None, **kwargs)
FlyingThings3D subset dataset.

Parameters

• direction (str) – Direction of flow, has 4 options ‘forward’, ‘backward’, ‘bidirection’, and ['forward', 'backward']. Default: ['forward', 'backward'].
• **scene** *(list, str, optional)* – Scene in Flyingthings3D dataset, if scene is None, it means collecting data in all of scene of Flyingthings3D dataset. Default: None.

```python
load_data_info() → None
```
Load data information, including file path of image1, image2 and optical flow.

```python
class mmflow.datasets.GaussianNoise(sigma_range=(0.04), clamp_range=(-inf, inf))
```
Add Gaussian Noise to images.

Add Gaussian Noise, with mean 0 and std sigma uniformly sampled from sigma_range, to images. And then clamp the images to clamp_range.

**Parameters**

- **sigma_range** *(list(float) / tuple(float))* – Uniformly sample sigma of gaussian noise in sigma_range. Default: (0.04)

- **clamp_range** *(list(float) / tuple(float))* – The min and max value to clamp the images after adding gaussian noise. Default: (float('-inf'), float('inf')).

```python
class mmflow.datasets.HD1K(*args, **kwargs)
```
HD1K dataset.

```python
load_data_info() → None
```
Load data information, including file path of image1, image2 and optical flow.

```python
class mmflow.datasets.ImageToTensor(keys: collections.abc.Sequence)
```
Convert image to `torch.Tensor` by given keys.

The dimension order of input image is (H, W, C). The pipeline will convert it to (C, H, W). If only 2 dimension (H, W) is given, the output would be (1, H, W).

**Parameters**

- **keys** *(Sequence[str])* – Key of images to be converted to Tensor.

```python
class mmflow.datasets.InputPad(exponent, mode=’edge’, position=’center’, **kwargs)
```
Pad images such that dimensions are divisible by $2^n$ used in test.

**Parameters**

- **exponent** *(int)* – the exponent n of $2^n$

- **mode** *(str)* – mode for numpy.pad(). Defaults to ‘edge’.

- **position** *(str)* – ‘center’, ‘left’, ‘right’, ‘top’ and ‘down’. Defaults to ‘center’

```python
class mmflow.datasets.InputResize(exponent)
```
Resize images such that dimensions are divisible by $2^n$ :param exponent: the exponent n of $2^n$ :type exponent: int

**Returns**

- Resized results, ‘img_shape’, ‘scale_factor’ keys are added into result dict.

**Return type**

dict

```python
class mmflow.datasets.KITTI2012(*args, **kwargs)
```
KITTI flow 2012 dataset.

```python
load_data_info() → None
```
Load data information, including file path of image1, image2 and optical flow.

```python
class mmflow.datasets.KITTI2015(*args, **kwargs)
```
KITTI flow 2015 dataset.

```python
load_data_info() → None
```
Load data information, including file path of image1, image2 and optical flow.
class mmflow.datasets.LoadImageFromFile:
  
  to_float32: bool = False, color_type: str = 'color',
  file_client_args: dict = {'backend': 'disk'}, imdecode_backend: str = 'cv2')

Load image1 and image2 from file.

Required keys are “img1_info” (dict that must contain the key “filename” and “filename2”). Added or updated keys are “img1”, “img2”, “img_shape”, “ori_shape” (same as img_shape), “pad_shape” (same as img_shape), “scale_factor” (1.0, 1.0) and “img_norm_cfg” (means=0 and stds=1).

Parameters
  - 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.
  - color_type (str) – The flag argument for mmcv.imfrombytes(). Defaults to ‘color’.
  - file_client_args (dict) – Arguments to instantiate a FileClient. See mmcv.fileio.FileClient for details. Defaults to dict(backend='disk').
  - imdecode_backend (str) – Backend for mmcv.imdecode(). Default: ‘cv2’

class mmflow.datasets.MixedBatchDistributedSampler:

  datasets: list[torch.utils.data.dataset.Dataset],
  sample_ratio: list[float], num_replicas: Optional[int] = None, rank: Optional[int] = None, shuffle: bool = True, seed: int = 0)

Distributed Sampler for mixed data batch.

Parameters
  - datasets (list) – List of datasets will be loaded.
  - sample_ratio (list) – List of the ratio of each dataset in a batch, e.g. datasets=[DatasetA, DatasetB], sample_ratio=[0.25, 0.75], sample_per_gpu=1, gpus=8, it means 2 gpus load DatasetA, and 6 gpus load DatasetB. The length of datasets must be equal to length of sample_ratio.
  - num_replicas (int, optional) – Number of processes participating in distributed training. By default, world_size is retrieved from the current distributed group.
  - rank (int, optional) – Rank of the current process within num_replicas. By default, rank is retrieved from the current distributed group.
  - shuffle (bool) – If True (default), sampler will shuffle the indices.
  - seed (int) – random seed used to shuffle the sampler if shuffle=True. This number should be identical across all processes in the distributed group. Default: 0.

set_epoch(epoch: int) → None

Sets the epoch for this sampler. When shuffle=True, this ensures all replicas use a different random ordering for each epoch. Otherwise, the next iteration of this sampler will yield the same ordering.

Parameters epoch (int) – Epoch number.

class mmflow.datasets.Normalize:

  mean, std, to_rgb=True)

Normalize the image.

Added key is “img_norm_cfg”: .param mean: Mean values of 3 channels. .type mean: sequence .param std: Std values of 3 channels. .type std: sequence .param to_rgb: Whether to convert the image from BGR to RGB, default is True.
class mmflow.datasets.PhotometricDistortion(brightness_delta=32, contrast_range=(0.5, 1.5), saturation_range=(0.5, 1.5), hue_delta=18)

Apply photometric distortion to image sequentially, every transformation is applied with a probability of 0.5.

The position of random contrast is in second or second to last. 1. random brightness 2. random contrast (mode 0) 3. convert color from BGR to HSV 4. random saturation 5. random hue 6. convert color from HSV to BGR 7. random contrast (mode 1) 8. randomly swap channels :param brightness_delta: delta of brightness. :type brightness_delta: int :param contrast_range: range of contrast. :type contrast_range: tuple :param saturation_range: range of saturation. :type saturation_range: tuple :param hue_delta: delta of hue. :type hue_delta: int

brightness(img)
Brightness distortion.

contrast(img)
Contrast distortion.

convert(img, alpha=1, beta=0)
Multiple with alpha and add beat with clip.

hue(img)
Hue distortion.

saturation(img)
Saturation distortion.


Random affine transformation of images, flow map and occlusion map (if available).

Keys of global_transform and relative_transform should be the subset of ('translates', 'zoom', 'shear', 'rotate'). And also, each key and its corresponding values has to satisfy the following rules:

- **translates**: the translation ratios along x axis and y axis. **Defaults** to(0., 0.).
- **zoom**: the min and max zoom ratios. **Defaults** to (1.0, 1.0).
- **shear**: the min and max shear ratios. **Defaults** to (1.0, 1.0).
- **rotate**: the min and max rotate degree. **Defaults** to (0., 0.).

Parameters

- **global_transform** (dict) – A dict which contains keys: transform, zoom, shear, rotate. global_transform will transform both img1 and img2.
- **relative_transform** (dict) – A dict which contains keys: transform, zoom, shear, rotate. relative_transform will only transform img2 after global_transform to both images.
- **preserve_valid** (bool) – Whether continue transforming until both images are valid. A valid affine transform is an affine transform which guarantees the transformed image covers the whole original picture frame. **Defaults** to True.
- **check_bound** (bool) – Whether to check out of bound for transformed occlusion maps. If True, all pixels in borders of img1 but not in borders of img2 will be marked occluded. **Defaults** to False.

class mmflow.datasets.RandomCrop(crop_size)
Random crop the image & flow.

Parameters **crop_size** (tuple) – Expected size after cropping, (h, w).
Crop from img
g
Randomly get a crop bounding box.
class mmflow.datasets.RandomFlip(prob, direction='horizontal')
Flip the image and flow map.

Parameters
• prob (float) – The flipping probability.
• direction (str) – The flipping direction. Options are ‘horizontal’ and ‘vertical’. Default: ‘horizontal’.
class mmflow.datasets.RandomRotation(prob, angle, auto_bound=False)
Random rotation of the image from -angle to angle (in degrees).

Parameters
• prob (float) – The rotation probability.
• angle (float) – max angle of the rotation in the range from -180 to 180.
• auto_bound (bool) – Whether to adjust the image size to cover the whole rotated image. Default: False
class mmflow.datasets.RandomTranslate(prob=0.0, x_offset=0.0, y_offset=0.0)
Random translation of the images and flow map.

Parameters
• prob (float) – the probability to do translation.
• x_offset (float | tuple) – translate ratio on x axis, randomly choice [-x_offset, x_offset] or the given [min, max]. Default: 0.
• y_offset (float | tuple) – translate ratio on y axis, randomly choice [-x_offset, x_offset] or the given [min, max]. Default: 0.
class mmflow.datasets.RepeatDataset(dataset, times)
A wrapper of repeated dataset.

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

Parameters
• dataset (Dataset) – The dataset to be repeated.
• times (int) – Repeat times.
class mmflow.datasets.Rerange(min_value=0, max_value=255)
Rerange the image pixel value.

Parameters
• min_value (float or int) – Minimum value of the reranged image. Default: 0.
• max_value (float or int) – Maximum value of the reranged image. Default: 255.
mmflow.datasets.

**Sintel**

*(args, pass_style: str = clean, scene: Optional[Union[str, Sequence[str]]] = None, **kwargs)*

Sintel optical flow dataset.

**Parameters**

- **pass_style** *(str)* - Pass style for Sintel dataset, and it has 2 options ['clean', 'final']. Default: 'clean'.
- **scene** *(str, list, optional)* - Scene in Sintel dataset, if scene is None, it means collecting data in all of scene of Sintel dataset. Default: None.

.. method:: load_data_info()

    Load data information, including file path of image1, image2 and optical flow.

.. method:: pre_pipeline(results: Sequence[dict])

    Prepare results dict for pipeline.

For Sintel, there is an additional annotation, invalid.

**SpacialTransform**

*(spacial_prob: float, stretch_prob: float, crop_size: Sequence, min_scale: float = -0.2, max_scale: float = 0.5, max_stretch: float = 0.2)*

Spacial Transform API for RAFT:

- **spacial_prob** *(float)* - probability to do spacial transform. Defaults to 0.
- **stretch_prob** *(float)* - probability to do stretch. Defaults to 0.
- **crop_size** *(tuple, list)* - the base size for resize.
- **min_scale** *(float)* - the exponent for min scale. Defaults to -0.2.
- **max_scale** *(float)* - the exponent for max scale. Defaults to 0.5.
- **max_stretch** *(float)* - the base size for resize.

**Returns**

Resized results, 'img_shape'.

**Return type**

dict

**resize_sparse_flow_map** *(flow: numpy.ndarray, valid: numpy.ndarray, fx: float = 1.0, fy: float = 1.0, x0: int = 0, y0: int = 0)*

Resize sparse optical flow function.

**Parameters**

- **flow** *(ndarray)* - optical flow data will be resized.
- **valid** *(ndarray)* - valid mask for sparse optical flow.
- **fx** *(float, optional)* - horizontal scale factor. Defaults to 1.0.
- **fy** *(float, optional)* - vertical scale factor. Defaults to 1.0.
- **x0** *(int, optional)* - abscissa of left-top point where the flow map will be crop from. Defaults to 0.
- **y0** *(int, optional)* - ordinate of left-top point where the flow map will be crop from. Defaults to 0.

**Returns**

the transformed flow map and valid mask.

**Return type**

Sequence[numpy.ndarray]

**spacial_transform** *(imgs: numpy.ndarray)*

Spacial transform function.

**Parameters**

- **imgs** *(ndarray)* - the images that will be transformed.

**Returns**

the transformed images, horizontal scale factor, vertical scale factor, coordinate of left-top point where the image maps will be crop from.
Return type  Tuple[array, float, float, int, int]

class mmflow.datasets.ToDataContainer(fields: collections.abc.Sequence = ([
    {'key': 'img1', 'stack': True},
    {'key': 'img2', 'stack': True},
    {'key': 'flow_gt'})])

Convert results to mmcv.DataContainer by given fields.

Parameters  fields (Sequence[dict]) – Each field is a dict like dict(key='xxx', **kwargs). The key in result will be converted to mmcv.DataContainer with **kwargs. Default: (dict(key='img1', stack=True), dict(key='img2', stack=True), dict(key='flow_gt')).

class mmflow.datasets.ToTensor(keys: collections.abc.Sequence)

Convert some results to torch.Tensor by given keys.

Parameters  keys (Sequence[str]) – Keys that need to be converted to Tensor.

class mmflow.datasets.Transpose(keys: collections.abc.Sequence, order: collections.abc.Sequence)

Transpose some results by given keys.

Parameters  • keys (Sequence[str]) – Keys of results to be transposed.
  • order (Sequence[int]) – Order of transpose.

class mmflow.datasets.Validation(max_flow: Union[float, int])

This Validation transform from RAFT is for return a mask for the flow is less than max_flow.

Parameters  max_flow (float, int) – the max flow for validated flow.

Returns  Resized results, ‘valid’ and ‘max_flow’ keys are added into result dict.

Return type  dict

mmflow.datasets.build_dataloader(dataset: torch.utils.data.dataset.Dataset, samples_per_gpu: int, workers_per_gpu: int, sample_ratio: Optional[Sequence] = None, num_gpus: int = 1, dist: bool = True, shuffle: bool = True, seed: Optional[int] = None, persistent_workers: bool = False, **kwargs)

Build PyTorch DataLoader.

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

Parameters  • dataset (Dataset) – A PyTorch dataset.
  • samples_per_gpu (int) – Number of training samples on each GPU, i.e., batch size of each GPU.
  • workers_per_gpu (int) – How many subprocesses to use for data loading for each GPU.
  • sample_ratio (list, optional) – The ratio for samples in mixed branch, sum of sample_ratio must be equal to 1. and the length must be equal to the length of datasets, e.g. branch=8, sample_ratio=(0.5,0.25,0.25) means in one branch 4 samples from dataset1, 2 samples from dataset2 and 2 samples from dataset3.
  • num_gpus (int) – Number of GPUs. Only used in non-distributed training.
  • dist (bool) – Distributed training/test or not. Default: True.
  • shuffle (bool) – Whether to shuffle the data at every epoch. Default: True.
mmflow

- **seed** (*int, optional*) – the seed for generating random numbers for data workers. Default to None.
- **persistent_workers** (*bool*) – If True, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers Dataset instances alive. The argument also has effect in PyTorch>=1.7.0. Default: False.
- **kwargs** – any keyword argument to be used to initialize DataLoader

**Returns** A PyTorch dataloader.

**Return type** DataLoader

    default_args: Optional[dict] = None*) → torch.utils.data.dataset.Dataset

Build Pytorch dataset.

**Parameters**

- **cfg** (*mmcv.Config*) – Config dict of dataset or list of config dict. It should at least contain the key “type”.
- **default_args** (*dict, optional*) – Default initialization arguments.

**Note:** If the input config is a list, this function will concatenate them automatically.

**Returns** The built dataset based on the input config.

**Return type** dataset

mmflow.datasets.read_flow(*name: str*) → numpy.ndarray

Read flow file with the suffix ‘.flo’.

This function is modified from https://lmb.informatik.uni-freiburg.de/resources/datasets/IO.py Copyright (c) 2011, LMB, University of Freiburg.

**Parameters** **name** (*str*) – Optical flow file path.

**Returns** Optical flow

**Return type** ndarray

mmflow.datasets.read_flow_kitti(*name: str*) → Tuple[numpy.ndarray, numpy.ndarray]

Read sparse flow file from KITTI dataset.

This function is modified from https://github.com/princeton-vl/RAFT/blob/master/core/utils/frame_utils.py. Copyright (c) 2020, princeton-vl Licensed under the BSD 3-Clause License

**Parameters** **name** (*str*) – The flow file

**Returns** flow and valid map

**Return type** Tuple[ndarray, ndarray]

mmflow.datasets.render_color_wheel(*save_file: str = ‘color_wheel.png’*) → numpy.ndarray

Render color wheel.

**Parameters** **save_file** (*str*) – The saved file name. Defaults to ‘color_wheel.png’.

**Returns** color wheel image.

**Return type** ndarray

17.1. datasets
**mmflow**

*mmflow.datasets.visualize_flow*(flow: numpy.ndarray, save_file: Optional[str] = None) → numpy.ndarray

Flow visualization function.

**Parameters**
- `flow` (ndarray) – The flow will be rendered.
- `save_dir` ([type], optional) – save dir. Defaults to None.

**Returns** flow map image with RGB order.

**Return type** ndarray

*mmflow.datasets.write_flow*(flow: numpy.ndarray, flow_file: str) → None

Write the flow in disk.

This function is modified from https://lmb.informatik.uni-freiburg.de/resources/datasets/IO.py Copyright (c) 2011, LMB, University of Freiburg.

**Parameters**
- `flow` (ndarray) – The optical flow that will be saved.
- `flow_file` (str) – The file for saving optical flow.

*mmflow.datasets.write_flow_kitti*(uv: numpy.ndarray, filename: str)

Write the flow in disk.

This function is modified from https://github.com/princeton-vl/RAFT/blob/master/core/utils/frame_utils.py. Copyright (c) 2020, princeton-vl Licensed under the BSD 3-Clause License

**Parameters**
- `uv` (ndarray) – The optical flow that will be saved.
- `filename` ([type]) – The file for saving optical flow.

### 17.2 pipelines

**class mmflow.datasets.pipelines.Collect**(keys: collections.abc.Sequence, meta_keys: collections.abc.Sequence = ('filename1', 'filename2', 'ori_filename1', 'ori_filename2', 'filename_flow', 'ori_filename_flow', 'ori_shape', 'img_shape', 'pad_shape', 'scale_factor', 'flip', 'flip_direction', 'img_norm_cfg'))

Collect data from the loader relevant to the specific task.

This is usually the last stage of the data loader pipeline. Typically keys is set to some subset of “img”, “flow_gt”.

The “img_meta” item is always populated. The contents of the “img_meta” dictionary depends on “meta_keys”. By default this includes:

- “**img_shape**”: shape of the image input to the network as a tuple (h, w, c). Note that images may be zero padded on the bottom/right if the batch tensor is larger than this shape.
- “**scale_factor**”: a float indicating the preprocessing scale
- “**flip**”: a boolean indicating if image flip transform was used
- “**filename1**”: path to the image1 file
- “**filename2**”: path to the image2 file
- “**ori_filename1**”: image1 file name
• “ori_filename2”: image2 file name
• “ori_shape”: original shape of the image as a tuple (h, w, c)
• “pad_shape”: image shape after padding
• “img_norm_cfg”: a dict of normalization information:
  – mean - per channel mean subtraction
  – std - per channel std divisor
  – to_rgb - bool indicating if bgr was converted to rgb

Parameters

• keys (Sequence[str]) – Keys of results to be collected in data.
• meta_keys (Sequence[str], optional) – Meta keys to be converted to mmcv.DataContainer and collected in data[img_metas]. Default: ('filename1', 'filename2', 'ori_filename1', 'ori_filename2', 'ori_shape', 'img_shape', 'pad_shape', 'scale_factor', 'flip', 'flip_direction', 'img_norm_cfg')

class mmflow.datasets.pipelines.ColorJitter(asymmetric_prob=0.0, brightness=0.0, contrast=0.0, saturation=0.0, hue=0.0)
Randomly change the brightness, contrast, saturation and hue of an image. :param asymmetric_prob: the probability to do color jitter for two images asymmetrically.

Parameters

• brightness (float, tuple) – 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, tuple) – 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, tuple) – 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, tuple) – 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.

class mmflow.datasets.pipelines.Compose(transforms: Sequence)
Compose multiple transforms sequentially.

Parameters transforms (Sequence[dict | callable]) – Sequence of transform object or config dict to be composed.

class mmflow.datasets.pipelines.DefaultFormatBundle
Default formatting bundle.

It simplifies the pipeline of formatting common fields, including “img” and “flow_gt”. These fields are formatted as follows.

• img1: (1) transpose, (2) to tensor, (3) to DataContainer (stack=True)
• img2: (1) transpose, (2) to tensor, (3) to DataContainer (stack=True)
• flow_gt: (1) transpose, (2) to tensor, (3) to DataContainer (stack=True)

```python
class mmflow.datasets.pipelines.Erase(prob: float, bounds: Sequence = [50, 100], max_num: int = 3)
```

Erase transform from RAFT is randomly erasing rectangular regions in img2 to simulate occlusions.

**Parameters**
- `prob (float)` – the probability for erase transform.
- `bounds (list, tuple)` – the bounds for erase regions (bound_x, bound_y).
- `max_num (int)` – the max number of erase regions.

**Returns** revised results, ‘img2’ and ‘erase_num’ are added into results.

```python
class mmflow.datasets.pipelines.GaussianNoise(sigma_range=(0, 0.04), clamp_range=()
```

Add Gaussian Noise to images.

Add Gaussian Noise, with mean 0 and std sigma uniformly sampled from sigma_range, to images. And then clamp the images to clamp_range.

**Parameters**
- `sigma_range (list(float) | tuple(float))` – Uniformly sample sigma of gaussian noise in sigma_range. Default: (0, 0.04)
- `clamp_range (list(float) | tuple(float))` – The min and max value to clamp the images after adding gaussian noise. Default: (float('-inf'), float('inf'))

```python
class mmflow.datasets.pipelines.ImageToTensor(keys: collections.abc.Sequence)
```

Convert image to `torch.Tensor` by given keys.

The dimension order of input image is (H, W, C). The pipeline will convert it to (C, H, W). If only 2 dimension (H, W) is given, the output would be (1, H, W).

**Parameters**
- `keys (Sequence[str])` – Key of images to be converted to Tensor.

```python
class mmflow.datasets.pipelines.InputPad(exponent, mode='edge', position='center', **kwargs)
```

Pad images such that dimensions are divisible by 2^n used in test.

**Parameters**
- `exponent (int)` – the exponent n of 2^n
- `mode (str)` – mode for numpy.pad(). Defaults to ‘edge’.
- `position (str)` – ‘center’, ‘left’, ‘right’, ‘top’ and ‘down’. Defaults to ‘center’

```python
class mmflow.datasets.pipelines.InputResize(exponent)
```

Resize images such that dimensions are divisible by 2^n used in test.

**Returns**
- Resized results, ‘img_shape’, ‘scale_factor’ keys are added into result dict.

**Return type** dict

```python
class mmflow.datasets.pipelines.LoadAnnotations(with_occ: bool = False, sparse: bool = False,
```

Load optical flow from file.

**Parameters**
- `with_occ (bool)` – whether to parse and load occlusion mask. Default to False.
class mmflow.datasets.pipelines.LoadImageFromFile(
to_float32: bool = False, color_type: str = 'color',
file_client_args: dict = dict(backend='disk'),
imdecode_backend: str = 'cv2')

Load image1 and image2 from file.

Required keys are “img1_info” (dict that must contain the key “filename” and “filename2”). Added or updated keys are “img1”, “img2”, “img_shape”, “ori_shape” (same as img_shape), “pad_shape” (same as img_shape), “scale_factor” (1.0, 1.0) and “img_norm_cfg” (means=0 and stds=1).

Parameters

- **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.
- **color_type (str)** – The flag argument for mmcv.imfrombytes(). Defaults to ‘color’.
- **file_client_args (dict)** – Arguments to instantiate a FileClient. See mmcv.fileio.FileClient for details. Defaults to dict(backend='disk').
- **imdecode_backend (str)** – Backend for mmcv.imdecode(). Default: ‘cv2’

class mmflow.datasets.pipelines.Normalize(mean, std, to_rgb=True)

Normalize the image.

Added key is “img_norm_cfg”. :param mean: Mean values of 3 channels. :type mean: sequence :param std: Std values of 3 channels. :type std: sequence :param to_rgb: Whether to convert the image from BGR to RGB, default is true.

class mmflow.datasets.pipelines.PhotoMetricDistortion(brightness_delta=32, contrast_range=(0.5, 1.5), saturation_range=(0.5, 1.5), hue_delta=18)

Apply photometric distortion to image sequentially, every transformation is applied with a probability of 0.5.

The position of random contrast is in second or second to last. 1. random brightness 2. random contrast (mode 0) 3. convert color from BGR to HSV 4. random saturation 5. random contrast (mode 1) 8. randomly swap channels :param brightness_delta: delta of brightness. :type brightness_delta: int :param contrast_range: range of contrast. :type contrast_range: tuple :param saturation_range: range of saturation. :type saturation_range: tuple :param hue_delta: delta of hue. :type hue_delta: int

**brightness**

Brightness distortion.

**contrast**

Contrast distortion.

**convert**

Multiple with alpha and add beat with clip.

**hue**

Hue distortion.

**saturation**

Saturation distortion.
class mmflow.datasets.pipelines.RandomAffine(
global_transform: Optional[dict] = None,
relative_transform: Optional[dict] = None,
preserve_valid: bool = True, check_bound: bool = False)

Random affine transformation of images, flow map and occlusion map (if available).

Keys of global_transform and relative_transform should be the subset of (‘translates’, ‘zoom’, ‘shear’, ‘rotate’).
And also, each key and its corresponding values has to satisfy the following rules:

- **translates**: the translation ratios along x axis and y axis. **Defaults** to (0., 0.).
- zoom: the min and max zoom ratios. Defaults to (1.0, 1.0).
- shear: the min and max shear ratios. Defaults to (1.0, 1.0).
- rotate: the min and max rotate degree. Defaults to (0., 0.).

Parameters

- **global_transform** (dict) – A dict which contains keys: transform, zoom, shear, rotate.
global_transform will transform both img1 and img2.
- **relative_transform** (dict) – A dict which contains keys: transform, zoom, shear, rotate.
relative_transform will only transform img2 after global_transform to both images.
- **preserve_valid** (bool) – Whether continue transforming until both images are valid. A
valid affine transform is an affine transform which guarantees the transformed image covers
the whole original picture frame. Defaults to True.
- **check_bound** (bool) – Whether to check out of bound for transformed occlusion maps.
If True, all pixels in borders of img1 but not in borders of img2 will be marked occluded.
Defaults to False.

class mmflow.datasets.pipelines.RandomCrop(crop_size)

Random crop the image & flow.

Parameters **crop_size** (tuple) – Expected size after cropping, (h, w).

crop(img, crop_bbox)

Crop from img

generate_crop_bbox(img_shape)

Randomly get a crop bounding box.

class mmflow.datasets.pipelines.RandomFlip(prob, direction='horizontal')

Flip the image and flow map.

Parameters

- **prob** (float) – The flipping probability.
- **direction** (str) – The flipping direction. Options are ‘horizontal’ and ‘vertical’. Default:
  ‘horizontal’.

class mmflow.datasets.pipelines.RandomRotation(prob, angle, auto_bound=False)

Random rotation of the image from -angle to angle (in degrees).

optical flow data.

Parameters

- **prob** (float) – The rotation probability.
- **angle** (float) – max angle of the rotation in the range from -180 to 180.
• **auto_bound** (bool) – Whether to adjust the image size to cover the whole rotated image. Default: False

class mmflow.datasets.pipelines.RandomTranslate(prob=0.0, x_offset=0.0, y_offset=0.0)
Random translation of the images and flow map.

**Parameters**

- **prob** (float) – the probability to do translation.
- **x_offset** (float / tuple) – translate ratio on x axis, randomly choice [-x_offset, x_offset] or the given [min, max]. Default: 0.
- **y_offset** (float / tuple) – translate ratio on y axis, randomly choice [-x_offset, x_offset] or the given [min, max]. Default: 0.

class mmflow.datasets.pipelines.Rerange(min_value=0, max_value=255)
Rerange the image pixel value.

**Parameters**

- **min_value** (float or int) – Minimum value of the reranged image. Default: 0.
- **max_value** (float or int) – Maximum value of the reranged image. Default: 255.

class mmflow.datasets.pipelines.SpatialTransform(spacial_prob: float, stretch_prob: float, crop_size: Sequence, min_scale: float = -0.2, max_scale: float = 0.5, max_stretch: float = 0.2)

**Returns** Resized results, ‘img_shape’;

**Return type** dict

resize_sparse_flow_map(flow: numpy.ndarray, valid: numpy.ndarray, fx: float = 1.0, fy: float = 1.0, x0: int = 0, y0: int = 0) → Sequence[numpy.ndarray]
Resize sparse optical flow function.

**Parameters**

- **flow** (ndarray) – optical flow data will be resized.
- **valid** (ndarray) – valid mask for sparse optical flow.
- **fx** (float, optional) – horizontal scale factor. Defaults to 1.0.
- **fy** (float, optional) – vertical scale factor. Defaults to 1.0.
- **x0** (int, optional) – abscissa of left-top point where the flow map will be crop from. Defaults to 0.
- **y0** (int, optional) – ordinate of left-top point where the flow map will be crop from. Defaults to 0.

**Returns** the transformed flow map and valid mask.

**Return type** Sequence[ndarray]

spacial_transform(imgs: numpy.ndarray) → Tuple[numpy.ndarray, float, int, int]
Spacial transform function.
Parameters `imgs` (ndarray) – the images that will be transformed.

Returns

the transformed images, horizontal scale factor, vertical scale factor, coordinate of left-top point where the image maps will be crop from.

Return type Tuple[ndarray, float, float, int, int]

class mmflow.datasets.pipelines.TestFormatBundle

Default formatting bundle.

It simplifies the pipeline of formatting common fields, including “img1” and “img2”. These fields are formatted as follows.

• img1: (1)transpose, (2)to tensor, (3)to DataContainer (stack=True)
• img2: (1)transpose, (2)to tensor, (3)to DataContainer (stack=True)

class mmflow.datasets.pipelines.ToDataContainer(fields: collections.abc.Sequence = ([dict(key='img1', stack=True), dict(key='img2', stack=True), dict(key='flow_gt')]))

Convert results to mmcv.DataContainer by given fields.

Parameters `fields` (Sequence[dict]) – Each field is a dict like dict(key='xxx', **kwargs). The key in result will be converted to mmcv.DataContainer with **kwargs. Default: (dict(key='img1', stack=True), dict(key='img2', stack=True), dict(key='flow_gt')).

class mmflow.datasets.pipelines.ToTensor(keys: collections.abc.Sequence)

Convert some results to torch.Tensor by given keys.

Parameters `keys` (Sequence[str]) – Keys that need to be converted to Tensor.

class mmflow.datasets.pipelines.Transpose(keys: collections.abc.Sequence, order: collections.abc.Sequence)

Transpose some results by given keys.

Parameters

• `keys` (Sequence[str]) – Keys of results to be transposed.
• `order` (Sequence[int]) – Order of transpose.

class mmflow.datasets.pipelines.Validation(max_flow: Union[float, int])

This Validation transform from RAFT is for return a mask for the flow is less than max_flow.

Parameters `max_flow` (float, int) – the max flow for validated flow.

Returns

Resized results, ‘valid’ and ‘max_flow’ keys are added into result dict.

Return type dict
18.1 encoders

18.2 decoders

18.3 flow_estimators

18.4 losses
CHAPTER

NINETEEN

INDICES AND TABLES

• genindex
• modindex
• search
m
mmflow.core.evaluation, 59
mmflow.core.hooks, 62
mmflow.datasets, 65
mmflow.datasets.pipelines, 76
INDEX

A
after_train_epoch() (mm-flow.core.evaluation.EvalHook method), 60
after_train_iter() (mm-flow.core.evaluation.EvalHook method), 60

B
before_run() (mmflow.core.hooks.LiteFlowNetStageLoadHook method), 62
brightness() (mmflow.datasets.PhotoMetricDistortion method), 71
brightness() (mmflow.datasets.pipelines.PhotoMetricDistortion method), 79
build_dataloader() (in module mmflow.datasets), 74
build_dataset() (in module mmflow.datasets), 75

C
ChairsSDHom (class in mmflow.datasets), 65
Collect (class in mmflow.datasets), 65
Collect (class in mmflow.datasets.pipelines), 76
ColorJitter (class in mmflow.datasets), 66
ColorJitter (class in mmflow.datasets.pipelines), 77
Compose (class in mmflow.datasets), 66
Compose (class in mmflow.datasets.pipelines), 77
ConcatDataset (class in mmflow.datasets), 66
contrast() (mmflow.datasets.PhotoMetricDistortion method), 71
contrast() (mmflow.datasets.pipelines.PhotoMetricDistortion method), 79
convert() (mmflow.datasets.PhotoMetricDistortion method), 71
convert() (mmflow.datasets.pipelines.PhotoMetricDistortion method), 79
crop() (mmflow.datasets.pipelines.RandomCrop method), 80
crop() (mmflow.datasets.RandomCrop method), 71

D
DefaultFormatBundle (class in mmflow.datasets), 66
DefaultFormatBundle (class in mmflow.datasets.pipelines), 77
DistEvalHook (class in mmflow.core.evaluation), 59
DistributedSampler (class in mmflow.datasets), 67

E
evaluate() (mmflow.core.evaluation.DistEvalHook method), 59
evaluate() (mmflow.core.evaluation.EvalHook method), 60
evaluate() (mmflow.datasets.ConcatDataset method), 66
end_point_error() (in module mmflow.core.evaluation), 60
end_point_error_map() (in module mmflow.core.evaluation), 60
Erase (class in mmflow.datasets), 67
EvalHook (class in mmflow.core.evaluation), 59
eval_metrics() (in module mmflow.core.evaluation), 60
EvalHook (class in mmflow.core.evaluation.DistEvalHook method), 59

evaluate() (mmflow.core.evaluation.EvalHook method), 60
evaluate() (mmflow.datasets.ConcatDataset method), 66

F
FlyingChairs (class in mmflow.datasets), 67
FlyingChairsOcc (class in mmflow.datasets), 68
FlyingThings3D (class in mmflow.datasets), 68
FlyingThings3DSubset (class in mmflow.datasets), 68

G
GaussianNoise (class in mmflow.datasets), 69
GaussianNoise (class in mmflow.datasets.pipelines), 78
get_crop_bbox() (mmflow.datasets.pipelines.RandomCrop method), 80
get_crop_bbox() (mmflow.datasets.RandomCrop method), 72
get_lr() (mmflow.core.hooks.MultiStageLrUpdaterHook method), 62

H
HD1K (class in mmflow.datasets), 69
mmflow

hue() (mmflow.datasets.PhotoMetricDistortion method), 71
hue() (mmflow.datasets.pipelines.PhotoMetricDistortion method), 79

ImageToTensor (class in mmflow.datasets), 69
ImageToTensor (class in mmflow.datasets.pipelines), 78
InputPad (class in mmflow.datasets), 69
InputPad (class in mmflow.datasets.pipelines), 78
InputResize (class in mmflow.datasets), 69
InputResize (class in mmflow.datasets.pipelines), 78

K
KITTI2012 (class in mmflow.datasets), 69
KITTI2015 (class in mmflow.datasets), 69

L
LiteFlowNetStageLoadHook (class in mmflow.core.hooks), 62
load_ann_info() (mmflow.datasets.FlyingChairs method), 67
load_ann_info() (mmflow.datasets.FlyingChairsOcc method), 68
load_data_info() (mmflow.datasets.ChairsSDHom method), 65
load_data_info() (mmflow.datasets.FlyingChairs method), 67
load_data_info() (mmflow.datasets.FlyingChairsOcc method), 68
load_data_info() (mmflow.datasets.FlyingThings3D method), 68
load_data_info() (mmflow.datasets.FlyingThings3DSubset method), 69
load_data_info() (mmflow.datasets.HD1K method), 69
load_data_info() (mmflow.datasets.KITTI2012 method), 69
load_data_info() (mmflow.datasets.KITTI2015 method), 69
load_data_info() (mmflow.datasets.Sintel method), 73
load_img_info() (mmflow.datasets.FlyingChairs method), 67
load_img_info() (mmflow.datasets.FlyingChairsOcc method), 68
LoadAnnotations (class in mmflow.datasets.pipelines), 78
LoadImageFromFile (class in mmflow.datasets), 69
LoadImageFromFile (class in mmflow.datasets.pipelines), 79

M
MixedBatchDistributedSampler (class in mmflow.datasets), 70

mmflow.core.evaluation module, 59
mmflow.core.hooks module, 62
mmflow.datasets module, 65
mmflow.datasets.pipelines module, 76
module

mmflow.core.evaluation, 59
mmflow.core.hooks, 62
mmflow.datasets, 65
mmflow.datasets.pipelines, 76
multi_gpu_online_evaluation() (in module mmflow.core.evaluation), 60
MultiStageLrUpdaterHook (class in mmflow.core.hooks), 62

N
Normalize (class in mmflow.datasets), 70
Normalize (class in mmflow.datasets.pipelines), 79

O
online_evaluation() (in module mmflow.core.evaluation), 61
optical_flow_outliers() (in module mmflow.core.evaluation), 61

P
PhotoMetricDistortion (class in mmflow.datasets), 70
PhotoMetricDistortion (class in mmflow.datasets.pipelines), 79
pre_pipeline() (mmflow.datasets.Sintel method), 73

R
RandomAffine (class in mmflow.datasets), 71
RandomAffine (class in mmflow.datasets.pipelines), 79
RandomCrop (class in mmflow.datasets), 71
RandomCrop (class in mmflow.datasets.pipelines), 80
RandomFlip (class in mmflow.datasets), 72
RandomFlip (class in mmflow.datasets.pipelines), 80
RandomRotation (class in mmflow.datasets), 72
RandomRotation (class in mmflow.datasets.pipelines), 80
RandomTranslate (class in mmflow.datasets), 72
RandomTranslate (class in mmflow.datasets.pipelines), 81
read_flow() (in module mmflow.datasets), 75
read_flow_kitti() (in module mmflow.datasets), 75
render_color_wheel() (in module mmflow.datasets), 75
RepeatDataset (class in mmflow.datasets), 72
Rerange (class in mmflow.datasets), 72
Rerange (class in mmflow.datasets.pipelines), 81
resize_sparse_flow_map() (mmflow.datasets.pipelines.SpacialTransform method), 81
resize_sparse_flow_map() (mmflow.datasets.SpacialTransform method), 73

S
saturation() (mmflow.datasets.PhotoMetricDistortion method), 71
saturation() (mmflow.datasets.pipelines.PhotoMetricDistortion method), 79
set_epoch() (mmflow.datasets.MixedBatchDistributedSampler method), 70
single_gpu_online_evaluation() (in module mmflow.core.evaluation), 61
Sintel (class in mmflow.datasets), 72
spacial_transform() (mmflow.datasets.pipelines.SpacialTransform method), 81
spacial_transform() (mmflow.datasets.SpacialTransform method), 73
SpacialTransform (class in mmflow.datasets), 73
SpacialTransform (class in mmflow.datasets.pipelines), 81

T
TestFormatBundle (class in mmflow.datasets.pipelines), 82
ToDataContainer (class in mmflow.datasets), 74
ToDataContainer (class in mmflow.datasets.pipelines), 82
ToTensor (class in mmflow.datasets), 74
ToTensor (class in mmflow.datasets.pipelines), 82
Transpose (class in mmflow.datasets), 74
Transpose (class in mmflow.datasets.pipelines), 82

V
Validation (class in mmflow.datasets), 74
Validation (class in mmflow.datasets.pipelines), 82
visualize_flow() (in module mmflow.datasets), 75

W
write_flow() (in module mmflow.datasets), 76
write_flow_kitti() (in module mmflow.datasets), 76