mmflow

MMFlow Author

Feb 07, 2023

GET STARTED

CHAPTER

ONE

OVERVIEW

This chapter introduces you to the framework of MMFlow, the basic conception of optical flow, 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 RHxWx3, the flow field U RHxWx2 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(TODO)

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 7 main parts, `apis`, `structures`, `datasets`, `models`, `engine`, `evaluation` and `visualization`.

- `apis`, provides high-level APIs for models training, testing, and inference,
- `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.
- `engine`
- `evaluation`
- `visualization`

1.3 How to Use this Guide(TODO)

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

1. For installation instructions, please see get_started.
2. Refer to the below tutorials to for the basic usage of MMFlow.
    - config
    - dataset prepare
    - inference
    - train and test
3. Refer to the below tutorials to dive deeper:

1.4 References

1. Michael Black, Optical flow: The "good parts" version, Machine Learning Summer School (MLSS), Tübiungen, 2013.
2. Black M J. Robust incremental optical flow[D]. Yale University, 1992.
3. Butler D J, Wulff J, Stanley G B, et al. A naturalistic open source movie for optical flow evaluation[C]//European conference on computer vision. Springer, Berlin, Heidelberg, 2012: 611-625.
4. Wulff J, Butler D J, Stanley G B, et al. Lessons and insights from creating a synthetic optical flow benchmark[C]//European Conference on Computer Vision. Springer, Berlin, Heidelberg, 2012: 168-177.
5. Horn B, Klaus B, Horn P. Robot vision[M]. MIT Press, 1986.
6. Sun D, Yang X, Liu M Y, et al. Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2018: 8934-8943.
7. Geiger A, Lenz P, Urtasun R. Are we ready for autonomous driving? the kitti vision benchmark suite[C]//2012 IEEE conference on computer vision and pattern recognition. IEEE, 2012: 3354-3361.
8. Menze M, Heipke C, Geiger A. Object scene flow[J]. ISPRS Journal of Photogrammetry and Remote Sensing, 2018, 140: 60-76.
9. Menze M, Heipke C, Geiger A. Joint 3d estimation of vehicles and scene flow[J]. ISPRS annals of the photogrammetry, remote sensing and spatial information sciences, 2015, 2: 427.

CHAPTER

TWO

GET STARTED: INSTALL AND RUN MMFLOW

2.1 Prerequisites

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
```

2.2 Installation

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

2.2.1 Best Practices

Step 0. Install MMCV using MIM.

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

Step 1. Install MMFlow.

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

```
git clone https://github.com/open-mmlab/mmflow.git
cd mmflow
git checkout dev-1.x
# branch 'dev-1.x' set up to track remote branch 'dev-1.x' from 'origin'.
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:

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

2.3 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.

```
mim download mmflow --config pwcnet-ft_4xb1_300k_sintel-final-384x768.py
```

The downloading will take several seconds or more, depending on your network environment. When it is done, you will find two files `pwcnet-ft_4xb1_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.

```
   python demo/image_demo.py demo/frame_0001.png demo/frame_0002.png \
       configs/pwcnet/pwcnet_ft_4x1_300k_sintel_final_384x768.py \
       checkpoints/pwcnet_ft_4x1_300k_sintel_final_384x768.pth results
```

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.

```
from mmflow.apis import inference_model, init_model
from mmengine.registry import init_default_scope

init_default_scope('mmflow')
config_file = 'pwcnet-ft_4xb1_300k_sintel-final-384x768.py'
checkpoint_file = 'pwcnet_ft_4x1_300k_sintel_final_384x768.pth'
```

(continues on next page)

(continued from previous page)

```
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.

2.4 Customize Installation

2.4.1 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.

2.4.2 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.

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

2.4.3 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.

Operator	Model
Correlation	PWC-Net, FlowNetC, FlowNet2, IRR-PWC, LiteFlowNet, LiteFlowNet2, MaskFlowNet

2.4.4 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>=2.0.0rc1
```

Step 2. Install MMFlow from the source.

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

Step 3. Verification.

```
import mmflow
print(mmflow.__version__)
# Example output: 1.0.0rc0
```

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.

2.4.5 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
```

2.5 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.

CHAPTER

THREE

TRAIN & TEST

3.1 Tutorial 1: Learn about Configs

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

3.1.1 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 based on PWC-Net, users may first inherit the basic PWC-Net structure by specifying `_base_ = ../pwcnet/pwcnet_8xb1_slong_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 mmengine for detailed documentation.

3.1.2 Config File Naming Convention

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

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

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

- `{model}`: model type like `pwcnet`, `flownets`, etc.
- `[gpu x batch_per_gpu]`: GPUs and samples per GPU, like `8xb1`.
- `{schedule}`: training schedule. Following FlowNet2's convention, we use `slong`, `sfine` and `sshort`, or number of iteration like `150k` 150k(iterations).
- `{training datasets}`: training dataset like `flyingchairs`, `flyingthings3d_subset`, `flyingthings3d`.
- `[input_size]`: the size of training images.

3.1.3 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 MMEngine.

```
_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/pwcnet.py is a basic model cfg file for PWC-Net.

```
model = dict(
    type='PWCNet',  # The algorithm name.
    data_preprocessor=dict(  # The config of data preprocessor, usually includes image␣
↪normalization and augmentation.
        type='FlowDataPreprocessor',  # The type of data preprocessor.
        mean=[0., 0., 0.],  # Mean values used for normalizing the input images.
        std=[255., 255., 255.],  # Standard variance used for normalizing the input␣
↪images.
        bgr_to_rgb=False,  # Whether to convert image from BGR to RGB.
        sigma_range=(0, 0.04),  # Add gaussian noise for data augmentation, the sigma is␣
↪uniformly sampled from [0, 0.04].
        clamp_range=(0., 1.)),  # After adding gaussian noise, clamp the range to [0., 1.
↪].
    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, then 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␣
↪of each pyramid level.
        strides=(2, 2, 2, 2, 2, 2),  # List of strides of each pyramid level.
        dilations=(1, 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(
            level6=81, level5=213, level4=181, level3=149, level2=117),  # Input␣
↪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', align_corners=True, use_mask=True),
        act_cfg=dict(type='LeakyReLU', negative_slope=0.1),
        scaled=False,  # Whether to use scaled correlation by the number of elements␣
↪involved to calculate correlation or not.
        post_processor=dict(type='ContextNet', in_channels=565),  # The configuration␣
↪for post processor.
```

(continues on next page)

(continued from previous page)

```
        flow_loss=dict(
            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,
                '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))
randomness = dict(seed=0, diff_rank_seed=True)  # Random seed.
```

in _base_/datasets/flyingchairs_384x448.py

```
dataset_type = 'FlyingChairs'  # Dataset type, which will be used to define the dataset.
data_root = 'data/FlyingChairs_release'  # Root path of the dataset.

# global_transform and relative_transform are intermediate variables used in RandomAffine
# 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.).
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))

backend_args = dict(backend='local')  # File client arguments.

train_pipeline = [  # Training pipeline.
```

(continues on next page)

(continued from previous page)

```
    dict(type='LoadImageFromFile', backend_args=backend_args),  # Load images.
    dict(type='LoadAnnotations', backend_args=backend_args),  # 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='RandomFlip', prob=0.5, direction='horizontal'),  # Random horizontal flip.
    dict(type='RandomFlip', prob=0.5, direction='vertical'),  # Random vertical flip.
    dict(
        type='RandomAffine',  # Random affine transformation of images.
        global_transform=global_transform,  # See comments above for global_transform.
        relative_transform=relative_transform),  # See comments above for relative_
↪transform.
    dict(type='RandomCrop', crop_size=(384, 448)),  # Random crop the image and flow as␣
↪(384, 448).
    dict(type='PackFlowInputs')  # Format the annotation data and decide which keys in␣
↪the data should be packed into data_samples.
]

test_pipeline = [  # Testing pipeline.
    dict(type='LoadImageFromFile'),  # Load images.
    dict(type='LoadAnnotations'),  # Load flow data.
    dict(type='InputResize', exponent=6),  # Resize the width and height of the input␣
↪images to a multiple of 2^6.
    dict(type='PackFlowInputs')  # Format the annotation data and decide which keys in␣
↪the data should be packed into data_samples.
]

# flyingchairs_train and flyingchairs_test are intermediate variables used in dataloader,
# they define the type, pipeline, root path and split file of FlyingChairs.
flyingchairs_train = dict(
    type=dataset_type,
    pipeline=train_pipeline,
    data_root=data_root,
    split_file='data/FlyingChairs_release/FlyingChairs_train_val.txt')  # train-
↪validation split file.
flyingchairs_test = dict(
    type=dataset_type,
    pipeline=test_pipeline,
    data_root=data_root,
    test_mode=True,  # Use test set.
    split_file='data/FlyingChairs_release/FlyingChairs_train_val.txt')  # train-
↪validation split file

train_dataloader = dict(
    batch_size=1,  # Batch size of a single GPU.
    num_workers=2,  # Worker to pre-fetch data for each single GPU.
```

(continues on next page)

(continued from previous page)

```
    sampler=dict(type='InfiniteSampler', shuffle=True),  # Randomly shuffle during␣
↪training.
    drop_last=True,  # Drop the last non-full batch during training.
    persistent_workers=True,  # Shut down the worker processes after an epoch end, which␣
↪can accelerate training speed.
    dataset=flyingchairs_train)

val_dataloader = dict(
    batch_size=1,  # Batch size of a single GPU.
    num_workers=2,  # Worker to pre-fetch data for each single GPU.
    sampler=dict(type='DefaultSampler', shuffle=False),  # Not shuffle during validation␣
↪and testing.
    drop_last=False,  # No need to drop the last non-full batch.
    persistent_workers=True,  # Shut down the worker processes after an epoch end, which␣
↪can accelerate training speed.
    dataset=flyingchairs_test)
test_dataloader = val_dataloader

# The metric to measure the accuracy. Here, we use EnePointError.
val_evaluator = dict(type='EndPointError')
test_evaluator = val_evaluator
```

in _base_/schedules/schedule_s_long.py

```
# training schedule for S_long schedule
train_cfg = dict(by_epoch=False, max_iters=1200000, val_interval=100)
val_cfg = dict(type='ValLoop')
test_cfg = dict(type='TestLoop')

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

# learning policy
param_scheduler = dict(
    type='MultiStepLR',
    by_epoch=False,
    gamma=0.5,
    milestones=[400000, 600000, 800000, 1000000])

# default hooks
default_hooks = dict(
    timer=dict(type='IterTimerHook'),  # Log the time spent during iteration.
    logger=dict(type='LoggerHook', interval=50, log_metric_by_epoch=False),  # Collect␣
↪and write logs from different components of ``Runner``.
    param_scheduler=dict(type='ParamSchedulerHook'),  # update some hyper-parameters in␣
↪optimizer, e.g., learning rate.
    checkpoint=dict(type='CheckpointHook', interval=100000, by_epoch=False),  # Save␣
↪checkpoints periodically.
    sampler_seed=dict(type='DistSamplerSeedHook'),  # Data-loading sampler for␣
↪distributed training.
```

(continues on next page)

(continued from previous page)

```
    visualization=dict(type='FlowVisualizationHook'))  # Show or Write the predicted␣
↪results during the process of testing and validation.
```

in _base_/default_runtime.py

```
# Set the default scope of the registry to mmflow.
default_scope = 'mmflow'

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

# visualizer
vis_backends = [dict(type='LocalVisBackend')]  # The backend of visualizer.
visualizer = dict(
    type='FlowLocalVisualizer', vis_backends=vis_backends, name='visualizer')

resume = False  # Whether to resume from existed model.
```

3.1.4 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-options 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 `sigma_range=(0, 0.04)` in `data_preprocessor` of `model`. If you want to change this key, you may specify in two ways:

  1. `--cfg-options model.data_preprocessor.sigma_range="(0, 0.05)"`. Note that the quotation mark " is necessary to support list/tuple data types.
  2. `--cfg-options model.data_preprocessor.sigma_range=0,0.05`. Note that **NO** white space is allowed in the specified value. In addition, if the original type is tuple, it will be automatically converted to list after this way.

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

3.1.5 FAQ

Ignore some fields in the base configs

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

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

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. For example, the original `pwcnet_8xb1_slong_flyingchairs-384x448.py` is

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

According to the setting: `vis_backends = [dict(type='LocalVisBackend')]` in `_base_/default_runtime.py`, we can only store the visualization results locally. If we want to store them on Tensorboard as well, then the `vis_backends` is the intermediate variable we would like to modify.

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

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

We first define the new `vis_backends` and pass them into `visualizer` again.

3.2 Tutorial 2: Prepare Datasets

MMFlow supports multiple datasets. Please follow the corresponding guidelines for data preparation.

- FlyingChairs
- FlyingChairsOcc
- FlyingThings3D
- FlyingThings3D subset
- ChairsSDHom
- Sintel
- KITTI2015

- KITTI2012
- HD1K

3.3 Tutorial 3: Inference 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 use existing models to inference on given images. As for how to test existing models on standard datasets, please see this guide

3.3.1 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. Please download the pre-trained model to the path specified by `checkpoint_file` first.

```
from mmflow.apis import init_model, inference_model
from mmflow.datasets import visualize_flow, write_flow
from mmengine.registry import init_default_scope

# Specify the path to model config and checkpoint file
config_file = 'configs/pwcnet/pwcnet_8xb1_slong_flyingchairs-384x448.py'
checkpoint_file = 'checkpoints/pwcnet_8x1_slong_flyingchairs_384x448.pth'

# register all modules in mmflow into the registries
init_default_scope('mmflow')

# 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)

# The original `result` is a list, and the elements inside are of the `FlowDataSample`␣
↪data type
# get prediction from result and convert to np
result = result[0].pred_flow_fw.data.permute(1, 2, 0).cpu().numpy()

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

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

An image demo can be found in demo/image_demo.py.

3.4 Tutorial 4: Train and test with existing models

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 and test *predefined* models on standard datasets.

3.4.1 Train models on standard datasets

Modify training schedule

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

```
# training schedule for S_long schedule
train_cfg = dict(by_epoch=False, max_iters=1200000, val_interval=100)
val_cfg = dict(type='ValLoop')
test_cfg = dict(type='TestLoop')

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

# learning policy
param_scheduler = dict(
    type='MultiStepLR',
    by_epoch=False,
    gamma=0.5,
    milestones=[400000, 600000, 800000, 1000000])

# basic hooks
default_hooks = dict(
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=50, log_metric_by_epoch=False),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=100000, by_epoch=False),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    visualization=dict(type='FlowVisualizationHook'))
```

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.

```
# use the pre-trained model for the whole PWC-Net
load_from = 'https://download.openmmlab.com/mmflow/pwcnet/pwcnet_8x1_slong_flyingchairs_
↪384x448.pth'  # model path can be found in model zoo
```

Training on a single GPU

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

```
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.
- `--amp`: Use auto mixed precision training.
- `--resume ${CHECKPOINT_FILE}`: Resume from a previous checkpoint file.
- `--cfg-options ${OVERRIDE_CONFIGS}`: 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'. Please see this guide for more details.

Below is the optional arguments for multi-gpu test:

- `--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.

Note:

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

`--resume` 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 fine-tuning.

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.

```
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.

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.

```
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.

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
```

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.

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 ${CONFIG_FILE} ${GPUS}
```

On the second machine:

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

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

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.

   ```
   GPUS=4 GPUS_PER_NODE=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME} config1.py $
   ↪{WORK_DIR} --cfg-options env_cfg.dist_cfg.port=29500
   GPUS=4 GPUS_PER_NODE=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME} config2.py $
   ↪{WORK_DIR} --cfg-options env_cfg.dist_cfg.port=29501
   ```

2. Modify the config files to set different communication ports.

   In `config1.py`, set

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

   In `config2.py`, set

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

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

   ```
   GPUS=4 GPUS_PER_NODE=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME} config1.py $
   ↪{WORK_DIR}
   GPUS=4 GPUS_PER_NODE=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME} config2.py $
   ↪{WORK_DIR}
   ```

3.4.2 Test models on standard datasets

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. It should be pointed that only FlownetS, GMA and RAFT support testing on CPU.

```
# single-gpu testing
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--work-dir ${WORK_DIR}] \
    [--show ${SHOW_FLOW}] \
    [--show-dir ${VISUALIZATION_DIRECTORY}] \
    [--wait-time ${SHOW_INTERVAL}] \
    [--cfg-options ${OVERRIDE_CONFIGS}]

# CPU testing
export CUDA_VISIBLE_DEVICES=-1
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--work-dir ${WORK_DIR}] \
    [--show ${SHOW_FLOW}] \
    [--show-dir ${VISUALIZATION_DIRECTORY}] \
    [--wait-time ${SHOW_INTERVAL}] \
    [--cfg-options ${OVERRIDE_CONFIGS}]

# multi-gpu testing
bash tools/dist_test.sh \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${GPU_NUM} \
    [--work-dir ${WORK_DIR}] \
    [--cfg-options ${OVERRIDE_CONFIGS}]
```

`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.

```
[GPUS=${GPUS}] ./tools/slurm_test.sh ${PARTITION} ${JOB_NAME} \
    ${CONFIG_FILE} ${CHECKPOINT_FILE} \
    [--work-dir ${OUTPUT_DIRECTORY}] \
    [--cfg-options ${OVERRIDE_CONFIGS}]
```

Optional arguments:

- `--work-dir`: If specified, results will be saved in this directory. If not specified, the results will be automatically saved to `work_dirs/{CONFIG_NAME}`.
- `--show`: Show prediction results at runtime, available when `--show-dir` is not specified.
- `--show-dir`: If specified, the visualized optical flow map will be saved in the specified directory.
- `--wait-time`: The interval of show (s), which takes effect when `--show` is activated. Default to 2.
- `--cfg-options`: If specified, the key-value pair in xxx=yyy format will be merged into config file. For example, '–cfg-option model.encoder.in_channels=6'. Please see this guide for more details.

Below is the optional arguments for multi-gpu test:

- `--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 FlyingChairs without saving predicted flow files. The basic usage is as follows.

```
python tools/test.py configs/pwcnet/pwcnet_8x1_slong_flyingchairs_384x448.py \
    checkpoints/pwcnet_8x1_slong_flyingchairs_384x448.pth
```

Since `--work-dir` is not specified, the folder `work_dirs/pwcnet_8x1_slong_flyingchairs_384x448` will be created automatically to save the evaluation results.

If you want to show the predicted optical flow at runtime, just run

```
python tools/test.py configs/pwcnet/pwcnet_8x1_slong_flyingchairs_384x448.py \
    checkpoints/pwcnet_8x1_slong_flyingchairs_384x448.pth --show
```

Every image shown consists of two images, the ground truth on the left and the prediction result on the right. The image will be shown for 2 seconds, you can adjust `--wait-time` to change the display time. According to the default setting, the results are show every 50 results. If you want to change the frequency, for example, you want every result to be shown, then add `--cfg-options default_hooks.visualization.interval=1` to the above command. Of course, you can also modify the relevant parameters in config files. For more details of visualization, please see this guide.

If you want to save the predicted optical flow, just specify the `--show-dir`. For example, if we want to save the predicted results in `show_dirs`, then run

```
python tools/test.py configs/pwcnet/pwcnet_8x1_slong_flyingchairs_384x448.py \
    checkpoints/pwcnet_8x1_slong_flyingchairs_384x448.pth --show-dir show_dirs
```

Similarly, you can change the frequency of saving results by the above method.

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/pwcnet/pwcnet_8x1_slong_flyingchairs_384x448.py \
    checkpoints/pwcnet_8x1_slong_flyingchairs_384x448.pth
```

CHAPTER

FOUR

USEFUL TOOLS

4.1 Visualization

MMFlow 1.x provides convenient ways for monitoring training status or visualizing data and model predictions.

4.1.1 Training status Monitor

MMFlow 1.x uses TensorBoard to monitor training status.

TensorBoard Configuration

Install TensorBoard following official instructions

```
pip install future tensorboard
```

Add `TensorboardVisBackend` in `vis_backend` of `visualizer` in `configs/_base_/default_runtime.py`:

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

The configuration contains `LocalVisBackend`, which means the scalars during training will be stored locally as well.

Examining scalars in TensorBoard

Launch training experiment e.g.

```
python tools/train.py configs/pwcnet/pwcnet_8xb1_slong_flyingchairs-384x448.py --work-
↪dir work_dirs/test_visual
```

You can specify the `save_dir` in `visualizer` to modify the storage path. The default storage path is `vis_data` under your `work_dir`. For example, the `vis_data` path of a particular experiment is

```
work_dirs/test_visual/20220831_165919/vis_data
```

The scalar file in `vis_data` includes learning rate, losses and data_time etc, and also record metrics results during evaluation. You can refer to logging tutorial in mmengine to log custom data. The TensorBoard visualization results are executed with the following command:

```
tensorboard --logdir work_dirs/test_visual/20220831_165919/vis_data
```

4.1.2 Prediction Visualization

MMFlow provides `FlowVisualizationHook` that can render optical flow of ground truth and prediction. Users can modify `visualization` in `default_hooks` to invoke the hook. MMFlow configures `default_hooks` in each file under `configs/_base_/schedules`. For example, in `configs/_base_/schedules/schedules_s_long.py`, let's modify the `FlowVisualizationHook` related parameters. Set `draw` to `True` to enable the storage of network inference results. `interval` indicates the sampling interval of the predicted results, defaults to 50, and when set to 1, each inference result of the network will be saved.

```
default_hooks = dict(
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=50, log_metric_by_epoch=False),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=100000, by_epoch=False),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    visualization=dict(type='FlowVisualizationHook', draw=True, interval=1))
```

There is a way not to change the files under `configs/_base_`. For example, in `configs/pwcnet/pwcnet_8xb1_slong_flyingchairs-384x448.py` inherited from `configs/_base_/schedules/schedules_s_long.py`, just add `visualization` field in this way:

```
default_hooks = dict(
    visualization=dict(type='FlowVisualizationHook', draw=True, interval=1))
```

Additionally, if you want to keep the original file under `configs` unchanged, you can specify `--cfg-options` in commands by referring to this guide.

```
python tools/test.py \
    configs/pwcnet/pwcnet_8xb1_slong_flyingchairs-384x448.py \
    work_dirs/download/hub/checkpoints/pwcnet_8x1_slong_flyingchairs_384x448.pth \
    --work-dir work_dirs/test_visual \
    --cfg-options default_hooks.visualization.draw=True default_hooks.visualization.
↪interval=1
```

The default backend of visualization is `LocalVisBackend`, which means storing the visualization results locally. Backend related configuration is in `configs/_base_/default_runtime.py`. In order to enable TensorBoard visualization as well, modify the `visulizer` just as this configuration. Assume the `vis_data` path of a particular test is

```
work_dirs/test_visual/20220831_114424/vis_data
```

The stored results of the local visualization are kept in `vis_image` under `vis_data`, while the TensorBoard visualization results can be executed with the following command:

```
tensorboard --logdir work_dirs/test_visual/20220831_114424/vis_data
```

The visualization image consists of two parts, the ground truth on the left and the network prediction result on the right.

If you would like to know more visualization usage, you can refer to visualization tutorial in mmengie.

4.2 Analysis tools

CHAPTER

FIVE

BASIC CONCEPTS

5.1 Data Flow

5.2 Structures

5.3 Models

5.4 Datasets

5.5 Data Transforms

5.5.1 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 training on FlyingChairs.

```
train_pipeline = [
    dict(type='LoadImageFromFile', backend_args=backend_args),
    dict(type='LoadAnnotations', backend_args=backend_args),
    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='RandomFlip', prob=0.5, direction='horizontal'),
    dict(type='RandomFlip', prob=0.5, direction='vertical'),
```

(continues on next page)

(continued from previous page)

```
    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='PackFlowInputs')
]
```

For each operation, we list the related dict fields that are added/updated/removed. Before pipelines, the information we can directly obtain from the datasets are img1_path, img2_path and flow_fw_path.

Data loading

`LoadImageFromFile`

- add: img1, img2, img_shape, ori_shape

`LoadAnnotations`

- add: gt_flow_fw, gt_flow_bw(None), sparse(False)

Note: FlyingChairs doesn't provide the ground truth of backward flow, so gt_flow_bw is None. Besides, FlyingChairs' ground truth is dense, so sparse is False. For some special datasets, such as HD1K and KITTI, their ground truth is sparse, so gt_valid_fw and gt_valid_bw will be added. FlyingChairsOcc and FlyingThing3d contain the ground truth of occlusion, so gt_occ_fw and gt_occ_bw will be added for these datasets. In the pipelines below, we only consider the case of FlyingChairs.

Pre-processing

`ColorJitter`

- update: img1, img2

`RandomGamma`

- add: gamma
- update: img1, img2

`RandomFlip`

- add: flip, flip_direction
- update: img1, img2, flow_gt

`RandomAffine`

- add: global_ndc_affine_mat, relative_ndc_affine_mat
- update: img1, img2, flow_gt

`RandomCrop`

- add: crop_bbox
- update: img1, img2, flow_gt, img_shape

Formatting

`PackFlowInputs`

- add: inputs, data_sample
- remove: img1 and img2 (merged into inputs), keys specified by `data_keys` (like gt_flow_fw, merged into data_sample) keys specified by `meta_keys` (merged into the metainfo of data_sample), all other keys

5.6 Evaluation

5.7 Engine

5.8 Conventions

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

5.8.1 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.

5.8.2 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`,

```
@MODELS.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]]:
```

(continues on next page)

(continued from previous page)

```
        """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:
            Sequence[Dict[str, ndarray]]: The batch of predicted optical flow
                with the same size of images before augmentation.
        """

        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.
        """
```

(continues on next page)

(continued from previous page)

```
        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
```

CHAPTER

SIX

COMPONENT CUSTOMIZATION

6.1 Adding New Modules

MMFlow decomposes a flow estimation method `flow_estimator` into `encoder` and `decoder`. This tutorial is for how to add new components.

6.1.1 Add a new encoder

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

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

```python
from mmengine.model import BaseModule

from mmflow.registry import MODELS

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

    def __init__(self, arg1, arg2):  # arg1 and arg2 need to be specified in config
        pass

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

    # optional
    def init_weights(self):
        pass
```

2. Import the module in `mmflow/models/encoders/__init__.py`.

```python
from .my_model import MyEncoder
```

6.1.2 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 mmengine, and overwrite `forward` and `init_weights`.

```python
from mmengine.model import BaseModule

from mmflow.registry import MODELS

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

    def __init__(self, arg1, arg2):  # arg1 and arg2 need to be specified in config
        pass

    def forward(self, *args):
        pass

    # optional
    def init_weights(self):
        pass

    def loss(self, *args, batch_data_samples):
        flow_pred = self.forward(*args)
        return self.loss_by_feat(flow_pred, batch_data_samples)

    def predict(self, *args, batch_img_metas):
        flow_pred = self.forward(*args)
        flow_results = flow_pred[self.end_level]
        return self.predict_by_feat(flow_results, batch_img_metas)
```

   `batch_data_samples` contains the ground truth and `batch_img_metas` contains the information of original input images, such as original shape. `loss_by_feat` is the loss function to compute the losses between the model output and target, and you can refer to the implementation of PWCNetDecoder. `predict_by_feat` aims to restore the flow shape as the original shape of input images, and you can refer to the implementations of BaseDecoder

2. Import the module in `mmflow/models/decoders/__init__.py`

```python
from .my_decoder import MyDecoder
```

6.1.3 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. A typical encoder-decoder estimator can be written like:

```python
from .base_flow_estimator import FlowEstimator

from mmflow.registry import MODELS

```

(continues on next page)

(continued from previous page)

```
@MODELS.register_module()
class MyEstimator(FlowEstimator):

    def __init__(self, encoder: dict, decoder: dict):
        pass

    def loss(self, batch_inputs, batch_data_samples):
        pass

    def predict(self, batch_inputs, batch_data_samples):
        pass

    def _forward(self, batch_inputs, data_samples):
        pass

    def extract_feat(self, batch_inputs):
        pass
```

`loss`, `predict`, `_forward` and `extract_feat` are abstract methods of `FlowEstimator`. They can be seen as high-level APIs of the methods in `MyEncoder` and `MyDecoder`.

2. Import the module in `mmflow/models/flow_estimators/__init__.py`

```
from .my_estimator import MyEstimator
```

3. Use it in your config file.

   It's worth pointing out that `data_preprocessor` is an important parameter of `FlowEstimator` which can be used to move data to a specified device (such as a GPU) and further format the input data. In addition, image normalization, adding Gaussian noise are implemented in `data_preprocessor` as well. Therefore, `data_preprocessor` needs to be specified in the config of `MyEstimator`. You can refer to the config of PWC-Net for a typical configuration of `data_preprocessor`.

```
model = dict(
    type='MyEstimator',
    data_preprocessor=dict(
        type='FlowDataPreprocessor',
        mean=[0., 0., 0.],
        std=[255., 255., 255.]),
    encoder=dict(
        type='MyEncoder',
        arg1=xxx,
        arg2=xxx),
    decoder=dict(
        type='MyDecoder',
        arg1=xxx,
        arg2=xxx))
```

6.1.4 Add new loss

1. Create a new file `mmflow/models/losses/my_loss.py`

   Assume you want to add a new loss as `MyLoss` for flow estimation.

```
import torch.nn as nn

from mmflow.registry import MODELS

def my_loss(pred, target, *args):
    pass

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

    def __init__(self, *args):
        super(MyLoss, self).__init__()

    def forward(self, preds_dict, target, *args):
        return my_loss(preds_dict, target, *args)
```

2. Import the module in `mmflow/models/losses/__init__.py`.

```
from .my_loss import MyLoss, my_loss
```

3. Modify the `flow_loss` field in the model to use `MyLoss`

```
flow_loss=dict(type='MyLoss')
```

6.2 Adding New Datasets

6.3 Adding New Data Transforms

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

```
from mmflow.registry import TRANSFORMS

@TRANSFORMS.register_module()
class MyTransform:

    def transforms(self, results):
        results['dummy'] = True
        return results
```

2. Import the new class.

```
from .my_transform import MyTransform
```

3. Use it in config files.

```
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='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='MyTransform'),
dict(type='PackFlowInputs')]
```

6.4 Adding New Metrics

6.5 Runtime Settings Customization

In this tutorial, we will introduce some methods about how to customize optimization methods, training schedules and hooks when running your own settings for the project.

6.5.1 Customize optimization settings

Optimization related configuration is now all managed by OptimWrapper, which is a high-level API of optimizer. The OptimWrapper supports different training strategies, including auto mixed precision training, gradient accumulation and gradient clipping. `optim_wrapper` usually has three fields: `optimizer`, `paramwise_cfg`, `clip_grad`, refer to OptimWrapper for more detail. See the example below, where `Adam` is used as an optimizer and gradient clipping is added.

```
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=dict(
        type='Adam', lr=0.0001, weight_decay=0.0004, betas=(0.9, 0.999)))
    clip_grad=dict(max_norm=0.01, norm_type=2)
```

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.

```
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.

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

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/engine/optimizers/my_optimizer.py`. And then implement the new optimizer in a file, e.g., in `mmflow/engine/optimizers/my_optimizer.py`:

```
from mmflow.registry import OPTIMIZERS
from torch.optim import Optimizer


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

    def __init__(self, a, b, c):
```

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/engine/optimizers/__init__.py` to import it.

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

```
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

```
custom_imports = dict(imports=['mmflow.engine.optimizers.my_optimizer'], allow_failed_
↪imports=False)
```

The module `mmflow.engine.optimizers.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.engine.optimizers.my_optimizer.MyOptimizer` **cannot** be imported directly.

3. Specify the optimizer in the config file

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

```
optim_wrapper = dict(
    type='OptimWrapper',
    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

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

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 wrapper constructor.

```
from mmengine.optim import DefaultOptimWrapperConstructor

from mmflow.registry import OPTIM_WRAPPER_CONSTRUCTORS
from .my_optimizer import MyOptimizer


@OPTIM_WRAPPER_CONSTRUCTORS.register_module()
class MyOptimizerConstructor:

    def __init__(self,
                 optimizer_wrapper_cfg: dict,
                 paramwise_cfg: Optional[dict] = None):
        pass

    def __call__(self, model: nn.Module) -> OptimWrapper:

        return optim_wrapper
```

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

Additional settings

Tricks not implemented by the optimizer should be implemented through optimizer wrapper 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 stabilize the training process. An example is as below:

```
optim_wrapper = dict(
    _delete_=True, clip_grad=dict(max_norm=35, norm_type=2))
```

  If your config inherits the base config which already sets the `optim_wrapper`, you might need `_delete_=True` to override the unnecessary settings. See the config documentation for more details.

- **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 CosineAnnealingLR and CosineAnnealingMomentum.

```
param_scheduler = [
    # learning rate scheduler
    # During the first 8 epochs, learning rate increases from 0 to lr * 10
    # during the next 12 epochs, learning rate decreases from lr * 10 to lr * 1e-4
    dict(
        type='CosineAnnealingLR',
        T_max=8,
        eta_min=lr * 10,
        begin=0,
        end=8,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=12,
        eta_min=lr * 1e-4,
        begin=8,
        end=20,
        by_epoch=True,
        convert_to_iter_based=True),
    # momentum scheduler
    # During the first 8 epochs, momentum increases from 0 to 0.85 / 0.95
    # during the next 12 epochs, momentum increases from 0.85 / 0.95 to 1
    dict(
        type='CosineAnnealingMomentum',
        T_max=8,
        eta_min=0.85 / 0.95,
        begin=0,
        end=8,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingMomentum',
        T_max=12,
```

(continues on next page)

(continued from previous page)

```
        eta_min=1,
        begin=8,
        end=20,
        by_epoch=True,
        convert_to_iter_based=True)
]
```

6.5.2 Customize training schedules

MultiStepLR schedule implemented in MMEngine is widely used in MMFlow. We also support many other learning rate schedules here, such as `CosineAnnealingLR` and `PolyLR` schedule. Here are some examples

- Poly schedule:

```
param_scheduler = [
    dict(
        type='PolyLR',
        power=0.9,
        eta_min=1e-4,
        begin=0,
        end=8,
        by_epoch=True)]
```

- ConsineAnnealing schedule:

```
param_scheduler = [
    dict(
        type='CosineAnnealingLR',
        T_max=8,
        eta_min=lr * 1e-5,
        begin=0,
        end=8,
        by_epoch=True)]
```

6.5.3 Customize hooks

Customize self-implemented hooks

1. Implement a new hook

MMEngine provides many useful hooks, but there are some occasions when the users might need to implement a new hook. MMFlow supports customized hooks in training in v1.0. Thus the users could implement a hook directly in mmflow and use the hook by only modifying the config in training. Here we give an example of creating a new hook in mmflow and using it in training.

```
from mmengine.hooks import Hook
from mmflow.registry import HOOKS


@HOOKS.register_module()
```

(continues on next page)

(continued from previous page)

```
class MyHook(Hook):

    def __init__(self, a, b):

    def before_run(self, runner) -> None:

    def after_run(self, runner) -> None:

    def before_train(self, runner) -> None:

    def after_train(self, runner) -> None:

    def before_train_epoch(self, runner) -> None:

    def after_train_epoch(self, runner) -> None:

    def before_train_iter(self,
                          runner,
                          batch_idx: int,
                          data_batch: DATA_BATCH = None) -> None:

    def after_train_iter(self,
                         runner,
                         batch_idx: int,
                         data_batch: DATA_BATCH = None,
                         outputs: Optional[dict] = None) -> None:
```

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_train`, `after_train` , `before_train_epoch`, `after_train_epoch`, `before_train_iter`, and `after_train_iter`. There are more points where hooks can be inserted, referring to base hook class for more detail.

2. Register the new hook

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

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

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

```
from .my_hook import MyHook
```

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

```
custom_imports = dict(imports=['mmflow.engine.hooks.my_hook'], allow_failed_
↪imports=False)
```

3. Modify the config

```
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

```
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.

Modify default runtime hooks

There are some common hooks that are registered through `default_hooks`, they are

- `IterTimerHook`: A hook that logs 'data_time' for loading data and 'time' for a model train step.
- `LoggerHook`: A hook that Collect logs from different components of `Runner` and write them to terminal, JSON file, tensorboard and wandb .etc.
- `ParamSchedulerHook`: A hook to update some hyper-parameters in optimizer, e.g., learning rate and momentum.
- `CheckpointHook`: A hook that saves checkpoints periodically.
- `DistSamplerSeedHook`: A hook that sets the seed for sampler and batch_sampler.
- `FlowVisualizationHook`: A hook used to visualize predicted optical flow during validation and testing.

`IterTimerHook`, `ParamSchedulerHook` and `DistSamplerSeedHook` are simple and no need to be modified usually, so here we reveals how what we can do with `LoggerHook`, `CheckpointHook` and `FlowVisualizationHook`.

CheckpointHook

Except saving checkpoints periodically, `CheckpointHook` provides other options such as `max_keep_ckpts`, `save_optimizer` and etc. The users could set `max_keep_ckpts` to only save small number of checkpoints or decide whether to store state dict of optimizer by `save_optimizer`. More details of the arguments are here.

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

LoggerHook

The LoggerHook enables to set intervals. And the detail usages can be found in the docstring.

```
default_hooks = dict(logger=dict(type='LoggerHook', interval=50))
```

FlowVisualizationHook

FlowVisualizationHook uses FlowLocalVisualizer to visualize prediction results, and FlowLocalVisualizer current supports different backends, e.g., TensorboardVisBackend (see docstring for more detail). The users could add multi backbends to do visualization, as follows.

```
default_hooks = dict(
    visualization=dict(type='FlowVisualizationHook', draw=True))

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

CHAPTER

SEVEN

MIGRATION FROM MMFLOW 0.X

CHAPTER

EIGHT

MMFLOW.APIS

CHAPTER

NINE

MMFLOW.DATASETS

9.1 datasets

9.2 transforms

9.3 samplers

CHAPTER

TEN

MMFLOW.ENGINE

10.1 hooks

10.2 loops

10.3 schedulers

CHAPTER

ELEVEN

MMFLOW.EVALUATION

11.1 metrics

CHAPTER

TWELVE

MMFLOW.MODELS

12.1 encoders

12.2 decoders

12.3 flow_estimators

12.4 losses

CHAPTER

THIRTEEN

MMFLOW.VISUALIZATION

CHAPTER

FOURTEEN

MMFLOW.UTILS

CHAPTER

FIFTEEN

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)
  - PWC-Net: CNNs for Optical Flow Using Pyramid, Warping, and Cost Volume (6 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

SIXTEEN

CHANGELOG OF V1.X

16.1 v1.0.0rc0 (31/8/2022)

We are excited to announce the release of MMFlow 1.0.0rc0. MMFlow 1.0.0rc0 is a part of the OpenMMLab 2.0 projects. Built upon the new training engine, MMFlow 1.x unifies the interfaces of dataset, models, evaluation, and visualization with faster training and testing speed.

16.1.1 Highlights

1. **New engines** MMFlow 1.x is based on MMEngine, which provides a general and powerful runner that allows more flexible customizations and significantly simplifies the entrypoints of high-level interfaces.
2. **Unified interfaces** As a part of the OpenMMLab 2.0 projects, MMFlow 1.x unifies and refactors the interfaces and internal logics of training, testing, datasets, models, evaluation, and visualization. All the OpenMMLab 2.0 projects share the same design in those interfaces and logics to allow the emergence of multi-task/modality algorithms.
3. **Faster speed** We optimize the training and inference speed for common models.
4. **More documentation and tutorials** We add a bunch of documentation and tutorials to help users get started more smoothly. Read it here.

16.1.2 Breaking Changes

We briefly list the major breaking changes here. We will update the *migration guide* to provide complete details and migration instructions.

Training and testing

- MMFlow 1.x runs on PyTorch>=1.6. We have deprecated the support of PyTorch 1.5 to embrace the mixed precision training and other new features since PyTorch 1.6. Some models can still run on PyTorch 1.5, but the full functionality of MMFlow 1.x is not guaranteed.
- MMFlow 1.x uses Runner in MMEngine rather than that in MMCV. The new Runner implements and unifies the building logic of dataset, model, evaluation, and visualization. Therefore, MMFlow 1.x no longer maintains the building logics of those modules in `mmflow.train.apis` and `tools/train.py`. Those code have been migrated into MMEngine. Please refer to the migration guide of Runner in MMEngine for more details.
- The Runner in MMEngine also supports testing and validation. The testing scripts are also simplified, which has similar logic as that in training scripts to build the runner.

- The execution points of hooks in the new Runner have been enriched to allow more flexible customization. Please refer to the migration guide of Hook in MMEngine for more details.
- Learning rate and momentum scheduling has been migrated from `Hook` to `Parameter Scheduler` in MMEngine. Please refer to the migration guide of Parameter Scheduler in MMEngine for more details.

Configs

- The Runner in MMEngine uses a different config structures to ease the understanding of the components in runner. Users can read the *config example of mmflow* or refer to the migration guide in MMEngine for migration details.
- The file names of configs and models are also refactored to follow the new rules unified across OpenMMLab 2.0 projects. Please refer to the *user guides of config* for more details.

Components

- Dataset
- Data Transforms
- Model
- Evaluation
- Visualization

16.1.3 Improvements

- The training speed of those models with some common training strategies are improved, including those with synchronized batch normalization and mixed precision training.
- Support mixed precision training of all the models. However, some models may got Nan results due to some numerical issues. We will update the documentation and list their results (accuracy of failure) of mixed precision training.

16.1.4 Ongoing changes

1. Inference interfaces: a unified inference interfaces will be supported in the future to ease the use of released models.
2. Interfaces of useful tools that can be used in notebook: more useful tools that implemented in the `tools` directory will have their python interfaces so that they can be used through notebook and in downstream libraries.
3. Documentation: we will add more design docs, tutorials, and migration guidance so that the community can deep dive into our new design, participate the future development, and smoothly migrate downstream libraries to MMFlow 1.x.

CHAPTER

SEVENTEEN

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.

17.1 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

EIGHTEEN

CHAPTER

NINETEEN

ENGLISH

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search