MMTracking

Release 0.13.0

MMTracking Authors

May 17, 2022

GET STARTED

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

CHAPTER

ONE

PREREQUISITES

- Linux | macOS | Windows
- Python 3.6+
- PyTorch 1.3+
- CUDA 9.2+ (If you build PyTorch from source, CUDA 9.0 is also compatible)
- GCC 5+
- MMCV
- MMDetection

The compatible MMTracking, MMCV, and MMDetection versions are as below. Please install the correct version to avoid installation issues.

CHAPTER

TWO

INSTALLATION

2.1 Detailed Instructions

1. Create a conda virtual environment and activate it.

```
conda create -n open-mmlab python=3.7 -y
conda activate open-mmlab
```

2. Install PyTorch and torchvision following the official instructions, e.g.,

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

Note: Make sure that your compilation CUDA version and runtime CUDA version match. You can check the supported CUDA version for precompiled packages on the PyTorch website.

`E.g.1` If you have CUDA 10.1 installed under `/usr/local/cuda` and would like to install PyTorch 1.5, you need to install the prebuilt PyTorch with CUDA 10.1.

```
conda install pytorch==1.5 cudatoolkit=10.1 torchvision -c pytorch
```

`E.g. 2` If you have CUDA 9.2 installed under `/usr/local/cuda` and would like to install PyTorch 1.3.1., you need to install the prebuilt PyTorch with CUDA 9.2.

```
conda install pytorch=1.3.1 cudatoolkit=9.2 torchvision=0.4.2 -c pytorch
```

If you build PyTorch from source instead of installing the prebuilt package, you can use more CUDA versions such as 9.0.

3. Install extra dependencies for VOT evaluation (optional)

If you need to evaluate on VOT Challenge, please install the vot-toolkit before the installation of mmcv and mmdetection to avoid possible numpy version requirement conflict among some dependencies.

```
pip install git+https://github.com/votchallenge/toolkit.git
```

4. Install mmcv-full, we recommend you to install the pre-build package as below.

```
# pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/{cu_version}/
↪{torch_version}/index.html
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu102/torch1.10.0/
↪index.html
```

mmcv-full is only compiled on PyTorch 1.x.0 because the compatibility usually holds between 1.x.0 and 1.x.1. If your PyTorch version is 1.x.1, you can install mmcv-full compiled with PyTorch 1.x.0 and it usually works well.

```
# We can ignore the micro version of PyTorch
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu102/torch1.10/
↪index.html
```

See here for different versions of MMCV compatible to different PyTorch and CUDA versions. Optionally you can choose to compile mmcv from source by the following command

```
git clone https://github.com/open-mmlab/mmcv.git
cd mmcv
MMCV_WITH_OPS=1 pip install -e .  # package mmcv-full will be installed after this␣
↪step
cd ..
```

5. Install MMDetection

```
pip install mmdet
```

Optionally, you can also build MMDetection from source in case you want to modify the code:

```
git clone https://github.com/open-mmlab/mmdetection.git
cd mmdetection
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

6. Clone the MMTracking repository.

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

7. Install build requirements and then install MMTracking.

```
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

8. Install extra dependencies

- For MOTChallenge evaluation:

```
pip install git+https://github.com/JonathonLuiten/TrackEval.git
```

- For LVIS evaluation:

```
pip install git+https://github.com/lvis-dataset/lvis-api.git
```

- For TAO evaluation:

```
pip install git+https://github.com/TAO-Dataset/tao.git
```

Note:

a. Following the above instructions, MMTracking is installed on `dev` mode , any local modifications made to the code will take effect without the need to reinstall it.

b. If you would like to use `opencv-python-headless` instead of `opencv-python`, you can install it before installing MMCV.

2.2 A from-scratch setup script

Assuming that you already have CUDA 10.1 installed, here is a full script for setting up MMTracking with conda.

```
conda create -n open-mmlab python=3.7 -y
conda activate open-mmlab

conda install pytorch==1.6.0 torchvision==0.7.0 cudatoolkit=10.1 -c pytorch -y

pip install git+https://github.com/votchallenge/toolkit.git (optional)
# install the latest mmcv
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu101/torch1.6.0/index.
↪html

# install mmdetection
pip install mmdet

# install mmtracking
git clone https://github.com/open-mmlab/mmtracking.git
cd mmtracking
pip install -r requirements/build.txt
pip install -v -e .
pip install git+https://github.com/JonathonLuiten/TrackEval.git
pip install git+https://github.com/lvis-dataset/lvis-api.git
pip install git+https://github.com/TAO-Dataset/tao.git
```

2.3 Developing with multiple MMTracking versions

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

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

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

CHAPTER

THREE

VERIFICATION

To verify whether MMTracking and the required environment are installed correctly, we can run MOT, VID, SOT demo script.

For example, run MOT demo and you will see a output video named `mot.mp4`:

```
python demo/demo_mot_vis.py configs/mot/deepsort/sort_faster-rcnn_fpn_4e_mot17-private.
↪py --input demo/demo.mp4 --output mot.mp4
```

CHAPTER

FOUR

MODEL ZOO STATISTICS

- Number of papers: 11
  - ABSTRACT: 11
- Number of checkpoints: 41
  - [ABSTRACT] ByteTrack: Multi-Object Tracking by Associating Every Detection Box (2 ckpts)
  - [ABSTRACT] Simple online and realtime tracking with a deep association metric (2 ckpts)
  - [ABSTRACT] Quasi-Dense Similarity Learning for Multiple Object Tracking (2 ckpts)
  - [ABSTRACT] Tracking without Bells and Whistles (7 ckpts)
  - [ABSTRACT] Siamrpn++: Evolution of Siamese Visual Tracking With Very Deep Networks (5 ckpts)
  - [ABSTRACT] Learning Spatio-Temporal Transformer for Visual Tracking (4 ckpts)
  - [ABSTRACT] Deep Feature Flow for Video Recognition (3 ckpts)
  - [ABSTRACT] Flow-guided Feature Aggregation for Video Object Detection (3 ckpts)
  - [ABSTRACT] Sequence Level Semantics Aggregation for Video Object Detection (4 ckpts)
  - [ABSTRACT] Temporal RoI Align for Video Object Recognition (3 ckpts)
  - [ABSTRACT] Video Instance Segmentation (6 ckpts)

CHAPTER

FIVE

BENCHMARK AND MODEL ZOO

5.1 Common settings

- We use distributed training.
- All pytorch-style pretrained backbones on ImageNet are from PyTorch model zoo.
- For fair comparison with other codebases, we report the GPU memory as the maximum value of `torch.cuda.max_memory_allocated()` for all 8 GPUs. Note that this value is usually less than what `nvidia-smi` shows.
- We report the inference time as the total time of network forwarding and post-processing, excluding the data loading time. Results are obtained with the script `tools/analysis/benchmark.py` which computes the average time on 2000 images.
- Speed benchmark environments

  HardWare

  - 8 NVIDIA Tesla V100 (32G) GPUs
  - Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz

  Software environment

  - Python 3.7
  - PyTorch 1.5
  - CUDA 10.1
  - CUDNN 7.6.03
  - NCCL 2.4.08

5.2 Baselines of video object detection

5.2.1 DFF (CVPR 2017)

Please refer to DFF for details.

5.2.2 FGFA (ICCV 2017)

Please refer to FGFA for details.

5.2.3 SELSA (ICCV 2019)

Please refer to SELSA for details.

5.2.4 Temporal RoI Align (AAAI 2021)

Please refer to Temporal RoI Align for details.

5.3 Baselines of multiple object tracking

5.3.1 SORT/DeepSORT (ICIP 2016/2017)

Please refer to SORT/DeepSORT for details.

5.3.2 Tracktor (ICCV 2019)

Please refer to Tracktor for details.

5.3.3 QDTrack (CVPR 2021)

Please refer to QDTrack for details.

5.3.4 ByteTrack (arXiv 2021)

Please refer to ByteTrack for details.

5.4 Baselines of single object tracking

5.4.1 SiameseRPN++ (CVPR 2019)

Please refer to SiameseRPN++ for details.

5.4.2 STARK (ICCV 2021)

Please refer to STARK for details.

5.5 Baselines of video instance segmentation

5.5.1 MaskTrack R-CNN (ICCV 2019)

Please refer to MaskTrack R-CNN for details.

CHAPTER

SIX

DATASET PREPARATION

This page provides the instructions for dataset preparation on existing benchmarks, include

- Video Object Detection
  - ILSVRC
- Multiple Object Tracking
  - MOT Challenge
  - CrowdHuman
  - LVIS
  - TAO
  - DanceTrack
- Single Object Tracking
  - LaSOT
  - UAV123
  - TrackingNet
  - OTB100
  - GOT10k
  - VOT2018
- Video Instance Segmentation
  - YouTube-VIS

6.1 1. Download Datasets

Please download the datasets from the official websites. It is recommended to symlink the root of the datasets to `$MMTRACKING/data`.

6.1.1 1.1 Video Object Detection

- For the training and testing of video object detection task, only ILSVRC dataset is needed.
- The `Lists` under `ILSVRC` contains the txt files from here.

6.1.2 1.2 Multiple Object Tracking

- For the training and testing of multi object tracking task, one of the MOT Challenge datasets (e.g. MOT17, TAO and DanceTrack) are needed, CrowdHuman and LVIS can be served as comlementary dataset.
- The `annotations` under `tao` contains the official annotations from here.
- The `annotations` under `lvis` contains the official annotations of lvis-v0.5 which can be downloaded according to here. The synset mapping file `coco_to_lvis_synset.json` used in `./tools/convert_datasets/tao/merge_coco_with_lvis.py` script can be found here.

6.1.3 1.3 Single Object Tracking

- For the training and testing of single object tracking task, the MSCOCO, ILSVRC, LaSOT, UAV123, TrackingNet, OTB100, GOT10k and VOT2018 datasets are needed.
- For OTB100 dataset, you don't need to download the dataset from the official website manually, since we provide a script to download it.

```
# download OTB100 dataset by web crawling
python ./tools/convert_datasets/otb100/download_otb100.py -o ./data/otb100/zips -p 8
```

- For VOT2018, we use the official downloading script.

```
# download VOT2018 dataset by web crawling
python ./tools/convert_datasets/vot/download_vot.py --dataset vot2018 --save_path ./data/
↪vot2018/data
```

6.1.4 1.4 Video Instance Segmentation

- For the training and testing of video instance segmetatioon task, only one of YouTube-VIS datasets (e.g. YouTube-VIS 2019) is needed.

6.1.5 1.5 Data Structure

If your folder structure is different from the following, you may need to change the corresponding paths in config files.

```
mmtracking
├── mmtrack
├── tools
├── configs
├── data
│   ├── coco
│   │   ├── train2017
│   │   ├── val2017
│   │   ├── test2017
```

(continues on next page)

(continued from previous page)

```
|   |   ├── annotations
|   |
|   ├── ILSVRC
|   |   ├── Data
|   |   |   ├── DET
|   |   |   |   ├── train
|   |   |   |   ├── val
|   |   |   |   ├── test
|   |   |   ├── VID
|   |   |   |   ├── train
|   |   |   |   ├── val
|   |   |   |   ├── test
|   |   ├── Annotations
|   |   |   ├── DET
|   |   |   |   ├── train
|   |   |   |   ├── val
|   |   |   ├── VID
|   |   |   |   ├── train
|   |   |   |   ├── val
|   |   ├── Lists
|   |
|   ├── MOT15/MOT16/MOT17/MOT20
|   |   ├── train
|   |   ├── test
|   |
|   ├── DanceTrack
|   |   ├── train
|   |   ├── val
|   |   ├── test
|   |
|   ├── crowdhuman
|   |   ├── annotation_train.odgt
|   |   ├── annotation_val.odgt
|   |   ├── train
|   |   |   ├── Images
|   |   |   ├── CrowdHuman_train01.zip
|   |   |   ├── CrowdHuman_train02.zip
|   |   |   ├── CrowdHuman_train03.zip
|   |   ├── val
|   |   |   ├── Images
|   |   |   ├── CrowdHuman_val.zip
|   |
|   ├── lvis
|   |   ├── train (the same as coco/train2017)
|   |   ├── val (the same as coco/val2017)
|   |   ├── test (the same as coco/test2017)
|   |   ├── annotations
|   |   |   ├── coco_to_lvis_synset.json
|   |   |   ├── lvis_v0.5_train.json
|   |   |   ├── lvis_v0.5_val.json
|   |   |   ├── lvis_v1_train.json
|   |   |   ├── lvis_v1_val.json
```

(continues on next page)

(continued from previous page)

```
│   │   │   ├── lvis_v1_image_info_test_challenge.json
│   │   │   ├── lvis_v1_image_info_test_dev.json
│   │
│   ├── tao
│   │   ├── annotations
│   │   │   ├── test_without_annotations.json
│   │   │   ├── train.json
│   │   │   ├── validation.json
│   │   │   ├── ......
│   │   ├── test
│   │   │   ├── ArgoVerse
│   │   │   ├── AVA
│   │   │   ├── BDD
│   │   │   ├── Charades
│   │   │   ├── HACS
│   │   │   ├── LaSOT
│   │   │   ├── YFCC100M
│   │   ├── train
│   │   ├── val
│   │
│   ├── lasot
│   │   ├── LaSOTBenchmark
│   │   │   ├── airplane
│   │   │   │   ├── airplane-1
│   │   │   │   ├── airplane-2
│   │   │   │   ├── ......
│   │   │   ├── ......
│   │
│   ├── UAV123
│   │   ├── data_seq
│   │   │   ├── UAV123
│   │   │   │   ├── bike1
│   │   │   │   ├── boat1
│   │   │   │   ├── ......
│   │   ├── anno
│   │   │   ├── UAV123
│   │
│   ├── trackingnet
│   │   ├── TEST.zip
│   │   ├── TRAIN_0.zip
│   │   ├── ......
│   │   ├── TRAIN_11.zip
│   │
│   ├── otb100
│   │   ├── zips
│   │   │   ├── Basketball.zip
│   │   │   ├── Biker.zip
│   │   │   ├──
│   │
│   ├── got10k
│   │   ├── full_data
│   │   │   ├── train_data
```

(continues on next page)

(continued from previous page)

```
|   |   |   |   |   ├── GOT-10k_Train_split_01.zip
|   |   |   |   |   ├── ......
|   |   |   |   |   ├── GOT-10k_Train_split_19.zip
|   |   |   |   |   ├── list.txt
|   |   |   |   ├── test_data.zip
|   |   |   |   ├── val_data.zip
|   |   |
|   |   ├── vot2018
|   |   |   ├── data
|   |   |   |   ├── ants1
|   |   |   |   |   ├──color
|   |   |
|   |   ├── youtube_vis_2019
|   |   |   ├── train
|   |   |   |   ├── JPEGImages
|   |   |   |   ├── ......
|   |   |   ├── valid
|   |   |   |   ├── JPEGImages
|   |   |   |   ├── ......
|   |   |   ├── test
|   |   |   |   ├── JPEGImages
|   |   |   |   ├── ......
|   |   |   ├── train.json (the official annotation files)
|   |   |   ├── valid.json (the official annotation files)
|   |   |   ├── test.json (the official annotation files)
|   |   |
|   |   ├── youtube_vis_2021
|   |   |   ├── train
|   |   |   |   ├── JPEGImages
|   |   |   |   ├── instances.json (the official annotation files)
|   |   |   |   ├── ......
|   |   |   ├── valid
|   |   |   |   ├── JPEGImages
|   |   |   |   ├── instances.json (the official annotation files)
|   |   |   |   ├── ......
|   |   |   ├── test
|   |   |   |   ├── JPEGImages
|   |   |   |   ├── instances.json (the official annotation files)
|   |   |   |   ├── ......
```

6.2 2. Convert Annotations

We use CocoVID to maintain all datasets in this codebase. In this case, you need to convert the official annotations to this style. We provide scripts and the usages are as following:

```
# ImageNet DET
python ./tools/convert_datasets/ilsvrc/imagenet2coco_det.py -i ./data/ILSVRC -o ./data/
↪ILSVRC/annotations

# ImageNet VID
```

(continues on next page)

(continued from previous page)

```
python ./tools/convert_datasets/ilsvrc/imagenet2coco_vid.py -i ./data/ILSVRC -o ./data/
↪ILSVRC/annotations

# MOT17
# The processing of other MOT Challenge dataset is the same as MOT17
python ./tools/convert_datasets/mot/mot2coco.py -i ./data/MOT17/ -o ./data/MOT17/
↪annotations --split-train --convert-det
python ./tools/convert_datasets/mot/mot2reid.py -i ./data/MOT17/ -o ./data/MOT17/reid --
↪val-split 0.2 --vis-threshold 0.3

# DanceTrack
python ./tools/convert_datasets/dancetrack/dancetrack2coco.py -i ./data/DanceTrack ./
↪data/DanceTrack/annotations

# CrowdHuman
python ./tools/convert_datasets/mot/crowdhuman2coco.py -i ./data/crowdhuman -o ./data/
↪crowdhuman/annotations

# LVIS
# Merge annotations from LVIS and COCO for training QDTrack
python ./tools/convert_datasets/tao/merge_coco_with_lvis.py --lvis ./data/lvis/
↪annotations/lvis_v0.5_train.json --coco ./data/coco/annotations/instances_train2017.
↪json --mapping ./data/lvis/annotations/coco_to_lvis_synset.json --output-json ./data/
↪lvis/annotations/lvisv0.5+coco_train.json

# TAO
# Generate filtered json file for QDTrack
python ./tools/convert_datasets/tao/tao2coco.py -i ./data/tao/annotations --filter-
↪classes

# LaSOT
python ./tools/convert_datasets/lasot/gen_lasot_infos.py -i ./data/lasot/LaSOTBenchmark -
↪o ./data/lasot/annotations

# UAV123
# download annotations
# due to the annotations of all videos in UAV123 are inconsistent, we just download the␣
↪information file generated in advance.
wget https://download.openmmlab.com/mmtracking/data/uav123_infos.txt -P data/uav123/
↪annotations

# TrackingNet
# unzip files in 'data/trackingnet/*.zip'
bash ./tools/convert_datasets/trackingnet/unzip_trackingnet.sh ./data/trackingnet
# generate annotations
python ./tools/convert_datasets/trackingnet/gen_trackingnet_infos.py -i ./data/
↪trackingnet -o ./data/trackingnet/annotations

# OTB100
# unzip files in 'data/otb100/zips/*.zip'
bash ./tools/convert_datasets/otb100/unzip_otb100.sh ./data/otb100
# download annotations
```

(continues on next page)

(continued from previous page)

```
# due to the annotations of all videos in OTB100 are inconsistent, we just need to␣
↪download the information file generated in advance.
wget https://download.openmmlab.com/mmtracking/data/otb100_infos.txt -P data/otb100/
↪annotations

# GOT10k
# unzip 'data/got10k/full_data/test_data.zip', 'data/got10k/full_data/val_data.zip' and␣
↪files in 'data/got10k/full_data/train_data/*.zip'
bash ./tools/convert_datasets/got10k/unzip_got10k.sh ./data/got10k
# generate annotations
python ./tools/convert_datasets/got10k/gen_got10k_infos.py -i ./data/got10k -o ./data/
↪got10k/annotations

# VOT2018
python ./tools/convert_datasets/vot/gen_vot_infos.py -i ./data/vot2018 -o ./data/vot2018/
↪annotations --dataset_type vot2018

# YouTube-VIS 2019
python ./tools/convert_datasets/youtubevis/youtubevis2coco.py -i ./data/youtube_vis_2019␣
↪-o ./data/youtube_vis_2019/annotations --version 2019

# YouTube-VIS 2021
python ./tools/convert_datasets/youtubevis/youtubevis2coco.py -i ./data/youtube_vis_2021␣
↪-o ./data/youtube_vis_2021/annotations --version 2021
```

The folder structure will be as following after your run these scripts:

```
mmtracking
├── mmtrack
├── tools
├── configs
├── data
    ├── coco
        ├── train2017
        ├── val2017
        ├── test2017
        ├── annotations

    ├── ILSVRC
        ├── Data
            ├── DET
                ├── train
                ├── val
                ├── test
            ├── VID
                ├── train
                ├── val
                ├── test
        ├── Annotations (the official annotation files)
            ├── DET
                ├── train
                ├── val
```

(continues on next page)

(continued from previous page)

```
│   │   │   ├── VID
│   │   │   │   ├── train
│   │   │   │   ├── val
│   │   ├── Lists
│   │   ├── annotations (the converted annotation files)
│   │
│   ├── MOT15/MOT16/MOT17/MOT20
│   │   ├── train
│   │   ├── test
│   │   ├── annotations
│   │   ├── reid
│   │   │   ├── imgs
│   │   │   ├── meta
│   │
│   ├── DanceTrack
│   │   ├── train
│   │   ├── val
│   │   ├── test
│   │   ├── annotations
│   │
│   ├── crowdhuman
│   │   ├── annotation_train.odgt
│   │   ├── annotation_val.odgt
│   │   ├── train
│   │   │   ├── Images
│   │   │   ├── CrowdHuman_train01.zip
│   │   │   ├── CrowdHuman_train02.zip
│   │   │   ├── CrowdHuman_train03.zip
│   │   ├── val
│   │   │   ├── Images
│   │   │   ├── CrowdHuman_val.zip
│   │   ├── annotations
│   │   │   ├── crowdhuman_train.json
│   │   │   ├── crowdhuman_val.json
│   │
│   ├── lvis
│   │   ├── train (the same as coco/train2017)
│   │   ├── val (the same as coco/val2017)
│   │   ├── test (the same as coco/test2017)
│   │   ├── annotations
│   │   │   ├── coco_to_lvis_synset.json
│   │   │   ├── lvisv0.5+coco_train.json
│   │   │   ├── lvis_v0.5_train.json
│   │   │   ├── lvis_v0.5_val.json
│   │   │   ├── lvis_v1_train.json
│   │   │   ├── lvis_v1_val.json
│   │   │   ├── lvis_v1_image_info_test_challenge.json
│   │   │   ├── lvis_v1_image_info_test_dev.json
│   │
│   ├── tao
│   │   ├── annotations
│   │   │   ├── test_482_classes.json
```

(continues on next page)

(continued from previous page)

```
│   │   │   ├── test_without_annotations.json
│   │   │   ├── train.json
│   │   │   ├── train_482_classes.json
│   │   │   ├── validation.json
│   │   │   ├── validation_482_classes.json
│   │   │   ├── ......
│   │   ├── test
│   │   │   ├── ArgoVerse
│   │   │   ├── AVA
│   │   │   ├── BDD
│   │   │   ├── Charades
│   │   │   ├── HACS
│   │   │   ├── LaSOT
│   │   │   ├── YFCC100M
│   │   ├── train
│   │   ├── val
│   │
│   ├── lasot
│   │   ├── LaSOTBenchmark
│   │   │   ├── airplane
│   │   │   │   ├── airplane-1
│   │   │   │   ├── airplane-2
│   │   │   │   ├── ......
│   │   │   ├── ......
│   │   ├── annotations
│   │
│   ├── UAV123
│   │   ├── data_seq
│   │   │   ├── UAV123
│   │   │   │   ├── bike1
│   │   │   │   ├── boat1
│   │   │   │   ├── ......
│   │   ├── anno (the official annotation files)
│   │   │   ├── UAV123
│   │   ├── annotations (the converted annotation file)
│   │
│   ├── trackingnet
│   │   ├── TEST
│   │   │   ├── anno (the official annotation files)
│   │   │   ├── zips
│   │   │   ├── frames (the unzipped folders)
│   │   │   │   ├── 0-6LB4FqxoE_0
│   │   │   │   ├── 07Ysk1C0ZX0_0
│   │   │   │   ├── ......
│   │   ├── TRAIN_0
│   │   │   ├── anno (the official annotation files)
│   │   │   ├── zips
│   │   │   ├── frames (the unzipped folders)
│   │   │   │   ├── -3TIfnTSM6c_2
│   │   │   │   ├── a1qoB1eERn0_0
│   │   │   │   ├── ......
│   │   ├── ......
```

(continues on next page)

(continued from previous page)

```
|   |   ├── TRAIN_11
|   |   ├── annotations (the converted annotation file)
|   |
|   ├── otb100
|   |   ├── zips
|   |   |   ├── Basketball.zip
|   |   |   ├── Biker.zip
|   |   |   ├── ......
|   |   ├── annotations
|   |   ├── data
|   |   |   ├── Basketball
|   |   |   |   ├── img
|   |   |   ├── ......
|   |
|   ├── got10k
|   |   ├── full_data
|   |   |   ├── train_data
|   |   |   |   ├── GOT-10k_Train_split_01.zip
|   |   |   |   ├── ......
|   |   |   |   ├── GOT-10k_Train_split_19.zip
|   |   |   |   ├── list.txt
|   |   |   ├── test_data.zip
|   |   |   ├── val_data.zip
|   |   ├── train
|   |   |   ├── GOT-10k_Train_000001
|   |   |   |   ├── ......
|   |   |   ├── GOT-10k_Train_009335
|   |   |   ├── list.txt
|   |   ├── test
|   |   |   ├── GOT-10k_Test_000001
|   |   |   |   ├── ......
|   |   |   ├── GOT-10k_Test_000180
|   |   |   ├── list.txt
|   |   ├── val
|   |   |   ├── GOT-10k_Val_000001
|   |   |   |   ├── ......
|   |   |   ├── GOT-10k_Val_000180
|   |   |   ├── list.txt
|   |   ├── annotations
|   |
|   ├── vot2018
|   |   ├── data
|   |   |   ├── ants1
|   |   |   |   ├──color
|   |   ├── annotations
|   |   |   ├── ......
|   |
|   ├── youtube_vis_2019
|   |   ├── train
|   |   |   ├── JPEGImages
|   |   |   ├── ......
|   |   ├── valid
```

(continues on next page)

(continued from previous page)

```
|   |   |   |   |── JPEGImages
|   |   |   |   |── ......
|   |   |   |── test
|   |   |   |   |── JPEGImages
|   |   |   |   |── ......
|   |   |   |── train.json (the official annotation files)
|   |   |   |── valid.json (the official annotation files)
|   |   |   |── test.json (the official annotation files)
|   |   |   |── annotations (the converted annotation file)
|   |   |
|   |   |── youtube_vis_2021
|   |   |   |── train
|   |   |   |   |── JPEGImages
|   |   |   |   |── instances.json (the official annotation files)
|   |   |   |   |── ......
|   |   |   |── valid
|   |   |   |   |── JPEGImages
|   |   |   |   |── instances.json (the official annotation files)
|   |   |   |   |── ......
|   |   |   |── test
|   |   |   |   |── JPEGImages
|   |   |   |   |── instances.json (the official annotation files)
|   |   |   |   |── ......
|   |   |   |── annotations (the converted annotation file)
```

6.2.1 The folder of annotations in ILSVRC

There are 3 JSON files in `data/ILSVRC/annotations`:

`imagenet_det_30plus1cls.json`: JSON file containing the annotations information of the training set in ImageNet DET dataset. The `30` in `30plus1cls` denotes the overlapped 30 categories in ImageNet VID dataset, and the `1cls` means we take the other 170 categories in ImageNet DET dataset as a category, named as `other_categeries`.

`imagenet_vid_train.json`: JSON file containing the annotations information of the training set in ImageNet VID dataset.

`imagenet_vid_val.json`: JSON file containing the annotations information of the validation set in ImageNet VID dataset.

6.2.2 The folder of annotations and reid in MOT15/MOT16/MOT17/MOT20

We take MOT17 dataset as examples, the other datasets share similar structure.

There are 8 JSON files in `data/MOT17/annotations`:

`train_cocoformat.json`: JSON file containing the annotations information of the training set in MOT17 dataset.

`train_detections.pkl`: Pickle file containing the public detections of the training set in MOT17 dataset.

`test_cocoformat.json`: JSON file containing the annotations information of the testing set in MOT17 dataset.

`test_detections.pkl`: Pickle file containing the public detections of the testing set in MOT17 dataset.

`half-train_cocoformat.json`, `half-train_detections.pkl`, `half-val_cocoformat.json`and `half-val_detections.pkl` share similar meaning with `train_cocoformat.json` and `train_detections.pkl`.

The half means we split each video in the training set into half. The first half videos are denoted as half-train set, and the second half videos are denoted ashalf-val set.

The structure of data/MOT17/reid is as follows:

```
reid
├── imgs
│   ├── MOT17-02-FRCNN_000002
│   │   ├── 000000.jpg
│   │   ├── 000001.jpg
│   │   ├── ...
│   ├── MOT17-02-FRCNN_000003
│   │   ├── 000000.jpg
│   │   ├── 000001.jpg
│   │   ├── ...
├── meta
│   ├── train_80.txt
│   ├── val_20.txt
```

The 80 in train_80.txt means the proportion of the training dataset to the whole ReID dataset is 80%. While the proportion of the validation dataset is 20%.

For training, we provide a annotation list train_80.txt. Each line of the list contains a filename and its corresponding ground-truth labels. The format is as follows:

```
MOT17-05-FRCNN_000110/000018.jpg 0
MOT17-13-FRCNN_000146/000014.jpg 1
MOT17-05-FRCNN_000088/000004.jpg 2
MOT17-02-FRCNN_000009/000081.jpg 3
```

MOT17-05-FRCNN_000110 denotes the 110-th person in MOT17-05-FRCNN video.

For validation, The annotation list val_20.txt remains the same as format above.

Images in reid/imgs are cropped from raw images in MOT17/train by the corresponding gt.txt. The value of ground-truth labels should fall in range [0, num_classes - 1].

6.2.3 The folder of annotations in crowdhuman

There are 2 JSON files in data/crowdhuman/annotations:

crowdhuman_train.json: JSON file containing the annotations information of the training set in CrowdHuman dataset. crowdhuman_val.json: JSON file containing the annotations information of the validation set in CrowdHuman dataset.

6.2.4 The folder of annotations in lvis

There are 8 JSON files in data/lvis/annotations

coco_to_lvis_synset.json: JSON file containing the mapping relationship between COCO and LVIS categories.

lvisv0.5+coco_train.json: JSON file containing the merged annotations.

lvis_v0.5_train.json: JSON file containing the annotations information of the training set in lvisv0.5.

lvis_v0.5_val.json: JSON file containing the annotations information of the validation set in lvisv0.5.

lvis_v1_train.json: JSON file containing the annotations information of the training set in lvisv1.

`lvis_v1_val.json`: JSON file containing the annotations information of the validation set in lvisv1.

`lvis_v1_image_info_test_challenge.json`: JSON file containing the annotations information of the testing set in lvisv1 available for year-round evaluation.

`lvis_v1_image_info_test_dev.json`: JSON file containing the annotations information of the testing set in lvisv1 available only once a year for LVIS Challenge.

6.2.5 The folder of annotations in tao

There are 9 JSON files in `data/tao/annotations`:

`test_categories.json`: JSON file containing a list of categories which will be evaluated on the TAO test set.

`test_without_annotations.json`: JSON for test videos. The 'images' and 'videos' fields contain the images and videos that will be evaluated on the test set.

`test_482_classes.json`: JSON file containing the converted results for test set.

`train.json`: JSON file containing annotations for LVIS categories in TAO train.

`train_482_classes.json`: JSON file containing the converted results for train set.

`train_with_freeform.json`: JSON file containing annotations for all categories in TAO train.

`validation.json`: JSON file containing annotations for LVIS categories in TAO train.

`validation_482_classes.json`: JSON file containing the converted results for validation set.

`validation_with_freeform.json`: JSON file containing annotations for all categories in TAO validation.

6.2.6 The folder of annotations in lasot

There are 2 JSON files in `data/lasot/annotations`:

`lasot_train.json`: JSON file containing the annotations information of the training set in LaSOT dataset.
`lasot_test.json`: JSON file containing the annotations information of the testing set in LaSOT dataset.

There are 2 TEXT files in `data/lasot/annotations`:

`lasot_train_infos.txt`: TEXT file containing the annotations information of the training set in LaSOT dataset.
`lasot_test_infos.txt`: TEXT file containing the annotations information of the testing set in LaSOT dataset.

6.2.7 The folder of annotations in UAV123

There are only 1 JSON files in `data/UAV123/annotations`:

`uav123.json`: JSON file containing the annotations information of the UAV123 dataset.

There are only 1 TEXT files in `data/UAV123/annotations`:

`uav123_infos.txt`: TEXT file containing the information of the UAV123 dataset.

6.2.8 The folder of frames and annotations in TrackingNet

There are 511 video directories of TrackingNet testset in `data/trackingnet/TEST/frames`, and each video directory contains all images of the video. Similar file structures can be seen in `data/trackingnet/TRAIN_{*}/frames`.

There are 2 JSON files in `data/trackingnet/annotations`:

`trackingnet_test.json`: JSON file containing the annotations information of the testing set in TrackingNet dataset.
`trackingnet_train.json`: JSON file containing the annotations information of the training set in TrackingNet dataset.

There are 2 TEXT files in `data/trackingnet/annotations`:

`trackingnet_test_infos.txt`: TEXT file containing the information of the testing set in TrackingNet dataset.
`trackingnet_train_infos.txt`: TEXT file containing the information of the training set in TrackingNet dataset.

6.2.9 The folder of data and annotations in OTB100

There are 98 video directories of OTB100 dataset in `data/otb100/data`, and the `img` folder under each video directory contains all images of the video.

There are only 1 JSON files in `data/otb100/annotations`:

`otb100.json`: JSON file containing the annotations information of the OTB100 dataset.

There are only 1 TEXT files in `data/otb100/annotations`:

`otb100_infos.txt`: TEXT file containing the information of the OTB100 dataset.

6.2.10 The folder of frames and annotations in GOT10k

There are training video directories in `data/got10k/train`, and each video directory contains all images of the video. Similar file structures can be seen in `data/got10k/test` and `data/got10k/val`.

There are 3 JSON files in `data/got10k/annotations`:

`got10k_train.json`: JSON file containing the annotations information of the training set in GOT10k dataset.

`got10k_test.json`: JSON file containing the annotations information of the testing set in GOT10k dataset.

`got10k_val.json`: JSON file containing the annotations information of the valuation set in GOT10k dataset.

There are 5 TEXT files in `data/got10k/annotations`:

`got10k_train_infos.txt`: TEXT file containing the information of the training set in GOT10k dataset.

`got10k_test_infos.txt`: TEXT file containing the information of the testing set in GOT10k dataset.

`got10k_val_infos.txt`: TEXT file containing the information of the valuation set in GOT10k dataset.

`got10k_train_vot_infos.txt`: TEXT file containing the information of the `train_vot` split in GOT10k dataset.

`got10k_val_vot_infos.txt`: TEXT file containing the information of the `val_vot` split in GOT10k dataset.

6.2.11 The folder of data and annotations in VOT2018

There are 60 video directories of VOT2018 dataset in `data/vot2018/data`, and the `color` folder under each video directory contains all images of the video.

There are only 1 JSON files in `data/vot2018/annotations`:

`vot2018.json`: JSON file containing the annotations information of the VOT2018 dataset.

There are only 1 TEXT files in `data/vot2018/annotations`:

`vot2018_infos.txt`: TEXT file containing the information of the VOT2018 dataset.

6.2.12 The folder of annotations in youtube_vis_2019/youtube_vis2021

There are 3 JSON files in `data/youtube_vis_2019/annotations` or `data/youtube_vis_2021/annotations`:

`youtube_vis_2019_train.json/youtube_vis_2021_train.json`: JSON file containing the annotations information of the training set in youtube_vis_2019/youtube_vis2021 dataset.

`youtube_vis_2019_valid.json/youtube_vis_2021_valid.json`: JSON file containing the annotations information of the validation set in youtube_vis_2019/youtube_vis2021 dataset.

`youtube_vis_2019_test.json/youtube_vis_2021_test.json`: JSON file containing the annotations information of the testing set in youtube_vis_2019/youtube_vis2021 dataset.

CHAPTER

SEVEN

RUN WITH EXISTING DATASETS AND MODELS

MMTracking provides various methods on existing benchmarks. Details about these methods and benchmarks are presented in model_zoo.md. This note will show how to perform common tasks on existing models and standard datasets, including:

- Inference existing models on a given video or image folder.
- Test (inference and evaluate) existing models on standard datasets.
- Train existing models on standard datasets.

7.1 Inference

We provide demo scripts to inference a given video or a folder that contains continuous images. The source codes are available here.

Note that if you use a folder as the input, the image names there must be **sortable**, which means we can re-order the images according to the numbers contained in the filenames. We now only support reading the images whose filenames end with '.jpg', '.jpeg' and '.png'.

7.1.1 Inference VID models

This script can inference an input video with a video object detection model.

```
python demo/demo_vid.py \
    ${CONFIG_FILE}\
    --input ${INPUT} \
    --checkpoint ${CHECKPOINT_FILE} \
    [--output ${OUTPUT}] \
    [--device ${DEVICE}] \
    [--show]
```

The `INPUT` and `OUTPUT` support both mp4 video format and the folder format.

Optional arguments:

- `OUTPUT`: Output of the visualized demo. If not specified, the `--show` is obligate to show the video on the fly.
- `DEVICE`: The device for inference. Options are `cpu` or `cuda:0`, etc.
- `--show`: Whether show the video on the fly.

Examples:

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

```
python ./demo/demo_vid.py \
    ./configs/vid/selsa/selsa_faster_rcnn_r101_dc5_1x_imagenetvid.py \
    --input ${VIDEO_FILE} \
    --checkpoint checkpoints/selsa_faster_rcnn_r101_dc5_1x_imagenetvid_20201218_172724-
↪aa961bcc.pth \
    --output ${OUTPUT} \
    --show
```

7.1.2 Inference MOT/VIS models

This script can inference an input video / images with a multiple object tracking or video instance segmentation model.

```
python demo/demo_mot_vis.py \
    ${CONFIG_FILE} \
    --input ${INPUT} \
    [--output ${OUTPUT}] \
    [--checkpoint ${CHECKPOINT_FILE}] \
    [--score-thr ${SCORE_THR} \
    [--device ${DEVICE}] \
    [--backend ${BACKEND}] \
    [--show]
```

The `INPUT` and `OUTPUT` support both mp4 video format and the folder format.

Optional arguments:

- `OUTPUT`: Output of the visualized demo. If not specified, the `--show` is obligate to show the video on the fly.
- `CHECKPOINT_FILE`: The checkpoint is optional in case that you already set up the pretrained models in the config by the key `pretrains`.
- `SCORE_THR`: The threshold of score to filter bboxes.
- `DEVICE`: The device for inference. Options are `cpu` or `cuda:0`, etc.
- `BACKEND`: The backend to visualize the boxes. Options are `cv2` and `plt`.
- `--show`: Whether show the video on the fly.

Examples of running mot model:

```
python demo/demo_mot_vis.py \
    configs/mot/deepsort/sort_faster-rcnn_fpn_4e_mot17-private.py \
    --input demo/demo.mp4 \
    --output mot.mp4 \
```

Important: When running `demo_mot_vis.py`, we suggest you use the config containing `private`, since `private` means the MOT method doesn't need external detections.

Examples of running vis model:

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

```
python demo/demo_mot_vis.py \
    configs/vis/masktrack_rcnn/masktrack_rcnn_r50_fpn_12e_youtubevis2019.py \
    --input ${VIDEO_FILE} \
    --checkpoint checkpoints/masktrack_rcnn_r50_fpn_12e_youtubevis2019_20211022_194830-
↪6ca6b91e.pth \
```

(continues on next page)

(continued from previous page)

```
    --output ${OUTPUT} \
    --show
```

7.1.3 Inference SOT models

This script can inference an input video with a single object tracking model.

```
python demo/demo_sot.py \
    ${CONFIG_FILE}\
    --input ${INPUT} \
    --checkpoint ${CHECKPOINT_FILE} \
    [--output ${OUTPUT}] \
    [--device ${DEVICE}] \
    [--show] \
    [--gt_bbox_file ${GT_BBOX_FILE}]
```

The `INPUT` and `OUTPUT` support both mp4 video format and the folder format.

Optional arguments:

- `OUTPUT`: Output of the visualized demo. If not specified, the `--show` is obligate to show the video on the fly.
- `DEVICE`: The device for inference. Options are `cpu` or `cuda:0`, etc.
- `--show`: Whether show the video on the fly.
- `--gt_bbox_file`: The gt_bbox file path of the video. We only use the gt_bbox of the first frame. If not specified, you would draw init bbox of the video manually.

Examples:

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

```
python ./demo/demo_sot.py \
    ./configs/sot/siamese_rpn/siamese_rpn_r50_20e_lasot.py \
    --input ${VIDEO_FILE} \
    --checkpoint checkpoints/siamese_rpn_r50_1x_lasot_20211203_151612-da4b3c66.pth \
    --output ${OUTPUT} \
    --show
```

7.2 Testing

This section will show how to test existing models on supported datasets. The following testing environments are supported:

- single GPU
- single node multiple GPU
- multiple nodes

During testing, different tasks share the same API and we only support `samples_per_gpu = 1`.

You can use the following commands for testing:

```
# single-gpu testing
python tools/test.py ${CONFIG_FILE} [--checkpoint ${CHECKPOINT_FILE}] [--out ${RESULT_
↪FILE}] [--eval ${EVAL_METRICS}]

# multi-gpu testing
./tools/dist_test.sh ${CONFIG_FILE} ${GPU_NUM} [--checkpoint ${CHECKPOINT_FILE}] [--out $
↪{RESULT_FILE}] [--eval ${EVAL_METRICS}]
```

Optional arguments:

- `CHECKPOINT_FILE`: Filename of the checkpoint. You do not need to define it when applying some MOT methods but specify the checkpoints in the config.
- `RESULT_FILE`: Filename of the output results in pickle format. If not specified, the results will not be saved to a file.
- `EVAL_METRICS`: Items to be evaluated on the results. Allowed values depend on the dataset, e.g., `bbox` is available for ImageNet VID, `track` is available for LaSOT, `bbox` and `track` are both suitable for MOT17.
- `--cfg-options`: If specified, the key-value pair optional cfg will be merged into config file
- `--eval-options`: If specified, the key-value pair optional eval cfg will be kwargs for dataset.evaluate() function, it's only for evaluation
- `--format-only`: If specified, the results will be formatted to the official format.

7.2.1 Examples of testing VID model

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

1. Test DFF on ImageNet VID, and evaluate the bbox mAP.

```
python tools/test.py configs/vid/dff/dff_faster_rcnn_r101_dc5_1x_imagenetvid.py \
    --checkpoint checkpoints/dff_faster_rcnn_r101_dc5_1x_imagenetvid_20201218_
↪172720-ad732e17.pth \
    --out results.pkl \
    --eval bbox
```

2. Test DFF with 8 GPUs on ImageNet VID, and evaluate the bbox mAP.

```
./tools/dist_test.sh configs/vid/dff/dff_faster_rcnn_r101_dc5_1x_imagenetvid.py 8 \
    --checkpoint checkpoints/dff_faster_rcnn_r101_dc5_1x_imagenetvid_20201218_
↪172720-ad732e17.pth \
    --out results.pkl \
    --eval bbox
```

7.2.2 Examples of testing MOT model

1. Test Tracktor on MOT17, and evaluate CLEAR MOT metrics.

```
python tools/test.py configs/mot/tracktor/tracktor_faster-rcnn_r50_fpn_4e_mot17-
↪public-half.py \
    --eval track
```

2. Test Tracktor with 8 GPUs on MOT17, and evaluate CLEAR MOT metrics.

```
./tools/dist_test.sh configs/mot/tracktor/tracktor_faster-rcnn_r50_fpn_4e_mot17-
↪public-half.py 8 \
    --eval track
```

3. If you want to test Tracktor with your detector and reid model, you need modify the corresponding key-value pair in config as follows:

```
model = dict(
    detector=dict(
        init_cfg=dict(
            type='Pretrained',
            checkpoint='/path/to/detector_model')),
    reid=dict(
        init_cfg=dict(
            type='Pretrained',
            checkpoint='/path/to/reid_model'))
    )
```

7.2.3 Examples of testing SOT model

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

1. Test SiameseRPN++ on LaSOT, and evaluate the success, precision and normed precision.

```
python tools/test.py configs/sot/siamese_rpn/siamese_rpn_r50_20e_lasot.py \
    --checkpoint checkpoints/siamese_rpn_r50_1x_lasot_20211203_151612-da4b3c66.pth \
    --out results.pkl \
    --eval track
```

2. Test SiameseRPN++ with 8 GPUs on LaSOT, and evaluate the success, precision and normed precision.

```
./tools/dist_test.sh configs/sot/siamese_rpn/siamese_rpn_r50_20e_lasot.py 8 \
    --checkpoint checkpoints/siamese_rpn_r50_1x_lasot_20211203_151612-da4b3c66.pth \
    --out results.pkl \
    --eval track
```

7.2.4 Examples of testing VIS model

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

1. Test MaskTrack R-CNN on YouTube-VIS 2019, and generate a zip file for submission.

```
python tools/test.py \
    configs/vis/masktrack_rcnn/masktrack_rcnn_r50_fpn_12e_youtubevis2019.py \
    --checkpoint checkpoints/masktrack_rcnn_r50_fpn_12e_youtubevis2019_20211022_
↪194830-6ca6b91e.pth \
    --out ${RESULTS_PATH}/results.pkl \
    --format-only \
    --eval-options resfile_path=${RESULTS_PATH}
```

2. Test MaskTrack R-CNN with 8 GPUs on YouTube-VIS 2019, and generate a zip file for submission.

```
./tools/dist_test.sh \
    configs/vis/masktrack_rcnn/masktrack_rcnn_r50_fpn_12e_youtubevis2019.py \
    --checkpoint checkpoints/masktrack_rcnn_r50_fpn_12e_youtubevis2019_20211022_
↪194830-6ca6b91e.pth \
    --out ${RESULTS_PATH}/results.pkl \
    --format-only \
    --eval-options resfile_path=${RESULTS_PATH}
```

7.3 Training

MMTracking also provides out-of-the-box tools for training models. This section will show how to train *predefined* models (under configs) on standard datasets.

By default we evaluate the model on the validation set after each epoch, you can change the evaluation interval by adding the interval argument in the training config.

```
evaluation = dict(interval=12)  # This evaluate the model per 12 epoch.
```

Important: The default learning rate in all config files is for 8 GPUs. According to the Linear Scaling Rule, you need to set the learning rate proportional to the batch size if you use different GPUs or images per GPU, e.g., `lr=0.01` for 8 GPUs * 1 img/gpu and `lr=0.04` for 16 GPUs * 2 imgs/gpu.

7.3.1 Training on a single GPU

```
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`.

7.3.2 Training on multiple GPUs

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

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

Optional arguments remain the same as stated above.

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 ./tools/dist_train.sh ${CONFIG_FILE} 4
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 ./tools/dist_train.sh ${CONFIG_FILE} 4
```

7.3.3 Training on multiple nodes

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 bash tools/dist_train.sh
↪$CONFIG $GPUS
```

On the second machine:

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

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

If you launch with slurm, the command is the same as that on single machine described above, but you need refer to slurm_train.sh to set appropriate parameters and environment variables.

7.3.4 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}] ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${WORK_DIR}
```

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 `--options`. This is more recommended since it does not change the original configs.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py ${WORK_DIR} --options 'dist_params.port=29500'
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py ${WORK_DIR} --options 'dist_params.port=29501'
```

(continues on next page)

(continued from previous page)

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

   In `config1.py`, set

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

   In `config2.py`, set

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

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

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py ${WORK_DIR}
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py ${WORK_DIR}
```

7.3.5 Examples of training VID model

1. Train DFF on ImageNet VID and ImageNet DET, then evaluate the bbox mAP at the last epoch.

```
bash ./tools/dist_train.sh ./configs/vid/dff/dff_faster_rcnn_r101_dc5_1x_
↪imagenetvid.py 8 \
    --work-dir ./work_dirs/
```

7.3.6 Examples of training MOT model

For the training of MOT methods like SORT, DeepSORT and Tracktor, you need train a detector and a reid model rather than directly training the MOT model itself.

1. Train a detector model

   If you want to train a detector for multiple object tracking or other applications, to be compatible with MMDetection, you only need to add a line of `USE_MMDET=True` in the config and run it with the same manner in mmdetection. A base example can be found at faster_rcnn_r50_fpn.py.

   Please NOTE that there are some differences between the base config in MMTracking and MMDetection: `detector` is only a submodule of the `model`. For example, the config of Faster R-CNN in MMDetection follows

```
model = dict(
    type='FasterRCNN',
    ...
)
```

   But in MMTracking, the config follows

```
model = dict(
    detector=dict(
        type='FasterRCNN',
        ...
```

(continues on next page)

(continued from previous page)

```
    )
)
```

Here is an example to train a detector model on MOT17, and evaluate the bbox mAP after each epoch.

```
bash ./tools/dist_train.sh ./configs/det/faster-rcnn_r50_fpn_4e_mot17-half.py 8 \
    --work-dir ./work_dirs/
```

2. Train a ReID model

   You may want to train a ReID model for multiple object tracking or other applications. We support ReID model training in MMTracking, which is built upon MMClassification.

   Here is an example to train a reid model on MOT17, then evaluate the mAP after each epoch.

```
bash ./tools/dist_train.sh ./configs/reid/resnet50_b32x8_MOT17.py 8 \
    --work-dir ./work_dirs/
```

3. After training a detector and a ReID model, you can refer to Examples of testing MOT model to test your multi-object tracker.

7.3.7 Examples of training SOT model

1. Train SiameseRPN++ on COCO, ImageNet VID and ImageNet DET, then evaluate the success, precision and normed precision from the 10-th epoch to 20-th epoch.

```
bash ./tools/dist_train.sh ./configs/sot/siamese_rpn/siamese_rpn_r50_20e_lasot.py 8␣
↪\
    --work-dir ./work_dirs/
```

7.3.8 Examples of training VIS model

1. Train MaskTrack R-CNN on YouTube-VIS 2019 dataset. There are no evaluation results during training, since the annotations of validation dataset in YouTube-VIS are not provided.

```
bash ./tools/dist_train.sh ./configs/vis/masktrack_rcnn/masktrack_rcnn_r50_fpn_12e_
↪youtubevis2019.py 8 \
    --work-dir ./work_dirs/
```

CHAPTER

EIGHT

RUN WITH CUSTOMIZED DATASETS AND MODELS

In this note, you will know how to inference, test, and train with customized datasets and models.

The basic steps are as below:

1. Prepare the customized dataset (if applicable)
2. Prepare the customized model (if applicable)
3. Prepare a config
4. Train a new model
5. Test and inference the new model

8.1 1. Prepare the customized dataset

There are two ways to support a new dataset in MMTracking:

Reorganize the dataset into CocoVID format. Implement a new dataset.

Usually we recommend to use the first method which is usually easier than the second.

Details for customizing datasets are provided in tutorials/customize_dataset.md.

8.2 2. Prepare the customized model

We provide instructions for cutomizing models of different tasks.

- tutorials/customize_vid_model.md
- tutorials/customize_mot_model.md
- tutorials/customize_sot_model.md

8.3 3. Prepare a config

The next step is to prepare a config thus the dataset or the model can be successfully loaded. More details about the config system are provided at tutorials/config.md.

8.4 4. Train a new model

To train a model with the new config, you can simply run

```
python tools/train.py ${NEW_CONFIG_FILE}
```

For more detailed usages, please refer to the training instructions above.

8.5 5. Test and inference the new model

To test the trained model, you can simply run

```
python tools/test.py ${NEW_CONFIG_FILE} ${TRAINED_MODEL} --eval bbox track
```

For more detailed usages, please refer to the testing or inference instructions above.

CHAPTER

NINE

LEARN ABOUT CONFIGS

We use python files as our config system. You can find all the provided configs under $MMTracking/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/analysis/print_config.py /PATH/TO/CONFIG` to see the complete config.

9.1 Modify config through script arguments

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

- Update config keys of dict chains.

  The config options can be specified following the order of the dict keys in the original config. For example, `--cfg-options model.detector.backbone.norm_eval=False` changes the all BN modules in model backbones to `train` mode.

- Update keys inside a list of configs.

  Some config dicts are composed as a list in your config. For example, the testing pipeline `data.test.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.test.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.

9.2 Config File Structure

There are 3 basic component types under `config/_base_`, dataset, model, default_runtime. Many methods could be easily constructed with one of each like DFF, FGFA, SELSA, SORT, DeepSORT. 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 exiting methods. For example, if some modification is made base on Faster R-CNN, user may first inherit the basic Faster R-CNN structure by specifying `_base_ = ../../_base_/models/faster_rcnn_r50_dc5.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_rcnn` under `configs`,

Please refer to mmcv for detailed documentation.

9.3 Config Name Style

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

```
{model}_[model setting]_{backbone}_{neck}_[norm setting]_[misc]_[gpu x batch_per_gpu]_
↪{schedule}_{dataset}
```

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

- `{model}`: model type like `dff`, `tracktor`, `siamese_rpn`, etc.
- `[model setting]`: specific setting for some model, like `faster_rcnn` for `dff`,`tracktor`, etc.
- `{backbone}`: backbone type like `r50` (ResNet-50), `x101` (ResNeXt-101).
- `{neck}`: neck type like `fpn`, `c5`.
- `[norm_setting]`: `bn` (Batch Normalization) is used unless specified, other norm layer type could be `gn` (Group Normalization), `syncbn` (Synchronized Batch Normalization). `gn-head/gn-neck` indicates GN is applied in head/neck only, while `gn-all` means GN is applied in the entire model, e.g. backbone, neck, head.
- `[misc]`: miscellaneous setting/plugins of model, e.g. `dconv`, `gcb`, `attention`, `albu`, `mstrain`.
- `[gpu x batch_per_gpu]`: GPUs and samples per GPU, `8x2` is used by default.
- `{schedule}`: training schedule, options is `4e`, `7e`, `20e`, etc. `20e` denotes 20 epochs.
- `{dataset}`: dataset like `imagenetvid`, `mot17`, `lasot`.

9.4 Detailed analysis of Config File

Please refer to the corresponding page for config file structure of different tasks.

Video Object Detection

Multi Object Tracking

Single Object Tracking

9.5 FAQ

9.5.1 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.

9.5.2 Use intermediate variables in configs

Some intermediate variables are used in the configs files, like `train_pipeline/test_pipeline` in datasets. It's worth noting that when modifying intermediate variables in the children configs, user need to pass the intermediate variables into corresponding fields again. For example, we would like to use testing strategy of adaptive stride to test a SELSA. `ref_img_sampler` is intermediate variable we would like modify.

```
_base_ = ['./selsa_faster_rcnn_r50_dc5_1x_imagenetvid.py']

# dataset settings
ref_img_sampler = dict(
    _delete_=True,
    num_ref_imgs=14,
    frame_range=[-7, 7],
    method='test_with_adaptive_stride')
data = dict(
    val=dict(
        ref_img_sampler=ref_img_sampler),
    test=dict(
        ref_img_sampler=ref_img_sampler))
```

We first define the new `ref_img_sampler` and pass them into `data`.

CHAPTER

TEN

CUSTOMIZE DATASETS

To customize a new dataset, you can convert them to the existing CocoVID style or implement a totally new dataset. In MMTracking, we recommend to convert the data into CocoVID style and do the conversion offline, thus you can use the `CocoVideoDataset` directly. In this case, you only need to modify the config's data annotation paths and the `classes`.

10.1 Convert the dataset into CocoVID style

10.1.1 The CocoVID annotation file

The annotation json files in CocoVID style has the following necessary keys:

- `videos`: contains a list of videos. Each video is a dictionary with keys `name`, `id`. Optional keys include `fps`, `width`, and `height`.
- `images`: contains a list of images. Each image is a dictionary with keys `file_name`, `height`, `width`, `id`, `frame_id`, and `video_id`. Note that the `frame_id` is **0-index** based.
- `annotations`: contains a list of instance annotations. Each annotation is a dictionary with keys `bbox`, `area`, `id`, `category_id`, `instance_id`, `image_id` and `video_id`. The `instance_id` is only required for tracking.
- `categories`: contains a list of categories. Each category is a dictionary with keys `id` and `name`.

A simple example is presented at here.

The examples of converting existing datasets are presented at here.

10.1.2 Modify the config

After the data pre-processing, the users need to further modify the config files to use the dataset. Here we show an example of using a custom dataset of 5 classes, assuming it is also in CocoVID format.

In `configs/my_custom_config.py`:

```
...
# dataset settings
dataset_type = 'CocoVideoDataset'
classes = ('a', 'b', 'c', 'd', 'e')
...
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
```

(continues on next page)

(continued from previous page)

```
    train=dict(
        type=dataset_type,
        classes=classes,
        ann_file='path/to/your/train/data',
        ...),
    val=dict(
        type=dataset_type,
        classes=classes,
        ann_file='path/to/your/val/data',
        ...),
    test=dict(
        type=dataset_type,
        classes=classes,
        ann_file='path/to/your/test/data',
        ...))
...
```

10.2 Using dataset wrappers

MMTracking also supports some dataset wrappers to mix the dataset or modify the dataset distribution for training. Currently it supports to three dataset wrappers as below:

- `RepeatDataset`: simply repeat the whole dataset.
- `ClassBalancedDataset`: repeat dataset in a class balanced manner.
- `ConcatDataset`: concat datasets.

10.2.1 Repeat dataset

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

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

10.2.2 Class balanced dataset

We use `ClassBalancedDataset` as wrapper to repeat the dataset based on category frequency. The dataset to repeat needs to instantiate function `self.get_cat_ids(idx)` to support `ClassBalancedDataset`. For example, to repeat `Dataset_A` with `oversample_thr=1e-3`, the config looks like the following

```
dataset_A_train = dict(
        type='ClassBalancedDataset',
        oversample_thr=1e-3,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

10.2.3 Concatenate dataset

There are three ways to concatenate the dataset.

1. If the datasets you want to concatenate are in the same type with different annotation files, you can concatenate the dataset configs like the following.

   ```
   dataset_A_train = dict(
       type='Dataset_A',
       ann_file = ['anno_file_1', 'anno_file_2'],
       pipeline=train_pipeline
   )
   ```

   If the concatenated dataset is used for test or evaluation, this manner supports to evaluate each dataset separately. To test the concatenated datasets as a whole, you can set `separate_eval=False` as below.

   ```
   dataset_A_train = dict(
       type='Dataset_A',
       ann_file = ['anno_file_1', 'anno_file_2'],
       separate_eval=False,
       pipeline=train_pipeline
   )
   ```

2. In case the dataset you want to concatenate is different, you can concatenate the dataset configs like the following.

   ```
   dataset_A_train = dict()
   dataset_B_train = dict()

   data = dict(
       imgs_per_gpu=2,
       workers_per_gpu=2,
       train = [
           dataset_A_train,
           dataset_B_train
       ],
       val = dataset_A_val,
       test = dataset_A_test
       )
   ```

If the concatenated dataset is used for test or evaluation, this manner also supports to evaluate each dataset separately.

3. We also support to define `ConcatDataset` explicitly as the following.

```
dataset_A_val = dict()
dataset_B_val = dict()

data = dict(
    imgs_per_gpu=2,
    workers_per_gpu=2,
    train=dataset_A_train,
    val=dict(
        type='ConcatDataset',
        datasets=[dataset_A_val, dataset_B_val],
        separate_eval=False))
```

This manner allows users to evaluate all the datasets as a single one by setting `separate_eval=False`.

Note:

1. The option `separate_eval=False` assumes the datasets use `self.data_infos` during evaluation. Therefore, CocoVID datasets do not support this behavior since CocoVID datasets do not fully rely on `self.data_infos` for evaluation. Combining different types of datasets and evaluating them as a whole is not tested thus is not suggested.
2. Evaluating `ClassBalancedDataset` and `RepeatDataset` is not supported thus evaluating concatenated datasets of these types is also not supported.

A more complex example that repeats `Dataset_A` and `Dataset_B` by N and M times, respectively, and then concatenates the repeated datasets is as the following.

```
dataset_A_train = dict(
    type='RepeatDataset',
    times=N,
    dataset=dict(
        type='Dataset_A',
        ...
        pipeline=train_pipeline
    )
)
dataset_A_val = dict(
    ...
    pipeline=test_pipeline
)
dataset_A_test = dict(
    ...
    pipeline=test_pipeline
)
dataset_B_train = dict(
    type='RepeatDataset',
    times=M,
    dataset=dict(
        type='Dataset_B',
        ...
        pipeline=train_pipeline
```

(continues on next page)

(continued from previous page)

```
    )
)
data = dict(
    imgs_per_gpu=2,
    workers_per_gpu=2,
    train = [
        dataset_A_train,
        dataset_B_train
    ],
    val = dataset_A_val,
    test = dataset_A_test
)
```

10.3 Subset of existing datasets

With existing dataset types, we can modify the class names of them to train subset of the annotations. For example, if you want to train only three classes of the current dataset, you can modify the classes of dataset. The dataset will filter out the ground truth boxes of other classes automatically.

```
classes = ('person', 'bicycle', 'car')
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

MMTracking also supports to read the classes from a file, which is common in real applications. For example, assume the `classes.txt` contains the name of classes as the following.

```
person
bicycle
car
```

Users can set the classes as a file path, the dataset will load it and convert it to a list automatically.

```
classes = 'path/to/classes.txt'
data = dict(
    train=dict(classes=classes),
    val=dict(classes=classes),
    test=dict(classes=classes))
```

CHAPTER

ELEVEN

CUSTOMIZE DATA PIPELINES

There are two types of data pipelines in MMTracking:

- Single image, which is consistent with MMDetection in most cases.
- Pair-wise / multiple images.

11.1 Data pipeline for a single image

For a single image, you may refer to the tutorial in MMDetection.

There are several differences in MMTracking:

- We implement `VideoCollect` which is similar to `Collect` in MMDetection but is more compatible with the video perception tasks. For example, the meta keys `frame_id` and `is_video_data` are collected by default.

11.2 Data pipeline for multiple images

In some cases, we may need to process multiple images simultaneously. This is basically because we need to sample reference images of the key image in the same video to facilitate the training or inference process.

Please firstly take a look at the case of a single images above because the case of multiple images is heavily rely on it. We explain the details of the pipeline below.

11.2.1 1. Sample reference images

We sample and load the annotations of the reference images once we get the annotations of the key image.

Take `CocoVideoDataset` as an example, there is a function `sample_ref_img` to sample and load the annotations of the reference images.

```
from mmdet.datasets import CocoDataset

class CocoVideoDataset(CocoDataset):

    def __init__(self,
                 ref_img_sampler=None,
                 *args,
                 **kwargs):
        super().__init__(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.ref_img_sampler = ref_img_sampler


    def ref_img_sampling(self, **kwargs):
        pass

    def prepare_data(self, idx):
        img_info = self.data_infos[idx]
        if self.ref_img_sampler is not None:
            img_infos = self.ref_img_sampling(img_info, **self.ref_img_sampler)
        ...
```

In this case, the loaded annotations is no longer a `dict` but `list[dict]` that contains the annotations for the key and reference images. The first item of the list indicates the annotations of the key image.

11.2.2 2. Sequentially process and collect the data

In this step, we apply the transformations and then collected the information of the images.

In contrast to the pipeline of a single image that take a dictionary as the input and also output a dictionary for the next transformation, the sequential pipelines take a list of dictionaries as the input and also output a list of dictionaries for the next transformation.

These sequential pipelines are generally inherited from the pipeline in MMDetection but process the list in a loop.

```
from mmdet.datasets.builder import PIPELINES
from mmdet.datasets.pipelines import LoadImageFromFile

@PIPELINES.register_module()
class LoadMultiImagesFromFile(LoadImageFromFile):

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

    def __call__(self, results):
        outs = []
        for _results in results:
            _results = super().__call__(_results)
            outs.append(_results)
        return outs
```

Sometimes you may need to add a parameter `share_params` to decide whether share the random seed of the transformation on these images.

11.2.3 3. Concat the reference images (if applicable)

If there are more than one reference image, we implement `ConcatVideoReferences` to collect the reference images to a dictionary. The length of the list is 2 after the process.

11.2.4 4. Format the output to a dictionary

In the end, we implement `SeqDefaultFormatBundle` to convert the list to a dictionary as the input of the model forward.

Here is an example of the data pipeline:

```
train_pipeline = [
    dict(type='LoadMultiImagesFromFile'),
    dict(type='SeqLoadAnnotations', with_bbox=True, with_track=True),
    dict(type='SeqResize', img_scale=(1000, 600), keep_ratio=True),
    dict(type='SeqRandomFlip', share_params=True, flip_ratio=0.5),
    dict(type='SeqNormalize', **img_norm_cfg),
    dict(type='SeqPad', size_divisor=16),
    dict(
        type='VideoCollect',
        keys=['img', 'gt_bboxes', 'gt_labels', 'gt_instance_ids']),
    dict(type='ConcatVideoReferences'),
    dict(type='SeqDefaultFormatBundle', ref_prefix='ref')
]
```

CHAPTER

TWELVE

CUSTOMIZE VID MODELS

We basically categorize model components into 3 types.

- detector: usually a detector to detect objects from an image, e.g., Faster R-CNN.
- motion: the component to compute motion information between two images, e.g., FlowNetSimple.
- aggregator: the component for aggregating features from multi images, e.g., EmbedAggregator.

12.1 Add a new detector

Please refer to tutorial in mmdetection for developping a new detector.

12.2 Add a new motion model

12.2.1 1. Define a motion model (e.g. MyFlowNet)

Create a new file `mmtrack/models/motion/my_flownet.py`.

```
from mmcv.runner import BaseModule

from ..builder import MOTION

@MOTION.register_module()
class MyFlowNet(BaseModule):

    def __init__(self,
                 arg1,
                 arg2):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

12.2.2 2. Import the module

You can either add the following line to `mmtrack/models/motion/__init__.py`,

```
from .my_flownet import MyFlowNet
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.motion.my_flownet.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

12.2.3 3. Modify the config file

```
motion=dict(
    type='MyFlowNet',
    arg1=xxx,
    arg2=xxx)
```

12.3 Add a new aggregator

12.3.1 1. Define a aggregator (e.g. MyAggregator)

Create a new file `mmtrack/models/aggregators/my_aggregator.py`.

```
from mmcv.runner import BaseModule

from ..builder import AGGREGATORS

@AGGREGATORS.register_module()
class MyAggregator(BaseModule):

    def __init__(self,
                 arg1,
                 arg2):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

12.3.2 2. Import the module

You can either add the following line to `mmtrack/models/aggregators/__init__.py`,

```
from .my_aggregator import MyAggregator
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.aggregators.my_aggregator.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

12.3.3 3. Modify the config file

```
aggregator=dict(
    type='MyAggregator',
    arg1=xxx,
    arg2=xxx)
```

CHAPTER

THIRTEEN

CUSTOMIZE MOT MODELS

We basically categorize model components into 5 types.

- tracker: the component that associate the objects across the video with the cues extracted by the components below.
- detector: usually a detector to detect objects from the input image, e.g., Faster R-CNN.
- motion: the component to compute motion information between consecutive frames, e.g., KalmanFilter.
- reid: usually an independent ReID model to extract the feature embeddings from the cropped image, e.g., BaseReID.
- track_head: the component to extract tracking cues but share the same backbone with the detector, e.g., a embedding head or a regression head.

13.1 Add a new tracker

13.1.1 1. Define a tracker (e.g. MyTracker)

Create a new file `mmtrack/models/mot/trackers/my_tracker.py`.

We implement a `BaseTracker` that provide basic APIs to maintain the tracks across the video. We recommend to inherit the new tracker from it. The users may refer to the documentations of BaseTracker for the details.

```
from mmtrack.models import TRACKERS
from .base_tracker import BaseTracker

@TRACKERS.register_module()
class MyTracker(BaseTracker):

    def __init__(self,
                 arg1,
                 arg2,
                 *args,
                 **kwargs):
        super().__init__(*args, **kwargs)
        pass

    def track(self, inputs):
        # implementation is ignored
        pass
```

13.1.2 2. Import the module

You can either add the following line to `mmtrack/models/mot/trackers/__init__.py`,

```
from .my_tracker import MyTracker
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.mot.trackers.my_tracker.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

13.1.3 3. Modify the config file

```
tracker=dict(
    type='MyTracker',
    arg1=xxx,
    arg2=xxx)
```

13.2 Add a new detector

Please refer to tutorial in mmdetection for developping a new detector.

13.3 Add a new motion model

13.3.1 1. Define a motion model (e.g. MyFlowNet)

Create a new file `mmtrack/models/motion/my_flownet.py`.

You can inherit the motion model from `BaseModule` in `mmcv.runner` if it is a deep learning module, and from `object` if not.

```
from mmcv.runner import BaseModule

from ..builder import MOTION

@MOTION.register_module()
class MyFlowNet(BaseModule):

    def __init__(self,
                 arg1,
                 arg2):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

13.3.2 2. Import the module

You can either add the following line to `mmtrack/models/motion/__init__.py`,

```
from .my_flownet import MyFlowNet
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.motion.my_flownet.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

13.3.3 3. Modify the config file

```
motion=dict(
    type='MyFlowNet',
    arg1=xxx,
    arg2=xxx)
```

13.4 Add a new reid model

13.4.1 1. Define a reid model (e.g. MyReID)

Create a new file `mmtrack/models/reid/my_reid.py`.

```
from mmcv.runner import BaseModule

from ..builder import REID

@REID.register_module()
class MyReID(BaseModule):

    def __init__(self,
                 arg1,
                 arg2):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

13.4.2 2. Import the module

You can either add the following line to `mmtrack/models/reid/__init__.py`,

```
from .my_reid import MyReID
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.reid.my_reid.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

13.4.3 3. Modify the config file

```
reid=dict(
    type='MyReID',
    arg1=xxx,
    arg2=xxx)
```

13.5 Add a new track head

13.5.1 1. Define a head (e.g. MyHead)

Create a new file `mmtrack/models/track_heads/my_head.py`.

```
from mmcv.runner import BaseModule

from mmdet.models import HEADS

@HEADS.register_module()
class MyHead(BaseModule):

    def __init__(self,
                 arg1,
                 arg2):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

13.5.2 2. Import the module

You can either add the following line to `mmtrack/models/track_heads/__init__.py`,

```
from .my_head import MyHead
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.track_heads.my_head.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

13.5.3 3. Modify the config file

```
track_head=dict(
    type='MyHead',
    arg1=xxx,
    arg2=xxx)
```

13.6 Add a new loss

13.6.1 1. define a loss (e.g. MyLoss)

Assume you want to add a new loss as `MyLoss`, for bounding box regression. To add a new loss function, the users need implement it in `mmtrack/models/losses/my_loss.py`. The decorator `weighted_loss` enable the loss to be weighted for each element.

```
import torch
import torch.nn as nn

from mmdet.models import LOSSES, weighted_loss

@weighted_loss
def my_loss(pred, target):
    assert pred.size() == target.size() and target.numel() > 0
    loss = torch.abs(pred - target)
    return loss

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

    def __init__(self, reduction='mean', loss_weight=1.0):
        super(MyLoss, self).__init__()
        self.reduction = reduction
        self.loss_weight = loss_weight

    def forward(self,
                pred,
```

(continues on next page)

(continued from previous page)

```
                target,
                weight=None,
                avg_factor=None,
                reduction_override=None):
        assert reduction_override in (None, 'none', 'mean', 'sum')
        reduction = (
            reduction_override if reduction_override else self.reduction)
        loss_bbox = self.loss_weight * my_loss(
            pred, target, weight, reduction=reduction, avg_factor=avg_factor)
        return loss_bbox
```

13.6.2 2. Import the module

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

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

Alternatively, you can add

```
custom_imports=dict(
    imports=['mmtrack.models.losses.my_loss'],
    allow_failed_imports=False)
```

to the config file and achieve the same goal.

13.6.3 3. Modify the config file

To use it, modify the `loss_xxx` field. Since MyLoss is for regression, you need to modify the `loss_bbox` field in the `head`.

```
loss_bbox=dict(type='MyLoss', loss_weight=1.0))
```

CHAPTER

FOURTEEN

CUSTOMIZE SOT MODELS

We basically categorize model components into 4 types.

- backbone: usually an FCN network to extract feature maps, e.g., ResNet, MobileNet.
- neck: the component between backbones and heads, e.g., ChannelMapper, FPN.
- head: the component for specific tasks, e.g., tracking bbox prediction.
- loss: the component in head for calculating losses, e.g., FocalLoss, L1Loss.

14.1 Add a new backbone

Here we show how to develop new components with an example of MobileNet.

14.1.1 1. Define a new backbone (e.g. MobileNet)

Create a new file `mmtrack/models/backbones/mobilenet.py`.

```
import torch.nn as nn
from mmcv.runner import BaseModule

from mmdet.models.builder import BACKBONES


@BACKBONES.register_module()
class MobileNet(BaseModule):

    def __init__(self, arg1, arg2, *args, **kwargs):
        pass

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

14.1.2 2. Import the module

You can either add the following line to `mmtrack/models/backbones/__init__.py`

```
from .mobilenet import MobileNet
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.backbones.mobilenet'],
    allow_failed_imports=False)
```

to the config file to avoid modifying the original code.

14.1.3 3. Use the backbone in your config file

```
model = dict(
    ...
    backbone=dict(
        type='MobileNet',
        arg1=xxx,
        arg2=xxx),
    ...
```

14.2 Add a new neck

14.2.1 1. Define a neck (e.g. MyFPN)

Create a new file `mmtrack/models/necks/my_fpn.py`.

```
from mmcv.runner import BaseModule

from mmdet.models.builder import NECKS

@NECKS.register_module()
class MyFPN(BaseModule):

    def __init__(self, arg1, arg2, *args, **kwargs):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

14.2.2 2. Import the module

You can either add the following line to mmtrack/models/necks/__init__.py,

```
from .my_fpn import MyFPN
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.necks.my_fpn.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

14.2.3 3. Modify the config file

```
neck=dict(
    type='MyFPN',
    arg1=xxx,
    arg2=xxx),
```

14.3 Add a new head

14.3.1 1. Define a head (e.g. MyHead)

Create a new file mmtrack/models/track_heads/my_head.py.

```
from mmcv.runner import BaseModule

from mmdet.models import HEADS

@HEADS.register_module()
class MyHead(BaseModule):

    def __init__(self, arg1, arg2, *args, **kwargs):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

14.3.2 2. Import the module

You can either add the following line to `mmtrack/models/track_heads/__init__.py`,

```
from .my_head import MyHead
```

or alternatively add

```
custom_imports = dict(
    imports=['mmtrack.models.track_heads.my_head.py'],
    allow_failed_imports=False)
```

to the config file and avoid modifying the original code.

14.3.3 3. Modify the config file

```
track_head=dict(
    type='MyHead',
    arg1=xxx,
    arg2=xxx)
```

14.4 Add a new loss

Please refer to Add a new loss for developping a new loss.

CHAPTER

FIFTEEN

CUSTOMIZE RUNTIME SETTINGS

15.1 Customize optimization settings

15.1.1 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.

```
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.

15.1.2 Customize self-implemented optimizer

1. Define a new optimizer

A customized optimizer could be defined as following.

Assume you want to add a optimizer named `MyOptimizer`, which has arguments `a`, `b`, and `c`. You need to create a new file named `mmtrack/core/optimizer/my_optimizer.py`.

```
from torch.optim import Optimizer
from mmcv.runner.optimizer import OPTIMIZERS


@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 options to achieve it.

- Modify `mmtrack/core/optimizer/__init__.py` to import it.

  The newly defined module should be imported in `mmtrack/core/optimizer/__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 = dict(imports=['mmtrack.core.optimizer.my_optimizer.py'], allow_
↪failed_imports=False)
```

The module `mmtrack.core.optimizer.my_optimizer.MyOptimizer` 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. `mmtrack.core.optimizer.my_optimizer.MyOptimizer` **cannot** be imported directly.

Actually users can use a totally different file directory structure using this importing method, as long as the module root can be located in `PYTHONPATH`.

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:

```
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

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

15.1.3 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.

```
from mmcv.utils import build_from_cfg

from mmcv.runner.optimizer import OPTIMIZER_BUILDERS, OPTIMIZERS
from mmtrack.utils import get_root_logger
from .my_optimizer import MyOptimizer


@OPTIMIZER_BUILDERS.register_module()
class MyOptimizerConstructor(object):

    def __init__(self, optimizer_cfg, paramwise_cfg=None):

    def __call__(self, model):
```

(continues on next page)

(continued from previous page)

```
        return my_optimizer
```

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

15.1.4 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:

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

  If your config inherits the base config which already sets the `optimizer_config`, 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 CyclicLrUpdater and CyclicMomentumUpdater.

```
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,
)
```

15.2 Customize training schedules

We support many other learning rate schedule here, such as `CosineAnnealing` and `Poly` schedule. Here are some examples

- Poly schedule:

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

- ConsineAnnealing schedule:

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

15.3 Customize workflow

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

```
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

```
[('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 `EvalHook` because `EvalHook` 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 validation set after each training epoch.

15.4 Customize hooks

15.4.1 Customize self-implemented hooks

1. Implement a new hook

There are some occasions when the users might need to implement a new hook. MMTracking supports customized hooks in training. Thus the users could implement a hook directly in mmtrack or their mmtrack-based codebases and use the hook by only modifying the config in training. Here we give an example of creating a new hook in mmtrack and using it in training.

```
from mmcv.runner import HOOKS, Hook


@HOOKS.register_module()
class MyHook(Hook):

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

(continues on next page)

(continued from previous page)

```
    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 `mmtrack/core/utils/my_hook.py` there are two ways to do that:

- Modify `mmtrack/core/utils/__init__.py` to import it.

  The newly defined module should be imported in `mmtrack/core/utils/__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=['mmtrack.core.utils.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.

15.4.2 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

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

15.4.3 Modify default runtime hooks

There are some common hooks that are not registered through `custom_hooks`, 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 covers 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 hook

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

```
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 hook

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.

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

Evaluation hook

The config of `evaluation` will be used to initialize the `EvalHook`. Except keys like `interval`, `start` and so on, other arguments such as `metric` will be passed to the `dataset.evaluate()`

```
evaluation = dict(interval=1, metric='bbox')
```

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

CHAPTER

SIXTEEN

MOT TEST-TIME PARAMETER SEARCH

`tools/analysis/mot/mot_param_search.py` can search the parameters of the `tracker` in MOT models. It is used as the same manner with `tools/test.py` but different in the configs.

Here is an example that shows how to modify the configs:

1. Define the desirable evaluation metrics to record.

   For example, you can define the search metrics as

   ```
   search_metrics = ['MOTA', 'IDF1', 'FN', 'FP', 'IDs', 'MT', 'ML']
   ```

2. Define the parameters and the values to search.

   Assume you have a tracker like

   ```
   model = dict(
       tracker=dict(
           type='BaseTracker',
           obj_score_thr=0.5,
           match_iou_thr=0.5
       )
   )
   ```

   If you want to search the parameters of the tracker, just change the value to a list as follow

   ```
   model = dict(
       tracker=dict(
           type='BaseTracker',
           obj_score_thr=[0.4, 0.5, 0.6],
           match_iou_thr=[0.4, 0.5, 0.6, 0.7]
       )
   )
   ```

   Then the script will test the totally 12 cases and log the results.

CHAPTER

SEVENTEEN

SIAMESERPN++ TEST-TIME PARAMETER SEARCH

`tools/analysis/sot/sot_siamrpn_param_search.py` can search the test-time tracking parameters in SiameseRPN++: `penalty_k`, `lr` and `window_influence`. You need to pass the searching range of each parameter into the argparser.

Example on UAV123 dataset:

```
./tools/analysis/sot/dist_sot_siamrpn_param_search.sh [${CONFIG_FILE}] [$GPUS] \
[--checkpoint ${CHECKPOINT}] [--log ${LOG_FILENAME}] [--eval ${EVAL}] \
[--penalty-k-range 0.01,0.22,0.05] [--lr-range 0.4,0.61,0.05] [--win-infu-range 0.01,0.
↪22,0.05]
```

Example on OTB100 dataset:

```
./tools/analysis/sot/dist_sot_siamrpn_param_search.sh [${CONFIG_FILE}] [$GPUS] \
[--checkpoint ${CHECKPOINT}] [--log ${LOG_FILENAME}] [--eval ${EVAL}] \
[--penalty-k-range 0.3,0.45,0.02] [--lr-range 0.35,0.5,0.02] [--win-infu-range 0.46,0.55,
↪0.02]
```

Example on VOT2018 dataset:

```
./tools/analysis/sot/dist_sot_siamrpn_param_search.sh [${CONFIG_FILE}] [$GPUS] \
[--checkpoint ${CHECKPOINT}] [--log ${LOG_FILENAME}] [--eval ${EVAL}] \
[--penalty-k-range 0.01,0.31,0.05] [--lr-range 0.2,0.51,0.05] [--win-infu-range 0.3,0.56,
↪0.05]
```

CHAPTER

EIGHTEEN

LOG ANALYSIS

`tools/analysis/analyze_logs.py` plots loss/mAP curves given a training log file.

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

Examples:

- Plot the classification loss of some run.

```
python tools/analysis/analyze_logs.py plot_curve log.json --keys loss_cls --legend␣
↪loss_cls
```

- Plot the classification and regression loss of some run, and save the figure to a pdf.

```
python tools/analysis/analyze_logs.py plot_curve log.json --keys loss_cls loss_bbox␣
↪--out losses.pdf
```

- Compare the bbox mAP of two runs in the same figure.

```
python tools/analysis/analyze_logs.py plot_curve log1.json log2.json --keys bbox_
↪mAP --legend run1 run2
```

- Compute the average training speed.

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

  The output is expected to be like the following.

```
-----Analyze train time of work_dirs/some_exp/20190611_192040.log.json-----
slowest epoch 11, average time is 1.2024
fastest epoch 1, average time is 1.1909
time std over epochs is 0.0028
average iter time: 1.1959 s/iter
```

CHAPTER

NINETEEN

MODEL CONVERSION

19.1 Prepare a model for publishing

`tools/analysis/publish_model.py` helps users to prepare their model for publishing.

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

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

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

E.g.,

```
python tools/analysis/publish_model.py work_dirs/dff_faster_rcnn_r101_dc5_1x_imagenetvid/
↪latest.pth dff_faster_rcnn_r101_dc5_1x_imagenetvid.pth
```

The final output filename will be `dff_faster_rcnn_r101_dc5_1x_imagenetvid_20201230-{hash id}.pth`.

CHAPTER

TWENTY

MISCELLANEOUS

20.1 Print the entire config

`tools/analysis/print_config.py` prints the whole config verbatim, expanding all its imports.

```
python tools/analysis/print_config.py ${CONFIG} [-h] [--options ${OPTIONS [OPTIONS...]}]
```

CHAPTER

TWENTYONE

MODEL SERVING

In order to serve an `MMTracking` model with `TorchServe`, you can follow the steps:

21.1 1. Convert model from MMTracking to TorchServe

```
python tools/torchserve/mmtrack2torchserve.py ${CONFIG_FILE} ${CHECKPOINT_FILE} \
--output-folder ${MODEL_STORE} \
--model-name ${MODEL_NAME}
```

${MODEL_STORE} needs to be an absolute path to a folder.

21.2 2. Build `mmtrack-serve` docker image

```
docker build -t mmtrack-serve:latest docker/serve/
```

21.3 3. Run `mmtrack-serve`

Check the official docs for running TorchServe with docker.

In order to run in GPU, you need to install nvidia-docker. You can omit the `--gpus` argument in order to run in CPU.

Example:

```
docker run --rm \
--cpus 8 \
--gpus device=0 \
-p8080:8080 -p8081:8081 -p8082:8082 \
--mount type=bind,source=$MODEL_STORE,target=/home/model-server/model-store \
mmtrack-serve:latest
```

Read the docs about the Inference (8080), Management (8081) and Metrics (8082) APIs

21.4 4. Test deployment

```
curl http://127.0.0.1:8080/predictions/${MODEL_NAME} -T demo/demo.mp4 -o result.mp4
```

The response will be a ".mp4" mask.

You can visualize the output as follows:

```
import cv2
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)
    while cap.isOpened():
        flag, frame = cap.read()
        if not flag:
            break
        cv2.imshow('result.mp4', frame)
        if cv2.waitKey(int(1000 / fps)) & 0xFF == ord('q'):
            break
    cap.release()
    cv2.destroyAllWindows()
```

And you can use `test_torchserve.py` to compare result of torchserve and pytorch, and visualize them.

```
python tools/torchserve/test_torchserve.py ${VIDEO_FILE} ${CONFIG_FILE} ${CHECKPOINT_
↪FILE} ${MODEL_NAME}
[--inference-addr ${INFERENCE_ADDR}] [--result-video ${RESULT_VIDEO}] [--device ${DEVICE}
↪]
[--score-thr ${SCORE_THR}]
```

Example:

```
python tools/torchserve/test_torchserve.py \
demo/demo.mp4 \
configs/vid/selsa/selsa_faster_rcnn_r101_dc5_1x_imagenetvid.py \
checkpoint/selsa_faster_rcnn_r101_dc5_1x_imagenetvid_20201218_172724-aa961bcc.pth \
selsa \
--result-video=result.mp4
```

CHAPTER

TWENTYTWO

CHANGELOG

22.1 v0.13.0 (29/04/2022)

22.1.1 Highlights

- Support tracking colab tutorial (#511)

22.1.2 New Features

- Refactor the training datasets of SiamRPN++ (#496), (#518)
- Support loading data from ceph for SOT datasets (#494)
- Support loading data from ceph for MOT challenge dataset (#517)
- Support evaluation metric for VIS task (#501)

22.1.3 Bug Fixes

- Fix a bug in the LaSOT datasets and update the pretrained models of STARK (#483), (#503)
- Fix a bug in the format_results function of VIS task (#504)

22.2 v0.12.0 (01/04/2022)

22.2.1 Highlights

- Support QDTrack algorithm in MOT (#433), (#451), (#461), (#469)

22.2.2 Bug Fixes

- Support empty tensor for selsa aggregator (#463)

22.3 v0.11.0 (04/03/2022)

22.3.1 Highlights

- Support STARK algorithm in SOT (#443), (#440), (#434), (#438), (#435), (#426)
- Support HOTA evaluation metrics for MOT (#417)

22.3.2 New Features

- Support TAO dataset in MOT (#415)

22.4 v0.10.0 (10/02/2022)

22.4.1 New Features

- Support CPU training (#404)

22.4.2 Improvements

- Refactor SOT datasets (#401), (#402), (#393)

22.5 v0.9.0 (05/01/2022)

22.5.1 Highlights

- Support arXiv 2021 manuscript 'ByteTrack: Multi-Object Tracking by Associating Every Detection Box' (#385), (#383), (#372)
- Support ICCV 2019 paper 'Video Instance Segmentation' (#304), (#303), (#298), (#292)

22.5.2 New Features

- Support CrowdHuman dataset for MOT (#366)
- Support VOT2018 dataset for SOT (#305)
- Support YouTube-VIS dataset for VIS (#290)

22.5.3 Bug Fixes

- Fix two significant bugs in SOT and provide new SOT pretrained models (#349)

22.5.4 Improvements

- Refactor LaSOT, TrackingNet dataset and support GOT-10K datasets (#296)
- Support persisitent workers (#348)

22.6 v0.8.0 (03/10/2021)

22.6.1 New Features

- Support OTB100 dataset in SOT (#271)
- Support TrackingNet dataset in SOT (#268)
- Support UAV123 dataset in SOT (#260)

22.6.2 Bug Fixes

- Fix a bug in mot_param_search.py (#270)

22.6.3 Improvements

- Use PyTorch sphinx theme (#274)
- Use pycocotools instead of mmpycocotools (#263)

22.7 v0.7.0 (03/09/2021)

22.7.1 Highlights

- Release code of AAAI 2021 paper 'Temporal ROI Align for Video Object Recognition' (#247)
- Refactor English documentations (#243)
- Add Chinese documentations (#248), (#250)

22.7.2 New Features

- Support fp16 training and testing (#230)
- Release model using ResNeXt-101 as backbone for all VID methods (#254)
- Support the results of Tracktor on MOT15, MOT16 and MOT20 datasets (#217)
- Support visualization for single gpu test (#216)

22.7.3 Bug Fixes

- Fix a bug in MOTP evaluation (#235)
- Fix two bugs in reid training and testing (#249)

22.7.4 Improvements

- Refactor anchor in SiameseRPN++ (#229)
- Unify model initialization (#235)
- Refactor unittest (#231)

22.8 v0.6.0 (30/07/2021)

22.8.1 Highlights

- Fix training bugs of all three tasks (#219), (#221)

22.8.2 New Features

- Support error visualization for mot task (#212)

22.8.3 Bug Fixes

- Fix a bug in SOT demo (#213)

22.8.4 Improvements

- Use MMCV registry (#220)
- Add README.md for reid training (#210)
- Modify dict keys of the outputs of SOT (#223)
- Add Chinese docs including install.md, quick_run.md, model_zoo.md, dataset.md (#205), (#214)

22.9 v0.5.3 (01/07/2021)

22.9.1 New Features

- Support ReID training (#177), (#179), (#180), (#181),
- Support MIM (#158)

22.9.2 Bug Fixes

- Fix evaluation hook (#176)
- Fix a typo in vid config (#171)

22.9.3 Improvements

- Refactor nms config (#167)

22.10 v0.5.2 (03/06/2021)

22.10.1 Improvements

- Fixed typos (#104, #121, #145)
- Added conference reference (#111)
- Updated the link of CONTRIBUTING to mmcv (#112)
- Adapt updates in mmcv (FP16Hook) (#114, #119)
- Added bibtex and links to other codebases (#122)
- Added docker files (#124)
- Used `collect_env` in mmcv (#129)
- Added and updated Chinese README (#135, #147, #148)

22.11 v0.5.1 (01/02/2021)

22.11.1 Bug Fixes

- Fixed ReID checkpoint loading (#80)
- Fixed empty tensor in `track_result` (#86)
- Fixed `wait_time` in MOT demo script (#92)

22.11.2 Improvements

- Support single-stage detector for DeepSORT (#100)

22.12 v0.5.0 (04/01/2021)

22.12.1 Highlights

- MMTracking is released!

22.12.2 New Features

- Support video object detection methods: DFF, FGFA, SELSA
- Support multi object tracking methods: SORT/DeepSORT, Tracktor
- Support single object tracking methods: SiameseRPN++

CHAPTER

TWENTYTHREE

ENGLISH

CHAPTER

TWENTYFOUR

CHAPTER

TWENTYFIVE

MMTRACK.APIS

CHAPTER

TWENTYSIX

MMTRACK.CORE

26.1 anchor

26.2 evaluation

26.3 motion

26.4 optimizer

26.5 track

26.6 utils

CHAPTER

TWENTYSEVEN

MMTRACK.DATASETS

27.1 datasets

27.2 parsers

27.3 pipelines

27.4 samplers

CHAPTER

TWENTYEIGHT

MMTRACK.MODELS

28.1 mot

28.2 sot

28.3 vid

28.4 aggregators

class mmtrack.models.aggregators.**EmbedAggregator**(*num_convs=1*, *channels=256*, *kernel_size=3*, *norm_cfg=None*, *act_cfg={'type': 'ReLU'}*, *init_cfg=None*)

Embedding convs to aggregate multi feature maps.

This module is proposed in "Flow-Guided Feature Aggregation for Video Object Detection". FGFA.

Parameters

- **num_convs** (*int*) – Number of embedding convs.
- **channels** (*int*) – Channels of embedding convs. Defaults to 256.
- **kernel_size** (*int*) – Kernel size of embedding convs, Defaults to 3.
- **norm_cfg** (*dict*) – Configuration of normlization method after each conv. Defaults to None.
- **act_cfg** (*dict*) – Configuration of activation method after each conv. Defaults to dict(type='ReLU').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*, *ref_x*)

Aggregate reference feature maps *ref_x*.

The aggregation mainly contains two steps: 1. Computing the cos similarity between *x* and *ref_x*. 2. Use the normlized (i.e. softmax) cos similarity to weightedly sum *ref_x*.

Parameters

- **x** (*Tensor*) – of shape [1, C, H, W]
- **ref_x** (*Tensor*) – of shape [N, C, H, W]. N is the number of reference feature maps.

Returns The aggregated feature map with shape [1, C, H, W].

Return type Tensor

class mmtrack.models.aggregators.**SelsaAggregator**(*in_channels*, *num_attention_blocks=16*, *init_cfg=None*)

Selsa aggregator module.

This module is proposed in "Sequence Level Semantics Aggregation for Video Object Detection". SELSA.

Parameters

- **in_channels** (*int*) – The number of channels of the features of proposal.
- **num_attention_blocks** (*int*) – The number of attention blocks used in selsa aggregator module. Defaults to 16.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*, *ref_x*)

Aggregate the features *ref_x* of reference proposals.

The aggregation mainly contains two steps: 1. Use multi-head attention to computing the weight between *x* and *ref_x*. 2. Use the normlized (i.e. softmax) weight to weightedly sum *ref_x*.

Parameters

- **x** (*Tensor*) – of shape [N, C]. N is the number of key frame proposals.
- **ref_x** (*Tensor*) – of shape [M, C]. M is the number of reference frame proposals.

Returns The aggregated features of key frame proposals with shape [N, C].

Return type Tensor

28.5 backbones

28.6 losses

28.7 motion

28.8 reid

28.9 roi_heads

28.10 track_heads

28.11 builder

CHAPTER

TWENTYNINE

MMTRACK.UTILS

mmtrack.utils.**collect_env**()
Collect the information of the running environments.

mmtrack.utils.**get_root_logger**(*log_file=None*, *log_level=20*)
Get root logger.

Parameters

- **log_file** (*str*) – File path of log. Defaults to None.
- **log_level** (*int*) – The level of logger. Defaults to logging.INFO.

Returns The obtained logger

Return type logging.Logger

CHAPTER

THIRTY

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX