MMTracking

Release 0.14.0

MMTracking Authors

Apr 25, 2023

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: 13
  - ABSTRACT: 13
- Number of checkpoints: 46
  - [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] Observation-Centric SORT: Rethinking SORT for Robust Multi-Object Tracking (1 ckpts)
  - [ABSTRACT] Quasi-Dense Similarity Learning for Multiple Object Tracking (4 ckpts)
  - [ABSTRACT] Tracking without Bells and Whistles (7 ckpts)
  - [ABSTRACT] MixFormer: End-to-End Tracking with Iterative Mixed Attention (2 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 (ECCV 2022)

Please refer to ByteTrack for details.

5.3.5 OC-SORT (ArXiv 2022)

Please refer to OC-SORT 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.

5.4.3 MixFormer (CVPR 2022)

Please refer to MixFormer 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) is needed. CrowdHuman and LVIS can be served as complementary datasets.
- 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 download 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 -o ./
↪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 contains 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 contains the annotations information of the training set in ImageNet VID dataset.

`imagenet_vid_val.json`: JSON file contains 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 contains the annotations information of the training set in MOT17 dataset.

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

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

`test_detections.pkl`: Pickle file contains 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 as`half-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 contains the annotations information of the training set in CrowdHuman dataset.
`crowdhuman_val.json`: JSON file contains 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 contains the mapping relationship between COCO and LVIS categories.

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

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

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

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

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

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

`lvis_v1_image_info_test_dev.json`: JSON file contains 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 contains 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 contains the converted results for test set.

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

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

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

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

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

`validation_with_freeform.json`: JSON file contains 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 contains the annotations information of the training set in LaSOT dataset. `lasot_test.json`: JSON file contains 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 contains the annotations information of the training set in LaSOT dataset. `lasot_test_infos.txt`: TEXT file contains 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 contains the annotations information of the UAV123 dataset.

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

`uav123_infos.txt`: TEXT file contains 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 contains the annotations information of the testing set in TrackingNet dataset.
`trackingnet_train.json`: JSON file contains 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 contains the information of the testing set in TrackingNet dataset.
`trackingnet_train_infos.txt`: TEXT file contains 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 contains the annotations information of the OTB100 dataset.

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

`otb100_infos.txt`: TEXT file contains 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 contains the annotations information of the training set in GOT10k dataset.

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

`got10k_val.json`: JSON file contains 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 contains the information of the training set in GOT10k dataset.

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

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

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

`got10k_val_vot_infos.txt`: TEXT file contains 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 contains the annotations information of the VOT2018 dataset.

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

`vot2018_infos.txt`: TEXT file contains 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 contains 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 contains 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 contains 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 v0.14.0 (19/09/2022)

22.1.1 Highlights

- Introduce the 1.0.0rc0 version of MMTracking (#725)

22.1.2 New Features

- Support OC-SORT method for MOT (#545)
- Support multi-class tracking in ByteTrack (#548)
- Support DanceTrack dataset for MOT (#543)
- Support TAO dataset for QDTrack (#585)

22.2 v0.13.0 (29/04/2022)

22.2.1 Highlights

- Support tracking colab tutorial (#511)

22.2.2 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)

22.2.3 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)

22.3 v0.12.0 (01/04/2022)

22.3.1 Highlights

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

22.3.2 Bug Fixes

- Support empty tensor for selsa aggregator (#463)

22.4 v0.11.0 (04/03/2022)

22.4.1 Highlights

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

22.4.2 New Features

- Support TAO dataset in MOT (#415)

22.5 v0.10.0 (10/02/2022)

22.5.1 New Features

- Support CPU training (#404)

22.5.2 Improvements

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

22.6 v0.9.0 (05/01/2022)

22.6.1 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)

22.6.2 New Features

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

22.6.3 Bug Fixes

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

22.6.4 Improvements

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

22.7 v0.8.0 (03/10/2021)

22.7.1 New Features

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

22.7.2 Bug Fixes

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

22.7.3 Improvements

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

22.8 v0.7.0 (03/09/2021)

22.8.1 Highlights

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

22.8.2 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)

22.8.3 Bug Fixes

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

22.8.4 Improvements

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

22.9 v0.6.0 (30/07/2021)

22.9.1 Highlights

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

22.9.2 New Features

- Support error visualization for mot task (#212)

22.9.3 Bug Fixes

- Fix a bug in SOT demo (#213)

22.9.4 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)

22.10 v0.5.3 (01/07/2021)

22.10.1 New Features

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

22.10.2 Bug Fixes

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

22.10.3 Improvements

- Refactor nms config (#167)

22.11 v0.5.2 (03/06/2021)

22.11.1 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)

22.12 v0.5.1 (01/02/2021)

22.12.1 Bug Fixes

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

22.12.2 Improvements

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

22.13 v0.5.0 (04/01/2021)

22.13.1 Highlights

- MMTracking is released!

22.13.2 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

mmtrack.apis.**inference_mot**(*model*, *img*, *frame_id*)

Inference image(s) with the mot model.

Parameters

- **model** (*nn.Module*) – The loaded mot model.
- **img** (*str | ndarray*) – Either image name or loaded image.
- **frame_id** (*int*) – frame id.

Returns ndarray]: The tracking results.

Return type dict[str

mmtrack.apis.**inference_sot**(*model*, *image*, *init_bbox*, *frame_id*)

Inference image with the single object tracker.

Parameters

- **model** (*nn.Module*) – The loaded tracker.
- **image** (*ndarray*) – Loaded images.
- **init_bbox** (*ndarray*) – The target needs to be tracked.
- **frame_id** (*int*) – frame id.

Returns ndarray]: The tracking results.

Return type dict[str

mmtrack.apis.**inference_vid**(*model*, *image*, *frame_id*, *ref_img_sampler={'frame_stride': 10, 'num_left_ref_imgs': 10}*)

Inference image with the video object detector.

Parameters

- **model** (*nn.Module*) – The loaded detector.
- **image** (*ndarray*) – Loaded images.
- **frame_id** (*int*) – Frame id.
- **ref_img_sampler** (*dict*) – The configuration for sampling reference images. Only used under video detector of fgfa style. Defaults to dict(frame_stride=2, num_left_ref_imgs=10).

Returns ndarray]: The detection results.

Return type dict[str

mmtrack.apis.**init_model**(*config*, *checkpoint=None*, *device='cuda:0'*, *cfg_options=None*, *verbose_init_params=False*)

Initialize a model from config file.

Parameters

- **config** (str or mmcv.Config) – Config file path or the config object.
- **checkpoint** (*str, optional*) – Checkpoint path. Default as None.
- **cfg_options** (*dict, optional*) – Options to override some settings in the used config. Default to None.
- **verbose_init_params** (*bool, optional*) – Whether to print the information of initialized parameters to the console. Default to False.

Returns The constructed detector.

Return type nn.Module

mmtrack.apis.**init_random_seed**(*seed=None*, *device='cuda'*)

Initialize random seed.

If the seed is not set, the seed will be automatically randomized, and then broadcast to all processes to prevent some potential bugs. :param seed: The seed. Default to None. :type seed: int, Optional :param device: The device where the seed will be put on.

Default to 'cuda'.

Returns Seed to be used.

Return type int

mmtrack.apis.**multi_gpu_test**(*model*, *data_loader*, *tmpdir=None*, *gpu_collect=False*)

Test model with multiple gpus.

This method tests model with multiple gpus and collects the results under two different modes: gpu and cpu modes. By setting 'gpu_collect=True' it encodes results to gpu tensors and use gpu communication for results collection. On cpu mode it saves the results on different gpus to 'tmpdir' and collects them by the rank 0 worker. 'gpu_collect=True' is not supported for now.

Parameters

- **model** (*nn.Module*) – Model to be tested.
- **data_loader** (*nn.Dataloader*) – Pytorch data loader.
- **tmpdir** (*str*) – Path of directory to save the temporary results from different gpus under cpu mode. Defaults to None.
- **gpu_collect** (*bool*) – Option to use either gpu or cpu to collect results. Defaults to False.

Returns The prediction results.

Return type dict[str, list]

mmtrack.apis.**single_gpu_test**(*model*, *data_loader*, *show=False*, *out_dir=None*, *fps=3*, *show_score_thr=0.3*)

Test model with single gpu.

Parameters

- **model** (*nn.Module*) – Model to be tested.
- **data_loader** (*nn.Dataloader*) – Pytorch data loader.
- **show** (*bool, optional*) – If True, visualize the prediction results. Defaults to False.

- **out_dir** (*str, optional*) – Path of directory to save the visualization results. Defaults to None.
- **fps** (*int, optional*) – FPS of the output video. Defaults to 3.
- **show_score_thr** (*float, optional*) – The score threshold of visualization (Only used in VID for now). Defaults to 0.3.

Returns The prediction results.

Return type dict[str, list]

mmtrack.apis.**train_model**(*model*, *dataset*, *cfg*, *distributed=False*, *validate=False*, *timestamp=None*, *meta=None*)

Train model entry function.

Parameters

- **model** (*nn.Module*) – The model to be trained.
- **dataset** (Dataset) – Train dataset.
- **cfg** (*dict*) – The config dict for training.
- **distributed** (*bool*) – Whether to use distributed training. Default: False.
- **validate** (*bool*) – Whether to do evaluation. Default: False.
- **timestamp** (*str | None*) – Local time for runner. Default: None.
- **meta** (*dict | None*) – Meta dict to record some important information. Default: None

CHAPTER

TWENTYSIX

MMTRACK.CORE

26.1 anchor

class mmtrack.core.anchor.**SiameseRPNAnchorGenerator**(*strides*, **args*, ***kwargs*)

Anchor generator for siamese rpn.

Please refer to *mmdet/core/anchor/anchor_generator.py:AnchorGenerator* for detailed docstring.

gen_2d_hanning_windows(*featmap_sizes*, *device='cuda'*)

Generate 2D hanning window.

Parameters

- **featmap_sizes** (*list[torch.size]*) – List of torch.size recording the resolution (height, width) of the multi-level feature maps.
- **device** (*str*) – Device the tensor will be put on. Defaults to 'cuda'.

Returns List of 2D hanning window with shape (num_base_anchors[i] * featmap_sizes[i][0] * featmap_sizes[i][1]).

Return type list[Tensor]

gen_single_level_base_anchors(*base_size*, *scales*, *ratios*, *center=None*)

Generate base anchors of a single level feature map.

Parameters

- **base_size** (*int | float*) – Basic size of an anchor.
- **scales** (*torch.Tensor*) – Scales of the anchor.
- **ratios** (*torch.Tensor*) – The ratio between between the height and width of anchors in a single level.
- **center** (*tuple[float], optional*) – The center of the base anchor related to a single feature grid. Defaults to None.

Returns Anchors of one spatial location in a single level feature map in [tl_x, tl_y, br_x, br_y] format.

Return type torch.Tensor

26.2 evaluation

class mmtrack.core.evaluation.**DistEvalHook**(*dataloader: torch.utils.data.dataloader.DataLoader*, *start: Optional[int] = None*, *interval: int = 1*, *by_epoch: bool = True*, *save_best: Optional[str] = None*, *rule: Optional[str] = None*, *test_fn: Optional[Callable] = None*, *greater_keys: Optional[List[str]] = None*, *less_keys: Optional[List[str]] = None*, *broadcast_bn_buffer: bool = True*, *tmpdir: Optional[str] = None*, *gpu_collect: bool = False*, *out_dir: Optional[str] = None*, *file_client_args: Optional[dict] = None*, ***eval_kwargs*)

Please refer to *mmcv.runner.hooks.evaluation.py:DistEvalHook* for detailed docstring.

class mmtrack.core.evaluation.**EvalHook**(*dataloader: torch.utils.data.dataloader.DataLoader*, *start: Optional[int] = None*, *interval: int = 1*, *by_epoch: bool = True*, *save_best: Optional[str] = None*, *rule: Optional[str] = None*, *test_fn: Optional[Callable] = None*, *greater_keys: Optional[List[str]] = None*, *less_keys: Optional[List[str]] = None*, *out_dir: Optional[str] = None*, *file_client_args: Optional[dict] = None*, ***eval_kwargs*)

Please refer to *mmcv.runner.hooks.evaluation.py:EvalHook* for detailed docstring.

mmtrack.core.evaluation.**bbox2region**(*bbox*)

Convert bbox to Rectangle or Polygon Class object.

Parameters **bbox** (*ndarray*) – the format of rectangle bbox is (x1, y1, w, h); the format of polygon is (x1, y1, x2, y2, ...).

Returns Rectangle or Polygon Class object.

mmtrack.core.evaluation.**eval_mot**(*results*, *annotations*, *logger=None*, *classes=None*, *iou_thr=0.5*, *ignore_iof_thr=0.5*, *ignore_by_classes=False*, *nproc=4*)

Evaluation CLEAR MOT metrics.

Parameters

- **results** (*list[list[list[ndarray]]]*) – The first list indicates videos, The second list indicates images. The third list indicates categories. The ndarray indicates the tracking results.
- **annotations** (*list[list[dict]]*) – The first list indicates videos, The second list indicates images. The third list indicates the annotations of each video. Keys of annotations are
  - *bboxes*: numpy array of shape (n, 4)
  - *labels*: numpy array of shape (n,)
  - *instance_ids*: numpy array of shape (n,)
  - *bboxes_ignore* (optional): numpy array of shape (k, 4)
  - *labels_ignore* (optional): numpy array of shape (k,)
- **logger** (*logging.Logger | str | None, optional*) – The way to print the evaluation results. Defaults to None.
- **classes** (*list, optional*) – Classes in the dataset. Defaults to None.
- **iou_thr** (*float, optional*) – IoU threshold for evaluation. Defaults to 0.5.
- **ignore_iof_thr** (*float, optional*) – Iof threshold to ignore results. Defaults to 0.5.

- **ignore_by_classes** (*bool, optional*) – Whether ignore the results by classes or not. Defaults to False.
- **nproc** (*int, optional*) – Number of the processes. Defaults to 4.

Returns Evaluation results.

Return type dict[str, float]

mmtrack.core.evaluation.**eval_sot_accuracy_robustness**(*results*, *annotations*, *burnin=10*, *ignore_unknown=True*, *videos_wh=None*)

Calculate accuracy and robustness over all tracking sequences.

Parameters

- **results** (*list[list[ndarray]]*) – The first list contains the tracking results of each video. The second list contains the tracking results of each frame in one video. The ndarray have two cases:
  - **bbox: denotes the normal tracking box in [x1, y1, w, h]** format.
  - **special tracking state: [0] denotes the unknown state,** namely the skipping frame after failure, [1] denotes the initialized state, and [2] denotes the failed state.
- **annotations** (*list[ndarray]*) – The list contains the gt_bboxes of each video. The ndarray is gt_bboxes of one video. It's in (N, 4) shape. Each bbox is in (x1, y1, w, h) format.
- **burnin** – number of frames that have to be ignored after the re-initialization when calculating accuracy. Default is 10.
- **ignore_unknown** (*bool*) – whether ignore the skipping frames after failures when calculating accuracy. Default is True.
- **videos_wh** (*list[tuple(width, height), ...]*) – The list contains the width and height of each video. Default is None.

Returns float}: accuracy and robustness in EAO evaluation metric.

Return type dict{str

mmtrack.core.evaluation.**eval_sot_eao**(*results*, *annotations*, *interval=[100, 356]*, *videos_wh=None*)

Calculate EAO socre over all tracking sequences.

Parameters

- **results** (*list[list[ndarray]]*) – The first list contains the tracking results of each video. The second list contains the tracking results of each frame in one video. The ndarray have two cases:
  - **bbox: denotes the normal tracking box in [x1, y1, w, h]** format.
  - **special tracking state: [0] denotes the unknown state,** namely the skipping frame after failure, [1] denotes the initialized state, and [2] denotes the failed state.
- **annotations** (*list[ndarray]*) – The list contains the gt_bboxes of each video. The ndarray is gt_bboxes of one video. It's in (N, 4) shape. Each bbox is in (x1, y1, w, h) format.
- **interval** – an specified interval in EAO curve used to calculate the EAO score. There are different settings in different VOT challenge. Default is VOT2018 setting: [100, 356].
- **videos_wh** (*list[tuple(width, height), ...]*) – The list contains the width and height of each video. Default is None.

Returns EAO score in EAO evaluation metric.

Return type dict[str, float]

mmtrack.core.evaluation.**eval_sot_ope**(*results*, *annotations*, *visible_infos=None*)
Evaluation in OPE protocol.

Parameters

- **results** (*list[list[ndarray]]*) – The first list contains the tracking results of each video. The second list contains the tracking results of each frame in one video. The ndarray denotes the tracking box in [tl_x, tl_y, br_x, br_y] format.
- **annotations** (*list[ndarray]*) – The list contains the bbox annotations of each video. The ndarray is gt_bboxes of one video. It's in (N, 4) shape. Each bbox is in (x1, y1, x2, y2) format.
- **visible_infos** (*list[ndarray] | None*) – If not None, the list contains the visible information of each video. The ndarray is visibility (with bool type) of object in one video. It's in (N,) shape. Default to None.

Returns OPE style evaluation metric (i.e. success, norm precision and precision).

Return type dict[str, float]

mmtrack.core.evaluation.**eval_vis**(*test_results*, *vis_anns*, *logger=None*)
Evaluation on VIS metrics.

Parameters

- **test_results** (*dict(list[dict])*) – Testing results of the VIS dataset.
- **vis_anns** (*dict(list[dict])*) – The annotation in the format of YouTube-VIS.
- **logger** (*logging.Logger | str | None*) – Logger used for printing related information during evaluation. Default: None.

Returns Evaluation results.

Return type dict[str, float]

26.3 motion

mmtrack.core.motion.**flow_warp_feats**(*x*, *flow*)
Use flow to warp feature map.

Parameters

- **x** (*Tensor*) – of shape (N, C, H_x, W_x).
- **flow** (*Tensor*) – of shape (N, C, H_f, W_f).

Returns The warpped feature map with shape (N, C, H_x, W_x).

Return type Tensor

26.4 optimizer

class mmtrack.core.optimizer.**SiameseRPNFp16OptimizerHook**(*backbone_start_train_epoch*, *backbone_train_layers*, ***kwargs*)

FP16Optimizer hook for siamese rpn.

Parameters

- **backbone_start_train_epoch** (*int*) – Start to train the backbone at *backbone_start_train_epoch*-th epoch. Note the epoch in this class counts from 0, while the epoch in the log file counts from 1.
- **backbone_train_layers** (*list(str)*) – List of str denoting the stages needed be trained in backbone.

before_train_epoch(*runner*)

If *runner.epoch* >= *self.backbone_start_train_epoch*, start to train the backbone.

class mmtrack.core.optimizer.**SiameseRPNLrUpdaterHook**(*lr_configs=[{'type': 'step', 'start_lr_factor': 0.2, 'end_lr_factor': 1.0, 'end_epoch': 5}, {'type': 'log', 'start_lr_factor': 1.0, 'end_lr_factor': 0.1, 'end_epoch': 20}]*, ***kwargs*)

Learning rate updater for siamese rpn.

Parameters **lr_configs** (*list[dict]*) – List of dict where each dict denotes the configuration of specifical learning rate updater and must have 'type'.

get_lr(*runner*, *base_lr*)

Get a specifical learning rate for each epoch.

class mmtrack.core.optimizer.**SiameseRPNOptimizerHook**(*backbone_start_train_epoch*, *backbone_train_layers*, ***kwargs*)

Optimizer hook for siamese rpn.

Parameters

- **backbone_start_train_epoch** (*int*) – Start to train the backbone at *backbone_start_train_epoch*-th epoch. Note the epoch in this class counts from 0, while the epoch in the log file counts from 1.
- **backbone_train_layers** (*list(str)*) – List of str denoting the stages needed be trained in backbone.

before_train_epoch(*runner*)

If *runner.epoch* >= *self.backbone_start_train_epoch*, start to train the backbone.

26.5 track

mmtrack.core.track.**depthwise_correlation**(*x*, *kernel*)

Depthwise cross correlation.

This function is proposed in SiamRPN++.

Parameters

- **x** (*Tensor*) – of shape (N, C, H_x, W_x).
- **kernel** (*Tensor*) – of shape (N, C, H_k, W_k).

Returns of shape (N, C, H_o, W_o). H_o = H_x - H_k + 1. So does W_o.

Return type Tensor

mmtrack.core.track.**embed_similarity**(*key_embeds*, *ref_embeds*, *method='dot_product'*, *temperature=- 1*)
Calculate feature similarity from embeddings.

Parameters

- **key_embeds** (*Tensor*) – Shape (N1, C).
- **ref_embeds** (*Tensor*) – Shape (N2, C).
- **method** (*str, optional*) – Method to calculate the similarity, options are 'dot_product' and 'cosine'. Defaults to 'dot_product'.
- **temperature** (*int, optional*) – Softmax temperature. Defaults to -1.

Returns Similarity matrix of shape (N1, N2).

Return type Tensor

mmtrack.core.track.**imrenormalize**(*img*, *img_norm_cfg*, *new_img_norm_cfg*)
Re-normalize the image.

Parameters

- **img** (*Tensor | ndarray*) – Input image. If the input is a Tensor, the shape is (1, C, H, W). If the input is a ndarray, the shape is (H, W, C).
- **img_norm_cfg** (*dict*) – Original configuration for the normalization.
- **new_img_norm_cfg** (*dict*) – New configuration for the normalization.

Returns Output image with the same type and shape of the input.

Return type Tensor | ndarray

mmtrack.core.track.**interpolate_tracks**(*tracks*, *min_num_frames=5*, *max_num_frames=20*)
Interpolate tracks linearly to make tracks more complete.

This function is proposed in "ByteTrack: Multi-Object Tracking by Associating Every Detection Box." **`ByteTrack<https://arxiv.org/abs/2110.06864>`_**.

Parameters

- **tracks** (*ndarray*) – With shape (N, 7). Each row denotes (frame_id, track_id, x1, y1, x2, y2, score).
- **min_num_frames** (*int, optional*) – The minimum length of a track that will be interpolated. Defaults to 5.
- **max_num_frames** (*int, optional*) – The maximum disconnected length in a track. Defaults to 20.

Returns

The interpolated tracks with shape (N, 7). Each row denotes (frame_id, track_id, x1, y1, x2, y2, score)

Return type ndarray

mmtrack.core.track.**outs2results**(*bboxes=None*, *labels=None*, *masks=None*, *ids=None*, *num_classes=None*, ***kwargs*)
Convert tracking/detection results to a list of numpy arrays.

Parameters

- **bboxes** (*torch.Tensor | np.ndarray*) – shape (n, 5)

- **labels** (*torch.Tensor | np.ndarray*) – shape (n,)
- **masks** (*torch.Tensor | np.ndarray*) – shape (n, h, w)
- **ids** (*torch.Tensor | np.ndarray*) – shape (n,)
- **num_classes** (*int*) – class number, not including background class

Returns

list(ndarray) | list[list[np.ndarray]]]: tracking/detection results of each class. It may contain keys as belows:

- **bbox_results (list[np.ndarray]): Each list denotes bboxes of one** category.
- **mask_results (list[list[np.ndarray]]): Each outer list denotes masks** of one category. Each inner list denotes one mask belonging to the category. Each mask has shape (h, w).

Return type dict[str

mmtrack.core.track.**results2outs**(*bbox_results=None*, *mask_results=None*, *mask_shape=None*, ***kwargs*)

Restore the results (list of results of each category) into the results of the model forward.

Parameters

- **bbox_results** (*list[np.ndarray]*) – Each list denotes bboxes of one category.
- **mask_results** (*list[list[np.ndarray]]*) – Each outer list denotes masks of one category. Each inner list denotes one mask belonging to the category. Each mask has shape (h, w).
- **mask_shape** (*tuple[int]*) – The shape (h, w) of mask.

Returns

tracking results of each class. It may contain keys as belows:

- bboxes (np.ndarray): shape (n, 5)
- labels (np.ndarray): shape (n,)
- masks (np.ndarray): shape (n, h, w)
- ids (np.ndarray): shape (n,)

Return type tuple

26.6 utils

mmtrack.core.utils.**crop_image**(*image*, *crop_region*, *crop_size*, *padding=(0, 0, 0)*)

Crop image based on *crop_region* and *crop_size*.

Parameters

- **image** (*ndarray*) – of shape (H, W, 3).
- **crop_region** (*ndarray*) – of shape (4,) in [x1, y1, x2, y2] format.
- **crop_size** (*int*) – Crop size.
- **padding** (*tuple | ndarray*) – of shape (3,) denoting the padding values.

Returns Cropped image of shape (crop_size, crop_size, 3).

Return type ndarray

mmtrack.core.utils.**imshow_mot_errors**(**args*, *backend='cv2'*, ***kwargs*)
Show the wrong tracks on the input image.

Parameters **backend** (*str, optional*) – Backend of visualization. Defaults to 'cv2'.

mmtrack.core.utils.**imshow_tracks**(**args*, *backend='cv2'*, ***kwargs*)
Show the tracks on the input image.

CHAPTER

TWENTYSEVEN

MMTRACK.DATASETS

27.1 datasets

class mmtrack.datasets.**BaseSOTDataset**(*img_prefix*, *pipeline*, *split*, *ann_file=None*, *test_mode=False*, *bbox_min_size=0*, *only_eval_visible=False*, *file_client_args={'backend': 'disk'}*, ***kwargs*)

Dataset of single object tracking. The dataset can both support training and testing mode.

Parameters

- **img_prefix** (*str*) – Prefix in the paths of image files.
- **pipeline** (*list[dict]*) – Processing pipeline.
- **split** (*str*) – Dataset split.
- **ann_file** (*str, optional*) – The file contains data information. It will be loaded and parsed in the *self.load_data_infos* function.
- **test_mode** (*bool, optional*) – Default to False.
- **bbox_min_size** (*int, optional*) – Only bounding boxes whose sizes are larger than *bbox_min_size* can be regarded as valid. Default to 0.
- **only_eval_visible** (*bool, optional*) – Whether to only evaluate frames where object are visible. Default to False.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. Default: dict(backend='disk').

evaluate(*results*, *metric=['track']*, *logger=None*)

Default evaluation standard is OPE.

Parameters

- **results** (*dict(list[ndarray])*) – tracking results. The ndarray is in (x1, y1, x2, y2, score) format.
- **metric** (*list, optional*) – defaults to ['track'].
- **logger** (*logging.Logger | str | None, optional*) – defaults to None.

get_ann_infos_from_video(*video_ind*)

Get annotation information in a video.

Parameters **video_ind** (*int*) – video index

Returns

{'bboxes': ndarray in (N, 4) shape, 'bboxes_isvalid': ndarray, 'visible':ndarray}. The annotation information in some datasets may contain 'visible_ratio'. The bbox is in (x1, y1, x2, y2) format.

Return type dict

`get_bboxes_from_video`(*video_ind*)
Get bboxes annotation about the instance in a video.

Parameters **video_ind** (*int*) – video index

Returns

in [N, 4] shape. The N is the number of bbox and the bbox is in (x, y, w, h) format.

Return type ndarray

`get_img_infos_from_video`(*video_ind*)
Get image information in a video.

Parameters **video_ind** (*int*) – video index

Returns {'filename': list[str], 'frame_ids':ndarray, 'video_id':int}

Return type dict

`get_len_per_video`(*video_ind*)
Get the number of frames in a video.

`get_visibility_from_video`(*video_ind*)
Get the visible information of instance in a video.

`load_as_video`
The self.data_info is a list, which the length is the number of videos. The default content is in the following format: [

{ 'video_path': the video path 'ann_path': the annotation path 'start_frame_id': the starting frame ID number contained in

the image name

'end_frame_id': the ending frame ID number contained in the image name

'framename_template': the template of image name

]

`pre_pipeline`(*results*)
Prepare results dict for pipeline.

The following keys in dict will be called in the subsequent pipeline.

`prepare_test_data`(*video_ind*, *frame_ind*)
Get testing data of one frame. We parse one video, get one frame from it and pass the frame information to the pipeline.

Parameters

- **video_ind** (*int*) – video index
- **frame_ind** (*int*) – frame index

Returns testing data of one frame.

Return type dict

prepare_train_data(*video_ind*)
Get training data sampled from some videos. We firstly sample two videos from the dataset and then parse the data information. The first operation in the training pipeline is frames sampling.

Parameters **video_ind** (*int*) – video index

Returns training data pairs, triplets or groups.

Return type dict

class mmtrack.datasets.**CocoVID**(**args: Any*, ***kwargs: Any*)
Inherit official COCO class in order to parse the annotations of bbox- related video tasks.

Parameters

- **annotation_file** (*str*) – location of annotation file. Defaults to None.
- **load_img_as_vid** (*bool*) – If True, convert image data to video data, which means each image is converted to a video. Defaults to False.

convert_img_to_vid(*dataset*)
Convert image data to video data.

createIndex()
Create index.

get_img_ids_from_ins_id(*insId*)
Get image ids from given instance id.

Parameters **insId** (*int*) – The given instance id.

Returns Image ids of given instance id.

Return type list[int]

get_img_ids_from_vid(*vidId*)
Get image ids from given video id.

Parameters **vidId** (*int*) – The given video id.

Returns Image ids of given video id.

Return type list[int]

get_ins_ids_from_vid(*vidId*)
Get instance ids from given video id.

Parameters **vidId** (*int*) – The given video id.

Returns Instance ids of given video id.

Return type list[int]

get_vid_ids(*vidIds=[]*)
Get video ids that satisfy given filter conditions.

Default return all video ids.

Parameters **vidIds** (*list[int]*) – The given video ids. Defaults to [].

Returns Video ids.

Return type list[int]

load_vids(*ids=[]*)
Get video information of given video ids.

Default return all videos information.

Parameters **ids** (*list[int]*) – The given video ids. Defaults to [].

Returns List of video information.

Return type list[dict]

class mmtrack.datasets.**CocoVideoDataset**(*load_as_video=True*, *key_img_sampler={'interval': 1}*, *ref_img_sampler={'filter_key_img': True, 'frame_range': 10, 'method': 'uniform', 'num_ref_imgs': 1, 'return_key_img': True, 'stride': 1}*, *test_load_ann=False*, **args*, ***kwargs*)

Base coco video dataset for VID, MOT and SOT tasks.

Parameters

- **load_as_video** (*bool*) – If True, using COCOVID class to load dataset, otherwise, using COCO class. Default: True.
- **key_img_sampler** (*dict*) – Configuration of sampling key images.
- **ref_img_sampler** (*dict*) – Configuration of sampling ref images.
- **test_load_ann** (*bool*) – If True, loading annotations during testing, otherwise, not loading. Default: False.

evaluate(*results*, *metric=['bbox', 'track']*, *logger=None*, *bbox_kwargs={'classwise': False, 'iou_thrs': None, 'metric_items': None, 'proposal_nums': (100, 300, 1000)}*, *track_kwargs={'ignore_by_classes': False, 'ignore_iof_thr': 0.5, 'iou_thr': 0.5, 'nproc': 4}*)

Evaluation in COCO protocol and CLEAR MOT metric (e.g. MOTA, IDF1).

Parameters

- **results** (*dict*) – Testing results of the dataset.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'bbox', 'segm', 'track'.
- **logger** (*logging.Logger | str | None*) – Logger used for printing related information during evaluation. Default: None.
- **bbox_kwargs** (*dict*) – Configuration for COCO styple evaluation.
- **track_kwargs** (*dict*) – Configuration for CLEAR MOT evaluation.

Returns COCO style and CLEAR MOT evaluation metric.

Return type dict[str, float]

get_ann_info(*img_info*)

Get COCO annotations by the information of image.

Parameters **img_info** (*int*) – Information of image.

Returns Annotation information of *img_info*.

Return type dict

key_img_sampling(*img_ids*, *interval=1*)

Sampling key images.

load_annotations(*ann_file*)

Load annotations from COCO/COCOVID style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation information from COCO/COCOVID api.

Return type list[dict]

load_video_anns(*ann_file*)
Load annotations from COCOVID style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation information from COCOVID api.

Return type list[dict]

prepare_data(*idx*)
Get data and annotations after pipeline.

Parameters **idx** (*int*) – Index of data.

Returns Data and annotations after pipeline with new keys introduced by pipeline.

Return type dict

prepare_results(*img_info*)
Prepare results for image (e.g. the annotation information, ...).

prepare_test_img(*idx*)
Get testing data after pipeline.

Parameters **idx** (*int*) – Index of data.

Returns Testing data after pipeline with new keys intorduced by pipeline.

Return type dict

prepare_train_img(*idx*)
Get training data and annotations after pipeline.

Parameters **idx** (*int*) – Index of data.

Returns Training data and annotations after pipeline with new keys introduced by pipeline.

Return type dict

ref_img_sampling(*img_info*, *frame_range*, *stride=1*, *num_ref_imgs=1*, *filter_key_img=True*, *method='uniform'*, *return_key_img=True*)
Sampling reference frames in the same video for key frame.

Parameters

- **img_info** (*dict*) – The information of key frame.
- **frame_range** (*List(int) | int*) – The sampling range of reference frames in the same video for key frame.
- **stride** (*int*) – The sampling frame stride when sampling reference images. Default: 1.
- **num_ref_imgs** (*int*) – The number of sampled reference images. Default: 1.
- **filter_key_img** (*bool*) – If False, the key image will be in the sampling reference candidates, otherwise, it is exclude. Default: True.
- **method** (*str*) – The sampling method. Options are 'uniform', 'bilateral_uniform', 'test_with_adaptive_stride', 'test_with_fix_stride'. 'uniform' denotes reference images are randomly sampled from the nearby frames of key frame. 'bilateral_uniform' denotes reference images are randomly sampled from the two sides of the nearby frames of key frame. 'test_with_adaptive_stride' is only used in testing, and denotes the sampling frame stride is equal to (video length / the number of reference images). test_with_fix_stride is only used in testing with sampling frame stride equalling to *stride*. Default: 'uniform'.

- **return_key_img** (*bool*) – If True, the information of key frame is returned, otherwise, not returned. Default: True.

Returns *img_info* and the reference images information or only the reference images information.

Return type list(dict)

class mmtrack.datasets.**DanceTrackDataset**(*visibility_thr=- 1*, *interpolate_tracks_cfg=None*, *detection_file=None*, **args*, ***kwargs*)

Dataset for DanceTrack: https://github.com/DanceTrack/DanceTrack.

Most content is inherited from MOTChallengeDataset.

get_benchmark_and_eval_split()

Get benchmark and dataset split to evaluate.

Get benchmark from upeper/lower-case image prefix and the dataset split to evaluate.

Returns The first string denotes the type of dataset. The second string denots the split of the dataset to eval.

Return type tuple(string)

class mmtrack.datasets.**GOT10kDataset**(**args*, ***kwargs*)

GOT10k Dataset of single object tracking.

The dataset can both support training and testing mode.

format_results(*results*, *resfile_path=None*, *logger=None*)

Format the results to txts (standard format for GOT10k Challenge).

Parameters

- **results** (*dict(list[ndarray])*) – Testing results of the dataset.
- **resfile_path** (*str*) – Path to save the formatted results. Defaults to None.
- **logger** (*logging.Logger | str | None, optional*) – defaults to None.

get_visibility_from_video(*video_ind*)

Get the visible information of instance in a video.

load_data_infos(*split='train'*)

Load dataset information.

Parameters **split** (*str, optional*) – the split of dataset. Defaults to 'train'.

Returns

the length of the list is the number of videos. The

inner dict is in the following format:

{ 'video_path': the video path 'ann_path': the annotation path 'start_frame_id': the starting frame number contained

in the image name

'end_frame_id': the ending frame number contained in the image name

'framename_template': the template of image name

}

Return type list[dict]

prepare_test_data(*video_ind*, *frame_ind*)
Get testing data of one frame. We parse one video, get one frame from it and pass the frame information to the pipeline.

Parameters

- **video_ind** (*int*) – video index
- **frame_ind** (*int*) – frame index

Returns testing data of one frame.

Return type dict

class mmtrack.datasets.**ImagenetVIDDataset**(**args*, ***kwargs*)
ImageNet VID dataset for video object detection.

load_annotations(*ann_file*)
Load annotations from COCO/COCOVID style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation information from COCO/COCOVID api.

Return type list[dict]

load_image_anns(*ann_file*)
Load annotations from COCO style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation information from COCO api.

Return type list[dict]

load_video_anns(*ann_file*)
Load annotations from COCOVID style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation information from COCOVID api.

Return type list[dict]

class mmtrack.datasets.**LaSOTDataset**(**args*, ***kwargs*)
LaSOT dataset of single object tracking.

The dataset can both support training and testing mode.

get_visibility_from_video(*video_ind*)
Get the visible information of instance in a video.

load_data_infos(*split='test'*)
Load dataset information.

Parameters **split** (*str, optional*) – Dataset split. Defaults to 'test'.

Returns

The length of the list is the number of videos. The

inner dict is in the following format:

{ 'video_path': the video path 'ann_path': the annotation path 'start_frame_id': the starting frame number contained

in the image name

'end_frame_id': the ending frame number contained in the image name

'framename_template': the template of image name

}

Return type list[dict]

class mmtrack.datasets.**MOTChallengeDataset**(*visibility_thr=-1*, *interpolate_tracks_cfg=None*, *detection_file=None*, **args*, ***kwargs*)

Dataset for MOTChallenge.

Parameters

- **visibility_thr** (*float, optional*) – The minimum visibility for the objects during training. Default to -1.
- **interpolate_tracks_cfg** (*dict, optional*) – If not None, Interpolate tracks linearly to make tracks more complete. Defaults to None. - min_num_frames (int, optional): The minimum length of a track

  that will be interpolated. Defaults to 5.

  - **max_num_frames (int, optional): The maximum disconnected length** in a track. Defaults to 20.

- **detection_file** (*str, optional*) – The path of the public detection file. Default to None.

evaluate(*results*, *metric='track'*, *logger=None*, *resfile_path=None*, *bbox_iou_thr=0.5*, *track_iou_thr=0.5*)

Evaluation in MOT Challenge.

Parameters

- **results** (*list[list | tuple]*) – Testing results of the dataset.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'bbox', 'track'. Defaults to 'track'.
- **logger** (*logging.Logger | str | None*) – Logger used for printing related information during evaluation. Default: None.
- **resfile_path** (*str, optional*) – Path to save the formatted results. Defaults to None.
- **bbox_iou_thr** (*float, optional*) – IoU threshold for detection evaluation. Defaults to 0.5.
- **track_iou_thr** (*float, optional*) – IoU threshold for tracking evaluation.. Defaults to 0.5.

Returns MOTChallenge style evaluation metric.

Return type dict[str, float]

format_bbox_results(*results*, *infos*, *resfile*)

Format detection results.

format_results(*results*, *resfile_path=None*, *metrics=['track']*)

Format the results to txts (standard format for MOT Challenge).

Parameters

- **results** (*dict(list[ndarray])*) – Testing results of the dataset.

- **resfile_path** (*str, optional*) – Path to save the formatted results. Defaults to None.
- **metrics** (*list[str], optional*) – The results of the specific metrics will be formatted.. Defaults to ['track'].

Returns (resfile_path, resfiles, names, tmp_dir), resfile_path is the path to save the formatted results, resfiles is a dict containing the filepaths, names is a list containing the name of the videos, tmp_dir is the temporal directory created for saving files.

Return type tuple

format_track_results(*results*, *infos*, *resfile*)
Format tracking results.

get_benchmark_and_eval_split()
Get benchmark and dataset split to evaluate.

Get benchmark from upeper/lower-case image prefix and the dataset split to evaluate.

Returns The first string denotes the type of dataset. The second string denotes the split of the dataset to eval.

Return type tuple(string)

get_dataset_cfg_for_hota(*gt_folder*, *tracker_folder*, *seqmap*)
Get default configs for trackeval.datasets.MotChallenge2DBox.

Parameters

- **gt_folder** (*str*) – the name of the GT folder
- **tracker_folder** (*str*) – the name of the tracker folder
- **seqmap** (*str*) – the file that contains the sequence of video names

Returns Dataset Configs for MotChallenge2DBox.

load_detections(*detection_file=None*)
Load public detections.

prepare_results(*img_info*)
Prepare results for image (e.g. the annotation information, ...).

class mmtrack.datasets.**OTB100Dataset**(**args*, ***kwargs*)
OTB100 dataset of single object tracking.

The dataset is only used to test.

get_bboxes_from_video(*video_ind*)
Get bboxes annotation about the instance in a video.

Parameters **video_ind** (*int*) – video index

Returns

in [N, 4] shape. The N is the bbox number and the bbox is in (x, y, w, h) format.

Return type ndarray

load_data_infos(*split='test'*)
Load dataset information.

Parameters **split** (*str, optional*) – Dataset split. Defaults to 'test'.

Returns

The length of the list is the number of videos. The

inner dict is in the following format:

{ 'video_path': the video path 'ann_path': the annotation path 'start_frame_id': the starting frame number contained

in the image name

'end_frame_id': the ending frame number contained in the image name

'framename_template': the template of image name 'init_skip_num': (optional) the number of skipped

frames when initializing tracker

}

Return type list[dict]

class mmtrack.datasets.**RandomSampleConcatDataset**(*dataset_cfgs*, *dataset_sampling_weights=None*)

A wrapper of concatenated dataset. Support randomly sampling one dataset from concatenated datasets and then getting samples from the sampled dataset.

Parameters

- **dataset_cfgs** (*list[dict]*) – The list contains all configs of concatenated datasets.
- **dataset_sampling_weights** (*list[float]*) – The list contains the sampling weights of each dataset.

class mmtrack.datasets.**ReIDDataset**(*pipeline*, *triplet_sampler=None*, **args*, ***kwargs*)

Dataset for ReID Dataset.

Parameters

- **pipeline** (*list*) – a list of dict, where each element represents a operation defined in *mmtrack.datasets.pipelines*
- **triplet_sampler** (*dict*) – The sampler for hard mining triplet loss.

evaluate(*results*, *metric='mAP'*, *metric_options=None*, *logger=None*)

Evaluate the ReID dataset.

Parameters

- **results** (*list*) – Testing results of the dataset.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Default value is *mAP*.
- **metric_options** – (dict, optional): Options for calculating metrics. Allowed keys are 'rank_list' and 'max_rank'. Defaults to None.
- **logger** (*logging.Logger | str, optional*) – Logger used for printing related information during evaluation. Defaults to None.

Returns evaluation results

Return type dict

load_annotations()

Load annotations from ImageNet style annotation file.

Returns Annotation information from ReID api.

Return type list[dict]

prepare_data(*idx*)
Prepare results for image (e.g. the annotation information, ...).

triplet_sampling(*pos_pid*, *num_ids=8*, *ins_per_id=4*)
Triplet sampler for hard mining triplet loss. First, for one pos_pid, random sample ins_per_id images with same person id.

Then, random sample num_ids - 1 negative ids. Finally, random sample ins_per_id images for each negative id.

Parameters

- **pos_pid** (*ndarray*) – The person id of the anchor.
- **num_ids** (*int*) – The number of person ids.
- **ins_per_id** (*int*) – The number of image for each person.

Returns Annotation information of num_ids X ins_per_id images.

Return type List

class mmtrack.datasets.**SOTCocoDataset**(*ann_file*, **args*, ***kwargs*)
Coco dataset of single object tracking.

The dataset only support training mode.

get_bboxes_from_video(*video_ind*)
Get bbox annotation about the instance in an image.

Parameters **video_ind** (*int*) – video index. Each video_ind denotes an instance.

Returns in [1, 4] shape. The bbox is in (x, y, w, h) format.

Return type ndarray

get_img_infos_from_video(*video_ind*)
Get all frame paths in a video.

Parameters **video_ind** (*int*) – video index. Each video_ind denotes an instance.

Returns all image paths

Return type list[str]

get_len_per_video(*video_ind*)
Get the number of frames in a video.

load_data_infos(*split='train'*)
Load dataset information. Each instance is viewed as a video.

Parameters **split** (*str, optional*) – The split of dataset. Defaults to 'train'.

Returns

The length of the list is the number of valid object annotations. The elemment in the list is annotation ID in coco API.

Return type list[int]

class mmtrack.datasets.**SOTImageNetVIDDataset**(*ann_file*, **args*, ***kwargs*)
ImageNet VID dataset of single object tracking.

The dataset only support training mode.

get_ann_infos_from_video(*video_ind*)
Get annotation information in a video. Note: We overload this function for speed up loading video information.

Parameters **video_ind** (*int*) – video index. Each video_ind denotes an instance.

Returns

{'bboxes': ndarray in (N, 4) shape, 'bboxes_isvalid': ndarray, 'visible':ndarray}. The bbox is in (x1, y1, x2, y2) format.

Return type dict

get_bboxes_from_video(*video_ind*)
Get bbox annotation about the instance in a video. Considering *get_bboxes_from_video* in *SOTBaseDataset* is not compatible with *SOTImageNetVIDDataset*, we oveload this function though it's not called by *self.get_ann_infos_from_video*.

Parameters **video_ind** (*int*) – video index. Each video_ind denotes an instance.

Returns in [N, 4] shape. The bbox is in (x, y, w, h) format.

Return type ndarray

get_img_infos_from_video(*video_ind*)
Get image information in a video.

Parameters **video_ind** (*int*) – video index

Returns {'filename': list[str], 'frame_ids':ndarray, 'video_id':int}

Return type dict

get_len_per_video(*video_ind*)
Get the number of frames in a video.

get_visibility_from_video(*video_ind*)
Get the visible information in a video.

Considering *get_visibility_from_video* in *SOTBaseDataset* is not compatible with *SOTImageNetVIDDataset*, we oveload this function though it's not called by *self.get_ann_infos_from_video*.

load_data_infos(*split='train'*)
Load dataset information.

Parameters **split** (*str, optional*) – The split of dataset. Defaults to 'train'.

Returns

The length of the list is the number of instances. The elemment in the list is instance ID in coco API.

Return type list[int]

class mmtrack.datasets.**SOTTestDataset**(**args*, ***kwargs*)
Dataset for the testing of single object tracking.

The dataset doesn't support training mode.

evaluate(*results*, *metric=['track']*, *logger=None*)
Evaluation in OPE protocol.

Parameters

- **results** (*dict*) – Testing results of the dataset.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'track'.

- **logger** (*logging.Logger | str | None*) – Logger used for printing related information during evaluation. Default: None.

Returns OPE style evaluation metric (i.e. success, norm precision and precision).

Return type dict[str, float]

class mmtrack.datasets.**SOTTrainDataset**(**args*, ***kwargs*)

Dataset for the training of single object tracking.

The dataset doesn't support testing mode.

get_snippet_of_instance(*idx*)

Get a snippet of an instance in a video.

Parameters **idx** (*int*) – Index of data.

Returns (snippet, image_id, instance_id), snippet is a list containing the successive image ids where the instance appears, image_id is a random sampled image id from the snippet.

Return type tuple

load_video_anns(*ann_file*)

Load annotations from COCOVID style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation information from COCOVID api.

Return type list[dict]

prepare_results(*img_id*, *instance_id*, *is_positive_pair*)

Get training data and annotations.

Parameters

- **img_id** (*int*) – The id of image.
- **instance_id** (*int*) – The id of instance.
- **is_positive_pair** (*bool*) – denoting positive or negative sample pair.

Returns The information of training image and annotation.

Return type dict

prepare_train_img(*idx*)

Get training data and annotations after pipeline.

Parameters **idx** (*int*) – Index of data.

Returns Training data and annotation after pipeline with new keys introduced by pipeline.

Return type dict

ref_img_sampling(*snippet*, *image_id*, *instance_id*, *frame_range=5*, *pos_prob=0.8*, *filter_key_img=False*, *return_key_img=True*, ***kwargs*)

Get a search image for an instance in an exemplar image.

If sampling a positive search image, the positive search image is randomly sampled from the exemplar image, where the sampled range is decided by *frame_range*. If sampling a negative search image, the negative search image and negative instance are randomly sampled from the entire dataset.

Parameters

- **snippet** (*list[int]*) – The successive image ids where the instance appears.
- **image_id** (*int*) – The id of exemplar image where the instance appears.

- **instance_id** (*int*) – The id of the instance.
- **frame_range** (*List(int) | int*) – The frame range of sampling a positive search image for the exemplar image. Default: 5.
- **pos_prob** (*float*) – The probability of sampling a positive search image. Default: 0.8.
- **filter_key_img** (*bool*) – If False, the exemplar image will be in the sampling candidates, otherwise, it is exclude. Default: False.
- **return_key_img** (*bool*) – If True, the *image_id* and *instance_id* are returned, otherwise, not returned. Default: True.

Returns (image_ids, instance_ids, is_positive_pair), image_ids is a list that must contain search image id and may contain *image_id*, instance_ids is a list that must contain search instance id and may contain *instance_id*, is_positive_pair is a bool denoting positive or negative sample pair.

Return type tuple

class mmtrack.datasets.**TaoDataset**(**args*, ***kwargs*)

Dataset for TAO.

evaluate(*results*, *metric=['bbox', 'track']*, *logger=None*, *resfile_path=None*)

Evaluation in COCO protocol and CLEAR MOT metric (e.g. MOTA, IDF1).

Parameters

- **results** (*dict*) – Testing results of the dataset.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'bbox', 'segm', 'track'.
- **logger** (*logging.Logger | str | None*) – Logger used for printing related information during evaluation. Default: None.
- **bbox_kwargs** (*dict*) – Configuration for COCO styple evaluation.
- **track_kwargs** (*dict*) – Configuration for CLEAR MOT evaluation.

Returns COCO style and CLEAR MOT evaluation metric.

Return type dict[str, float]

format_results(*results*, *resfile_path=None*)

Format the results to json (standard format for TAO evaluation).

Parameters

- **results** (*list[ndarray]*) – Testing results of the dataset.
- **resfile_path** (*str, optional*) – Path to save the formatted results. Defaults to None.

Returns (result_files, tmp_dir), result_files is a dict containing the json filepaths, tmp_dir is the temporal directory created for saving json files when resfile_path is not specified.

Return type tuple

load_annotations(*ann_file*)

Load annotation from annotation file.

load_lvis_anns(*ann_file*)

Load annotation from COCO style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation info from COCO api.

Return type list[dict]

load_tao_anns(*ann_file*)
Load annotation from COCOVID style annotation file.

Parameters **ann_file** (*str*) – Path of annotation file.

Returns Annotation info from COCOVID api.

Return type list[dict]

class mmtrack.datasets.**TrackingNetDataset**(*chunks_list=['all']*, **args*, ***kwargs*)
TrackingNet dataset of single object tracking.

The dataset can both support training and testing mode.

format_results(*results*, *resfile_path=None*, *logger=None*)
Format the results to txts (standard format for TrackingNet Challenge).

Parameters

- **results** (*dict(list[ndarray])*) – Testing results of the dataset.
- **resfile_path** (*str*) – Path to save the formatted results. Defaults to None.
- **logger** (*logging.Logger | str | None, optional*) – defaults to None.

load_data_infos(*split='train'*)
Load dataset information.

Parameters **split** (*str, optional*) – the split of dataset. Defaults to 'train'.

Returns

the length of the list is the number of videos. The

inner dict is in the following format:

{ 'video_path': the video path 'ann_path': the annotation path 'start_frame_id': the starting frame ID number

contained in the image name

'end_frame_id': the ending frame ID number contained in the image name

'framename_template': the template of image name

}

Return type list[dict]

prepare_test_data(*video_ind*, *frame_ind*)
Get testing data of one frame. We parse one video, get one frame from it and pass the frame information to the pipeline.

Parameters

- **video_ind** (*int*) – video index
- **frame_ind** (*int*) – frame index

Returns testing data of one frame.

Return type dict

class mmtrack.datasets.**UAV123Dataset**(**args*, ***kwargs*)

UAV123 dataset of single object tracking.

The dataset is only used to test.

load_data_infos(*split='test'*)

Load dataset information.

Parameters **split** (*str, optional*) – Dataset split. Defaults to 'test'.

Returns

The length of the list is the number of videos. The

inner dict is in the following format:

{ 'video_path': the video path 'ann_path': the annotation path 'start_frame_id': the starting frame number contained

in the image name

'end_frame_id': the ending frame number contained in the image name

'framename_template': the template of image name

}

Return type list[dict]

class mmtrack.datasets.**VOTDataset**(*dataset_type='vot2018'*, **args*, ***kwargs*)

VOT dataset of single object tracking.

The dataset is only used to test.

evaluate(*results*, *metric=['track']*, *logger=None*, *interval=None*)

Evaluation in VOT protocol.

Parameters

- **results** (*dict*) – Testing results of the dataset. The tracking bboxes are in (tl_x, tl_y, br_x, br_y) format.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'track'.
- **logger** (*logging.Logger | str | None*) – Logger used for printing related information during evaluation. Default: None.
- **interval** (*list*) – an specified interval in EAO curve used to calculate the EAO score. There are different settings in different VOT challenges.

Returns

Return type dict[str, float]

get_ann_infos_from_video(*video_ind*)

Get bboxes annotation about the instance in a video.

Parameters **video_ind** (*int*) – video index

Returns

in [N, 8] shape. The N is the bbox number and the bbox is in (x1, y1, x2, y2, x3, y3, x4, y4) format.

Return type ndarray

load_data_infos(*split='test'*)
Load dataset information.

Parameters **split** (*str, optional*) – Dataset split. Defaults to 'test'.

Returns

The length of the list is the number of videos. The

inner dict is in the following format:

{ 'video_path': the video path 'ann_path': the annotation path 'start_frame_id': the starting frame number contained

in the image name

'end_frame_id': the ending frame number contained in the image name

'framename_template': the template of image name

}

Return type list[dict]

class mmtrack.datasets.**YouTubeVISDataset**(*dataset_version*, **args*, ***kwargs*)
YouTube VIS dataset for video instance segmentation.

convert_back_to_vis_format()
Convert the annotation back to the format of YouTube-VIS. The main difference between the two is the format of 'annotation'. Before modification, it is recorded in the unit of images, and after modification, it is recorded in the unit of instances.This operation is to make it easier to use the official eval API.

Returns

A dict with 3 keys, categories, annotations and videos.

- categories (list[dict]): Each dict has 2 keys, id and name.
- videos (list[dict]): Each dict has 4 keys of video info, id, name, width and height.
- annotations (list[dict]): Each dict has 7 keys of video info, category_id, segmentations, bboxes, video_id, areas, id and iscrowd.

Return type dict

evaluate(*results*, *metric=['track_segm']*, *logger=None*)
Evaluation in COCO protocol.

Parameters

- **results** (*dict*) – Testing results of the dataset.
- **metric** (*str | list[str]*) – Metrics to be evaluated. Options are 'track_segm'.
- **logger** (*logging.Logger | str | None*) – Logger used for printing related information during evaluation. Default: None.

Returns COCO style evaluation metric.

Return type dict[str, float]

format_results(*results*, *resfile_path=None*, *metrics=['track_segm']*, *save_as_json=True*)
Format the results to a zip file (standard format for YouTube-VIS Challenge).

Parameters

- **results** (*dict(list[ndarray])*) – Testing results of the dataset.
- **resfile_path** (*str, optional*) – Path to save the formatted results. Defaults to None.
- **metrics** (*list[str], optional*) – The results of the specific metrics will be formatted. Defaults to ['track_segm'].
- **save_as_json** (*bool, optional*) – Whether to save the json results file. Defaults to True.

Returns (resfiles, tmp_dir), resfiles is the path of the result json file, tmp_dir is the temporal directory created for saving files.

Return type tuple

mmtrack.datasets.**build_dataloader**(*dataset*, *samples_per_gpu*, *workers_per_gpu*, *num_gpus=1*, *samples_per_epoch=None*, *dist=True*, *shuffle=True*, *seed=None*, *persistent_workers=False*, ***kwargs*)
Build PyTorch DataLoader.

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

Parameters

- **dataset** (*Dataset*) – A PyTorch dataset.
- **samples_per_gpu** (*int*) – Number of training samples on each GPU, i.e., batch size of each GPU.
- **workers_per_gpu** (*int*) – How many subprocesses to use for data loading for each GPU.
- **num_gpus** (*int*) – Number of GPUs. Only used in non-distributed training.
- **samples_per_epoch** (*int | None, Optional*) – The number of samples per epoch. If equal to -1, using all samples in the datasets per epoch. Otherwise, using the *samples_per_epoch* samples. Default: None.
- **dist** (*bool*) – Distributed training/test or not. Default: True.
- **shuffle** (*bool*) – Whether to shuffle the data at every epoch. Default: True.
- **seed** (*int, Optional*) – Seed to be used. Default: None.
- **persistent_workers** (*bool*) – If True, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. This argument is only valid when PyTorch>=1.7.0. Default: False.
- **kwargs** – any keyword argument to be used to initialize DataLoader

Returns A PyTorch dataloader.

Return type DataLoader

27.2 parsers

class mmtrack.datasets.parsers.CocoVID(**args: Any*, ***kwargs: Any*)

Inherit official COCO class in order to parse the annotations of bbox- related video tasks.

Parameters

- **annotation_file** (*str*) – location of annotation file. Defaults to None.
- **load_img_as_vid** (*bool*) – If True, convert image data to video data, which means each image is converted to a video. Defaults to False.

convert_img_to_vid(*dataset*)

Convert image data to video data.

createIndex()

Create index.

get_img_ids_from_ins_id(*insId*)

Get image ids from given instance id.

Parameters **insId** (*int*) – The given instance id.

Returns Image ids of given instance id.

Return type list[int]

get_img_ids_from_vid(*vidId*)

Get image ids from given video id.

Parameters **vidId** (*int*) – The given video id.

Returns Image ids of given video id.

Return type list[int]

get_ins_ids_from_vid(*vidId*)

Get instance ids from given video id.

Parameters **vidId** (*int*) – The given video id.

Returns Instance ids of given video id.

Return type list[int]

get_vid_ids(*vidIds=[]*)

Get video ids that satisfy given filter conditions.

Default return all video ids.

Parameters **vidIds** (*list[int]*) – The given video ids. Defaults to [].

Returns Video ids.

Return type list[int]

load_vids(*ids=[]*)

Get video information of given video ids.

Default return all videos information.

Parameters **ids** (*list[int]*) – The given video ids. Defaults to [].

Returns List of video information.

Return type list[dict]

27.3 pipelines

class mmtrack.datasets.pipelines.**CheckPadMaskValidity**(*stride*)

Check the validity of data. Generally, it's used in such case: The image padding masks generated in the image preprocess need to be downsampled, and then passed into Transformer model, like DETR. The computation in the subsequent Transformer model must make sure that the values of downsampled mask are not all zeros.

Parameters **stride** (*int*) – the max stride of feature map.

class mmtrack.datasets.pipelines.**ConcatSameTypeFrames**(*num_key_frames=1*)

Concat the frames of the same type. We divide all the frames into two types: 'key' frames and 'reference' frames.

The input list contains as least two dicts. We concat the first *num_key_frames* dicts to one dict, and the rest of dicts are concated to another dict.

In SOT field, 'key' denotes template image and 'reference' denotes search image.

Parameters **num_key_frames** (*int, optional*) – the number of key frames. Defaults to 1.

concat_one_mode_results(*results*)

Concatenate the results of the same mode.

class mmtrack.datasets.pipelines.**ConcatVideoReferences**

Concat video references.

If the input list contains at least two dicts, concat the input list of dict to one dict from 2-nd dict of the input list.

Note: the 'ConcatVideoReferences' class will be deprecated in the future, please use 'ConcatSameTypeFrames' instead.

class mmtrack.datasets.pipelines.**LoadDetections**

Load public detections from MOT benchmark.

Parameters **results** (*dict*) – Result dict from mmtrack.CocoVideoDataset.

class mmtrack.datasets.pipelines.**LoadMultiImagesFromFile**(**args*, ***kwargs*)

Load multi images from file.

Please refer to *mmdet.datasets.pipelines.loading.py:LoadImageFromFile* for detailed docstring.

class mmtrack.datasets.pipelines.**MatchInstances**(*skip_nomatch=True*)

Matching objects on a pair of images.

Parameters

- **skip_nomatch** (*bool, optional*) – Whether skip the pair of image
- **training when there are no matched objects. Default** (*during*) –
- **True.** (*to*) –

class mmtrack.datasets.pipelines.**PairSampling**(*frame_range=5*, *pos_prob=0.8*, *filter_template_img=False*)

Pair-style sampling. It's used in `SiameseRPN++

<https://arxiv.org/abs/1812.11703.>`_.

Parameters

- **frame_range** (*List(int) | int*) – the sampling range of search frames in the same video for template frame. Defaults to 5.
- **pos_prob** (*float, optional*) – the probility of sampling positive sample pairs. Defaults to 0.8.
- **filter_template_img** (*bool, optional*) – if False, the template image will be in the sampling search candidates, otherwise, it is exclude. Defaults to False.

prepare_data(*video_info*, *sampled_inds*, *is_positive_pairs=False*)
Prepare sampled training data according to the sampled index.

Parameters

- **video_info** (*dict*) – the video information. It contains the keys: ['bboxes','bboxes_isvalid','filename','frame_ids', 'video_id','visible'].
- **sampled_inds** (*list[int]*) – the sampled frame indexes.
- **is_positive_pairs** (*bool, optional*) – whether it's the positive pairs. Defaults to False.

Returns contains the information of sampled data.

Return type List[dict]

class mmtrack.datasets.pipelines.**ReIDFormatBundle**(**args*, ***kwargs*)
ReID formatting bundle.

It first concatenates common fields, then simplifies the pipeline of formatting common fields, including "img", and "gt_label". These fields are formatted as follows.

- img: (1) transpose, (2) to tensor, (3) to DataContainer (stack=True)
- gt_labels: (1) to tensor, (2) to DataContainer

reid_format_bundle(*results*)
Transform and format gt_label fields in results.

Parameters **results** (*dict*) – Result dict contains the data to convert.

Returns The result dict contains the data that is formatted with ReID bundle.

Return type dict

class mmtrack.datasets.pipelines.**SeqBboxJitter**(*scale_jitter_factor*, *center_jitter_factor*, *crop_size_factor*)
Bounding box jitter augmentation. The jittered bboxes are used for subsequent image cropping, like *SeqCropLikeStark*.

Parameters

- **scale_jitter_factor** (*list[int | float]*) – contains the factor of scale jitter.
- **center_jitter_factor** (*list[int | float]*) – contains the factor of center jitter.
- **crop_size_factor** (*list[int | float]*) – contains the ratio of crop size to bbox size.

class mmtrack.datasets.pipelines.**SeqBlurAug**(*prob=[0.0, 0.2]*)
Blur augmention for images.

Parameters **prob** (*list[float]*) – The probability to perform blur augmention for each image. Defaults to [0.0, 0.2].

class mmtrack.datasets.pipelines.**SeqBrightnessAug**(*jitter_range=0*)
Brightness augmention for images.

Parameters **jitter_range** (*float*) – The range of brightness jitter. Defaults to 0..

class mmtrack.datasets.pipelines.**SeqColorAug**(*prob=[1.0, 1.0]*, *rgb_var=[[- 0.55919361, 0.98062831, - 0.41940627], [1.72091413, 0.19879334, - 1.82968581], [4.64467907, 4.73710203, 4.88324118]]*)
Color augmention for images.

Parameters

- **prob** (*list[float]*) – The probability to perform color augmention for each image. Defaults to [1.0, 1.0].

- **rgb_var** (*list[list]*) – The values of color augmentaion. Defaults to [[-0.55919361, 0.98062831, -0.41940627], [1.72091413, 0.19879334, -1.82968581], [4.64467907, 4.73710203, 4.88324118]].

class mmtrack.datasets.pipelines.**SeqCropLikeSiamFC**(*context_amount=0.5*, *exemplar_size=127*, *crop_size=511*)

Crop images as SiamFC did.

The way of cropping an image is proposed in "Fully-Convolutional Siamese Networks for Object Tracking." SiamFC.

Parameters

- **context_amount** (*float*) – The context amount around a bounding box. Defaults to 0.5.
- **exemplar_size** (*int*) – Exemplar size. Defaults to 127.
- **crop_size** (*int*) – Crop size. Defaults to 511.

crop_like_SiamFC(*image*, *bbox*, *context_amount=0.5*, *exemplar_size=127*, *crop_size=511*)

Crop an image as SiamFC did.

Parameters

- **image** (*ndarray*) – of shape (H, W, 3).
- **bbox** (*ndarray*) – of shape (4,) in [x1, y1, x2, y2] format.
- **context_amount** (*float*) – The context amount around a bounding box. Defaults to 0.5.
- **exemplar_size** (*int*) – Exemplar size. Defaults to 127.
- **crop_size** (*int*) – Crop size. Defaults to 511.

Returns The cropped image of shape (crop_size, crop_size, 3).

Return type ndarray

generate_box(*image*, *gt_bbox*, *context_amount*, *exemplar_size*)

Generate box based on cropped image.

Parameters

- **image** (*ndarray*) – The cropped image of shape (self.crop_size, self.crop_size, 3).
- **gt_bbox** (*ndarray*) – of shape (4,) in [x1, y1, x2, y2] format.
- **context_amount** (*float*) – The context amount around a bounding box.
- **exemplar_size** (*int*) – Exemplar size. Defaults to 127.

Returns Generated box of shape (4,) in [x1, y1, x2, y2] format.

Return type ndarray

class mmtrack.datasets.pipelines.**SeqCropLikeStark**(*crop_size_factor*, *output_size*)

Crop images as Stark did.

The way of cropping an image is proposed in "Learning Spatio-Temporal Transformer for Visual Tracking." Stark.

Parameters

- **crop_size_factor** (*list[int | float]*) – contains the ratio of crop size to bbox size.

- **output_size** (*list[int | float]*) – contains the size of resized image (always square).

crop_like_stark(*img*, *bbox*, *crop_size_factor*, *output_size*)

Crop an image as Stark did.

Parameters

- **image** (*ndarray*) – of shape (H, W, 3).
- **bbox** (*ndarray*) – of shape (4,) in [x1, y1, x2, y2] format.
- **crop_size_factor** (*float*) – the ratio of crop size to bbox size
- **output_size** (*int*) – the size of resized image (always square).

Returns

the cropped image of shape (crop_size, crop_size, 3).

resize_factor (float): the ratio of original image scale to cropped image scale.

pdding_mask (ndarray): the padding mask caused by cropping.

Return type img_crop_padded (ndarray)

generate_box(*bbox_gt*, *bbox_cropped*, *resize_factor*, *output_size*, *normalize=False*)

Transform the box coordinates from the original image coordinates to the coordinates of the cropped image.

Parameters

- **bbox_gt** (*ndarray*) – of shape (4,) in [x1, y1, x2, y2] format.
- **bbox_cropped** (*ndarray*) – of shape (4,) in [x1, y1, x2, y2] format.
- **resize_factor** (*float*) – the ratio of original image scale to cropped image scale.
- **output_size** (*float*) – the size of output image.
- **normalize** (*bool*) – whether to normalize the output box. Default to True.

Returns generated box of shape (4,) in [x1, y1, x2, y2] format.

Return type ndarray

class mmtrack.datasets.pipelines.**SeqDefaultFormatBundle**(*ref_prefix='ref'*)

Sequence Default formatting bundle.

It simplifies the pipeline of formatting common fields, including “img”, “img_metas”, “proposals”, “gt_bboxes”, “gt_instance_ids”, “gt_match_indices”, “gt_bboxes_ignore”, “gt_labels”, “gt_masks”, “gt_semantic_seg” and ‘padding_mask’. These fields are formatted as follows.

- img: (1) transpose, (2) to tensor, (3) to DataContainer (stack=True)
- img_metas: (1) to DataContainer (cpu_only=True)
- proposals: (1) to tensor, (2) to DataContainer
- gt_bboxes: (1) to tensor, (2) to DataContainer
- gt_instance_ids: (1) to tensor, (2) to DataContainer
- gt_match_indices: (1) to tensor, (2) to DataContainer
- gt_bboxes_ignore: (1) to tensor, (2) to DataContainer
- gt_labels: (1) to tensor, (2) to DataContainer
- gt_masks: (1) to DataContainer (cpu_only=True)
- gt_semantic_seg: (1) unsqueeze dim-0 (2) to tensor, (3) to DataContainer (stack=True)
- padding_mask: (1) to tensor, (2) to DataContainer

Parameters **ref_prefix** (*str*) – The prefix of key added to the second dict of input list. Defaults to ‘ref’.

default_format_bundle(*results*)
Transform and format common fields in results.

Parameters **results** (*dict*) – Result dict contains the data to convert.

Returns The result dict contains the data that is formatted with default bundle.

Return type dict

class mmtrack.datasets.pipelines.**SeqGrayAug**(*prob=0.0*)
Gray augmention for images.

Parameters **prob** (*float*) – The probability to perform gray augmention. Defaults to 0..

class mmtrack.datasets.pipelines.**SeqLoadAnnotations**(*with_track=False*, **args*, ***kwargs*)
Sequence load annotations.

Please refer to *mmdet.datasets.pipelines.loading.py:LoadAnnotations* for detailed docstring.

Parameters **with_track** (*bool*) – If True, load instance ids of bboxes.

class mmtrack.datasets.pipelines.**SeqNormalize**(**args*, ***kwargs*)
Normalize images.

Please refer to *mmdet.datasets.pipelines.transforms.py:Normalize* for detailed docstring.

class mmtrack.datasets.pipelines.**SeqPad**(**args*, ***kwargs*)
Pad images.

Please refer to *mmdet.datasets.pipelines.transforms.py:Pad* for detailed docstring.

class mmtrack.datasets.pipelines.**SeqPhotoMetricDistortion**(*share_params=True*, *brightness_delta=32*, *contrast_range=(0.5, 1.5)*, *saturation_range=(0.5, 1.5)*, *hue_delta=18*)
Apply photometric distortion to image sequentially, every transformation is applied with a probability of 0.5. The position of random contrast is in second or second to last.

1. random brightness
2. random contrast (mode 0)
3. convert color from BGR to HSV
4. random saturation
5. random hue
6. convert color from HSV to BGR
7. random contrast (mode 1)
8. randomly swap channels

Parameters

- **brightness_delta** (*int*) – delta of brightness.
- **contrast_range** (*tuple*) – range of contrast.
- **saturation_range** (*tuple*) – range of saturation.
- **hue_delta** (*int*) – delta of hue.

get_params()
Generate parameters.

photo_metric_distortion(*results*, *params=None*)
Call function to perform photometric distortion on images.

Parameters

- **results** (*dict*) – Result dict from loading pipeline.

- **params** (*dict, optional*) – Pre-defined parameters. Default to None.

Returns Result dict with images distorted.

Return type dict

class mmtrack.datasets.pipelines.**SeqRandomCrop**(*crop_size*, *allow_negative_crop=False*, *share_params=False*, *bbox_clip_border=False*)

Sequentially random crop the images & bboxes & masks.

The absolute *crop_size* is sampled based on *crop_type* and *image_size*, then the cropped results are generated.

Parameters

- **crop_size** (*tuple*) – The relative ratio or absolute pixels of height and width.
- **allow_negative_crop** (*bool, optional*) – Whether to allow a crop that does not contain any bbox area. Default False.
- **share_params** (*bool, optional*) – Whether share the cropping parameters for the images.
- **bbox_clip_border** (*bool, optional*) – Whether clip the objects outside the border of the image. Defaults to True.

Note:

- **If the image is smaller than the absolute crop size, return the** original image.
- The keys for bboxes, labels and masks must be aligned. That is, *gt_bboxes* corresponds to *gt_labels* and *gt_masks*, and *gt_bboxes_ignore* corresponds to *gt_labels_ignore* and *gt_masks_ignore*.
- If the crop does not contain any gt-bbox region and *allow_negative_crop* is set to False, skip this image.

---

get_offsets(*img*)

Random generate the offsets for cropping.

random_crop(*results*, *offsets=None*)

Call function to randomly crop images, bounding boxes, masks, semantic segmentation maps.

Parameters

- **results** (*dict*) – Result dict from loading pipeline.
- **offsets** (*tuple, optional*) – Pre-defined offsets for cropping. Default to None.

Returns Randomly cropped results, 'img_shape' key in result dict is updated according to crop size.

Return type dict

class mmtrack.datasets.pipelines.**SeqRandomFlip**(*share_params*, **args*, ***kwargs*)

Randomly flip for images.

Please refer to *mmdet.datasets.pipelines.transforms.py:RandomFlip* for detailed docstring.

Parameters **share_params** (*bool*) – If True, share the flip parameters for all images. Defaults to True.

class mmtrack.datasets.pipelines.**SeqResize**(*share_params=True*, **args*, ***kwargs*)

Resize images.

Please refer to *mmdet.datasets.pipelines.transforms.py:Resize* for detailed docstring.

Parameters **share_params** (*bool*) – If True, share the resize parameters for all images. Defaults to True.

class mmtrack.datasets.pipelines.**SeqShiftScaleAug**(*target_size=[127, 255]*, *shift=[4, 64]*, *scale=[0.05, 0.18]*)

Shift and rescale images and bounding boxes.

Parameters

- **target_size** (*list[int]*) – list of int denoting exemplar size and search size, respectively. Defaults to [127, 255].
- **shift** (*list[int]*) – list of int denoting the max shift offset. Defaults to [4, 64].
- **scale** (*list[float]*) – list of float denoting the max rescale factor. Defaults to [0.05, 0.18].

class mmtrack.datasets.pipelines.**ToList**

Use list to warp each value of the input dict.

Parameters **results** (*dict*) – Result dict contains the data to convert.

Returns Updated result dict contains the data to convert.

Return type dict

class mmtrack.datasets.pipelines.**TridentSampling**(*num_search_frames=1*, *num_template_frames=2*, *max_frame_range=[200]*, *cls_pos_prob=0.5*, *train_cls_head=False*, *min_num_frames=20*)

Multitemplate-style sampling in a trident manner. It's firstly used in STARK.

Parameters

- **num_search_frames** (*int, optional*) – the number of search frames
- **num_template_frames** (*int, optional*) – the number of template frames
- **max_frame_range** (*list[int], optional*) – the max frame range of sampling a positive search image for the template image. Its length is equal to the number of extra templates, i.e., *num_template_frames*-1. Default length is 1.
- **cls_pos_prob** (*float, optional*) – the probility of sampling positive samples in classification training.
- **train_cls_head** (*bool, optional*) – whether to train classification head.
- **min_num_frames** (*int, optional*) – the min number of frames to be sampled.

prepare_cls_data(*video_info*, *video_info_another*, *sampled_inds*)

Prepare the sampled classification training data according to the sampled index.

Parameters

- **video_info** (*dict*) – the video information. It contains the keys: ['bboxes','bboxes_isvalid','filename','frame_ids', 'video_id','visible'].
- **video_info_another** (*dict*) – the another video information. It's only used to get negative samples in classification train. It contains the keys: ['bboxes','bboxes_isvalid','filename', 'frame_ids','video_id','visible'].
- **sampled_inds** (*list[int]*) – the sampled frame indexes.

Returns contains the information of sampled data.

Return type List[dict]

prepare_data(*video_info*, *sampled_inds*, *with_label=False*)

Prepare sampled training data according to the sampled index.

Parameters

- **video_info** (*dict*) – the video information. It contains the keys: ['bboxes','bboxes_isvalid','filename','frame_ids', 'video_id','visible'].
- **sampled_inds** (*list[int]*) – the sampled frame indexes.
- **with_label** (*bool, optional*) – whether to recode labels in ann infos. Only set True in classification training. Defaults to False.

Returns contains the information of sampled data.

Return type List[dict]

random_sample_inds(*video_visibility*, *num_samples=1*, *frame_range=None*, *allow_invisible=False*, *force_invisible=False*)

Random sampling a specific number of samples from the specified frame range of the video. It also considers the visibility of each frame.

Parameters

- **video_visibility** (*ndarray*) – the visibility of each frame in the video.
- **num_samples** (*int, optional*) – the number of samples. Defaults to 1.
- **frame_range** (*list | None, optional*) – the frame range of sampling. Defaults to None.
- **allow_invisible** (*bool, optional*) – whether to allow to get invisible samples. Defaults to False.
- **force_invisible** (*bool, optional*) – whether to force to get invisible samples. Defaults to False.

Returns The sampled frame indexes.

Return type List

sampling_trident(*video_visibility*)

Sampling multiple template images and one search images in one video.

Parameters **video_visibility** (*ndarray*) – the visibility of each frame in the video.

Returns the indexes of template and search images.

Return type List

class mmtrack.datasets.pipelines.**VideoCollect**(*keys*, *meta_keys=None*, *default_meta_keys=('filename', 'ori_filename', 'ori_shape', 'img_shape', 'pad_shape', 'scale_factor', 'flip', 'flip_direction', 'img_norm_cfg', 'frame_id', 'is_video_data')*)

Collect data from the loader relevant to the specific task.

Parameters

- **keys** (*Sequence[str]*) – Keys of results to be collected in `data`.
- **meta_keys** (*Sequence[str]*) – Meta keys to be converted to `mmcv.DataContainer` and collected in `data[img_metas]`. Defaults to None.
- **default_meta_keys** (*tuple*) – Default meta keys. Defaults to ('filename', 'ori_filename', 'ori_shape', 'img_shape', 'pad_shape', 'scale_factor', 'flip', 'flip_direction', 'img_norm_cfg', 'frame_id', 'is_video_data').

27.4 samplers

class mmtrack.datasets.samplers.**DistributedQuotaSampler**(*dataset*, *samples_per_epoch*, *num_replicas=None*, *rank=None*, *replacement=False*, *seed=0*)

Sampler that gets fixed number of samples per epoch.

It is especially useful in conjunction with `torch.nn.parallel.DistributedDataParallel`. In such case, each process can pass a DistributedSampler instance as a DataLoader sampler, and load a subset of the original dataset that is exclusive to it.

Note: Dataset is assumed to be of constant size.

Parameters

- **dataset** – Dataset used for sampling.
- **samples_per_epoch** (*int*) – The number of samples per epoch.
- **num_replicas** (*optional*) – Number of processes participating in distributed training.
- **rank** (*optional*) – Rank of the current process within num_replicas.
- **replacement** (*bool*) – samples are drawn with replacement if `True`, Default: False.
- **seed** (*int, optional*) – random seed used to shuffle the sampler if `shuffle=True`. This number should be identical across all processes in the distributed group. Default: 0.

class mmtrack.datasets.samplers.**DistributedVideoSampler**(*dataset*, *num_replicas=None*, *rank=None*, *shuffle=False*)

Put videos to multi gpus during testing.

Parameters

- **dataset** (*Dataset*) – Test dataset must have *data_infos* attribute. Each data_info in *data_infos* records information of one frame or one video (in SOT Dataset). If not SOT Dataset, each video must have one data_info that includes *data_info['frame_id'] == 0*.
- **num_replicas** (*int*) – The number of gpus. Defaults to None.
- **rank** (*int*) – Gpu rank id. Defaults to None.
- **shuffle** (*bool*) – If True, shuffle the dataset. Defaults to False.

class mmtrack.datasets.samplers.**SOTVideoSampler**(*dataset*)

Only used for sot testing on single gpu.

Parameters **dataset** (*Dataset*) – Test dataset must have *num_frames_per_video* attribute. It records the frame number of each video.

CHAPTER

TWENTYEIGHT

MMTRACK.MODELS

28.1 mot

class mmtrack.models.mot.**BaseMultiObjectTracker**(*init_cfg=None*)

Base class for multiple object tracking.

aug_test(*imgs*, *img_metas*, ***kwargs*)

Test function with test time augmentation.

forward(*img*, *img_metas*, *return_loss=True*, ***kwargs*)

Calls either *forward_train()* or *forward_test()* depending on whether `return_loss` is `True`.

Note this setting will change the expected inputs. When `return_loss=True`, img and img_meta are single-nested (i.e. Tensor and List[dict]), and when `resturn_loss=False`, img and img_meta should be double nested (i.e. List[Tensor], List[List[dict]]), with the outer list indicating test time augmentations.

forward_test(*imgs*, *img_metas*, ***kwargs*)

Parameters

- **imgs** (*List[Tensor]*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains all images in the batch.
- **img_metas** (*List[List[dict]]*) – the outer list indicates test-time augs (multi-scale, flip, etc.) and the inner list indicates images in a batch.

abstract forward_train(*imgs*, *img_metas*, ***kwargs*)

Parameters

- **img** (*list[Tensor]*) – List of tensors of shape (1, C, H, W). Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – List of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys, see `mmdet.datasets.pipelines.Collect`.
- **kwargs** (*keyword arguments*) – Specific to concrete implementation.

freeze_module(*module*)

Freeze module during training.

show_result(*img*, *result*, *score_thr=0.0*, *thickness=1*, *font_scale=0.5*, *show=False*, *out_file=None*, *wait_time=0*, *backend='cv2'*, ***kwargs*)

Visualize tracking results.

Parameters

- **img** (*str | ndarray*) – Filename of loaded image.
- **result** (*dict*) – Tracking result. - The value of key 'track_bboxes' is list with length num_classes, and each element in list is ndarray with shape(n, 6) in [id, tl_x, tl_y, br_x, br_y, score] format. - The value of key 'det_bboxes' is list with length num_classes, and each element in list is ndarray with shape(n, 5) in [tl_x, tl_y, br_x, br_y, score] format.
- **thickness** (*int, optional*) – Thickness of lines. Defaults to 1.
- **font_scale** (*float, optional*) – Font scales of texts. Defaults to 0.5.
- **show** (*bool, optional*) – Whether show the visualizations on the fly. Defaults to False.
- **out_file** (*str | None, optional*) – Output filename. Defaults to None.
- **backend** (*str, optional*) – Backend to draw the bounding boxes, options are *cv2* and *plt*. Defaults to 'cv2'.

Returns Visualized image.

Return type ndarray

abstract simple_test(*img*, *img_metas*, ***kwargs*)
Test function with a single scale.

train_step(*data*, *optimizer*)
The iteration step during training.

This method defines an iteration step during training, except for the back propagation and optimizer updating, which are done in an optimizer hook. Note that in some complicated cases or models, the whole process including back propagation and optimizer updating is also defined in this method, such as GAN.

Parameters

- **data** (*dict*) – The output of dataloader.
- **optimizer** (`torch.optim.Optimizer` | dict) – The optimizer of runner is passed to `train_step()`. This argument is unused and reserved.

Returns

It should contain at least 3 keys: `loss`, `log_vars`, `num_samples`.

- `loss` is a tensor for back propagation, which can be a

weighted sum of multiple losses. - `log_vars` contains all the variables to be sent to the logger. - `num_samples` indicates the batch size (when the model is DDP, it means the batch size on each GPU), which is used for averaging the logs.

Return type dict

val_step(*data*, *optimizer*)
The iteration step during validation.

This method shares the same signature as *train_step()*, but used during val epochs. Note that the evaluation after training epochs is not implemented with this method, but an evaluation hook.

property with_detector
whether the framework has a detector.

Type bool

property with_motion
whether the framework has a motion model.

Type bool

property with_reid
whether the framework has a reid model.

Type bool

property with_track_head
whether the framework has a track_head.

Type bool

property with_tracker
whether the framework has a tracker.

Type bool

class mmtrack.models.mot.**ByteTrack**(*detector=None*, *tracker=None*, *motion=None*, *init_cfg=None*)
ByteTrack: Multi-Object Tracking by Associating Every Detection Box.

This multi object tracker is the implementation of ByteTrack.

Parameters

- **detector** (*dict*) – Configuration of detector. Defaults to None.
- **tracker** (*dict*) – Configuration of tracker. Defaults to None.
- **motion** (*dict*) – Configuration of motion. Defaults to None.
- **init_cfg** (*dict*) – Configuration of initialization. Defaults to None.

forward_train(**args*, ***kwargs*)
Forward function during training.

simple_test(*img*, *img_metas*, *rescale=False*, ***kwargs*)
Test without augmentations.

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.
- **rescale** (*bool, optional*) – If False, then returned bboxes and masks will fit the scale of img, otherwise, returned bboxes and masks will fit the scale of original image shape. Defaults to False.

Returns list(ndarray)]: The tracking results.

Return type dict[str

class mmtrack.models.mot.**DeepSORT**(*detector=None*, *reid=None*, *tracker=None*, *motion=None*, *pretrains=None*, *init_cfg=None*)
Simple online and realtime tracking with a deep association metric.

Details can be found at **`DeepSORT<https://arxiv.org/abs/1703.07402>`_**.

forward_train(**args*, ***kwargs*)
Forward function during training.

simple_test(*img*, *img_metas*, *rescale=False*, *public_bboxes=None*, ***kwargs*)
Test without augmentations.

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.
- **rescale** (*bool, optional*) – If False, then returned bboxes and masks will fit the scale of img, otherwise, returned bboxes and masks will fit the scale of original image shape. Defaults to False.
- **public_bboxes** (*list[Tensor], optional*) – Public bounding boxes from the benchmark. Defaults to None.

Returns list(ndarray)]: The tracking results.

Return type dict[str

class mmtrack.models.mot.**OCSORT**(*detector=None*, *tracker=None*, *motion=None*, *init_cfg=None*)
OCOSRT: Observation-Centric SORT: Rethinking SORT for Robust Multi-Object Tracking

This multi object tracker is the implementation of OC-SORT.

Parameters

- **detector** (*dict*) – Configuration of detector. Defaults to None.
- **tracker** (*dict*) – Configuration of tracker. Defaults to None.
- **motion** (*dict*) – Configuration of motion. Defaults to None.
- **init_cfg** (*dict*) – Configuration of initialization. Defaults to None.

forward_train(**args*, ***kwargs*)
Forward function during training.

simple_test(*img*, *img_metas*, *rescale=False*, ***kwargs*)
Test without augmentations.

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.
- **rescale** (*bool, optional*) – If False, then returned bboxes and masks will fit the scale of img, otherwise, returned bboxes and masks will fit the scale of original image shape. Defaults to False.

Returns list(ndarray)]: The tracking results.

Return type dict[str

class mmtrack.models.mot.**QDTrack**(*detector=None*, *track_head=None*, *tracker=None*, *freeze_detector=False*, **args*, ***kwargs*)
Quasi-Dense Similarity Learning for Multiple Object Tracking.

This multi object tracker is the implementation of QDTrack.

Parameters

- **detector** (*dict*) – Configuration of detector. Defaults to None.
- **track_head** (*dict*) – Configuration of track head. Defaults to None.
- **tracker** (*dict*) – Configuration of tracker. Defaults to None.
- **freeze_detector** (*bool*) – If True, freeze the detector weights. Defaults to False.

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_match_indices*, *ref_img*, *ref_img_metas*, *ref_gt_bboxes*, *ref_gt_labels*, *gt_bboxes_ignore=None*, *gt_masks=None*, *ref_gt_bboxes_ignore=None*, *ref_gt_masks=None*, ***kwargs*)

Forward function during training.

Args:

img (Tensor): of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.

img_metas (list[dict]): list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.

gt_bboxes (list[Tensor]): Ground truth bboxes of the image, each item has a shape (num_gts, 4).

gt_labels (list[Tensor]): Ground truth labels of all images. each has a shape (num_gts,).

gt_match_indices (list(Tensor)): Mapping from gt_instance_ids to ref_gt_instance_ids of the same tracklet in a pair of images.

ref_img (Tensor): of shape (N, C, H, W) encoding input reference images. Typically these should be mean centered and std scaled.

ref_img_metas (list[dict]): list of reference image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.

ref_gt_bboxes (list[Tensor]): Ground truth bboxes of the reference image, each item has a shape (num_gts, 4).

ref_gt_labels (list[Tensor]): Ground truth labels of all reference images, each has a shape (num_gts,).

gt_masks (list[Tensor]) [Masks for each bbox, has a shape] (num_gts, h , w).

gt_bboxes_ignore (list[Tensor], None): Ground truth bboxes to be ignored, each item has a shape (num_ignored_gts, 4).

ref_gt_bboxes_ignore (list[Tensor], None): Ground truth bboxes of reference images to be ignored, each item has a shape (num_ignored_gts, 4).

ref_gt_masks (list[Tensor]) [Masks for each reference bbox,] has a shape (num_gts, h , w).

Returns Tensor]: All losses.

Return type dict[str

simple_test(*img*, *img_metas*, *rescale=False*)

Test forward.

Args:

img (Tensor): of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.

img_metas (list[dict]): list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.

rescale (bool): whether to rescale the bboxes.

Returns Tensor]: Track results.

Return type dict[str

class mmtrack.models.mot.**Tracktor**(*detector=None*, *reid=None*, *tracker=None*, *motion=None*, *pretrains=None*, *init_cfg=None*)

Tracking without bells and whistles.

Details can be found at **`Tracktor<https://arxiv.org/abs/1903.05625>`_**.

forward_train(**args*, ***kwargs*)

Forward function during training.

simple_test(*img*, *img_metas*, *rescale=False*, *public_bboxes=None*, ***kwargs*)

Test without augmentations.

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.
- **rescale** (*bool, optional*) – If False, then returned bboxes and masks will fit the scale of img, otherwise, returned bboxes and masks will fit the scale of original image shape. Defaults to False.
- **public_bboxes** (*list[Tensor], optional*) – Public bounding boxes from the benchmark. Defaults to None.

Returns list(ndarray)]: The tracking results.

Return type dict[str

property with_cmc

whether the framework has a camera model compensation model.

Type bool

property with_linear_motion

whether the framework has a linear motion model.

Type bool

28.2 sot

class mmtrack.models.sot.**MixFormer**(*backbone*, *head=None*, *init_cfg=None*, *frozen_modules=None*, *train_cfg=None*, *test_cfg=None*)

MixFormer: End-to-End Tracking with Iterative Mixed Attention.

This single object tracker is the implementation of `MixFormer<https://arxiv.org/abs/2203.11082>`_.

forward_train(*imgs*, *img_metas*, *search_img*, *search_img_metas*, ***kwargs*)

forward of training.

Parameters

- **img** (*Tensor*) – template images of shape (N, num_templates, C, H, W) Typically, there are 2 template images, and H and W are both equal to 128.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'image_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **search_img** (*Tensor*) – of shape (N, 1, C, H, W) encoding input search images. 1 denotes there is only one search image for each exemplar image. Typically H and W are both equal to 320.
- **search_img_metas** (*list[list[dict]]*) – The second list only has one element. The first list contains search image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape' and 'img_norm_cfg'. For details on the values of there keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for template images with shape (N, 4) in [tl_x, tl_y, br_x, br_y] format.
- **padding_mask** (*Tensor*) – padding mask of template images. It's of shape (N, num_templates, H, W). Typically, there are 2 padding masks of tehmplate images, and H and W are both equal to that of template images.
- **search_gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for search images with shape (N, 5) in [0., tl_x, tl_y, br_x, br_y] format.
- **search_padding_mask** (*Tensor*) – padding mask of search images. Its of shape (N, 1, H, W). There are 1 padding masks of search image, and H and W are both equal to that of search image.
- **search_gt_labels** (*list[Tensor], optional*) – Ground truth labels for search images with shape (N, 2).

Returns a dictionary of loss components.

Return type dict[str, Tensor]

init(*img*, *bbox*)

Initialize the single object tracker in the first frame.

Parameters

- **img** – (Tensor): input image of shape (1, C, H, W).
- **bbox** (*list | Tensor*) – in [cx, cy, w, h] format.

track(*img*, *bbox*)

Track the box *bbox* of previous frame to current frame *img*

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W).
- **bbox** (*list | Tensor*) – The bbox in previous frame. The shape of the bbox is (4,) in [x, y, w, h] format.

update_template(*img*, *bbox*, *conf_score*)
Update the dynamic templates.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W).
- **bbox** (*list | ndarray*) – in [cx, cy, w, h] format.
- **conf_score** (*float*) – the confidence score of the predicted bbox.

class mmtrack.models.sot.**SiamRPN**(*backbone*, *neck=None*, *head=None*, *pretrains=None*, *init_cfg=None*, *frozen_modules=None*, *train_cfg=None*, *test_cfg=None*)
SiamRPN++: Evolution of Siamese Visual Tracking with Very Deep Networks.

This single object tracker is the implementation of SiamRPN++.

forward_search(*x_img*)
Extract the features of search images.

Parameters **x_img** (*Tensor*) – of shape (N, C, H, W) encoding input search images. Typically H and W equal to 255.

Returns Multi level feature map of search images.

Return type tuple(Tensor)

forward_template(*z_img*)
Extract the features of exemplar images.

Parameters **z_img** (*Tensor*) – of shape (N, C, H, W) encoding input exemplar images. Typically H and W equal to 127.

Returns Multi level feature map of exemplar images.

Return type tuple(Tensor)

forward_train(*img*, *img_metas*, *gt_bboxes*, *search_img*, *search_img_metas*, *search_gt_bboxes*, *is_positive_pairs*, ***kwargs*)

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input exemplar images. Typically H and W equal to 127.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each exemplar image with shape (1, 4) in [tl_x, tl_y, br_x, br_y] format.
- **search_img** (*Tensor*) – of shape (N, 1, C, H, W) encoding input search images. 1 denotes there is only one search image for each exemplar image. Typically H and W equal to 255.

- **search_img_metas** (*list[list[dict]]*) – The second list only has one element. The first list contains search image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **search_gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each search image with shape (1, 5) in [0.0, tl_x, tl_y, br_x, br_y] format.
- **is_positive_pairs** (*list[bool]*) – list of bool denoting whether each exemplar image and corresponding search image is positive pair.

Returns a dictionary of loss components.

Return type dict[str, Tensor]

get_cropped_img(*img*, *center_xy*, *target_size*, *crop_size*, *avg_channel*)

Crop image.

Only used during testing.

This function mainly contains two steps: 1. Crop *img* based on center *center_xy* and size *crop_size*. If the cropped image is out of boundary of *img*, use *avg_channel* to pad. 2. Resize the cropped image to *target_size*.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding original input image.
- **center_xy** (*Tensor*) – of shape (2,) denoting the center point for cropping image.
- **target_size** (*int*) – The output size of cropped image.
- **crop_size** (*Tensor*) – The size for cropping image.
- **avg_channel** (*Tensor*) – of shape (3,) denoting the padding values.

Returns of shape (1, C, target_size, target_size) encoding the resized cropped image.

Return type Tensor

init(*img*, *bbox*)

Initialize the single object tracker in the first frame.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding original input image.
- **bbox** (*Tensor*) – The given instance bbox of first frame that need be tracked in the following frames. The shape of the box is (4,) with [cx, cy, w, h] format.

Returns z_feat is a tuple[Tensor] that contains the multi level feature maps of exemplar image, avg_channel is Tensor with shape (3,), and denotes the padding values.

Return type tuple(z_feat, avg_channel)

init_weights()

Initialize the weights of modules in single object tracker.

simple_test(*img*, *img_metas*, *gt_bboxes*, ***kwargs*)

Test without augmentation.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image.

- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **gt_bboxes** (*list[Tensor]*) – list of ground truth bboxes for each image with shape (1, 4) in [tl_x, tl_y, br_x, br_y] format or shape (1, 8) in [x1, y1, x2, y2, x3, y3, x4, y4].

Returns ndarray]: The tracking results.

Return type dict[str

simple_test_ope(*img*, *frame_id*, *gt_bboxes*)
Test using OPE test mode.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image.
- **frame_id** (*int*) – the id of current frame in the video.
- **gt_bboxes** (*list[Tensor]*) – list of ground truth bboxes for each image with shape (1, 4) in [tl_x, tl_y, br_x, br_y] format or shape (1, 8) in [x1, y1, x2, y2, x3, y3, x4, y4].

Returns

in [tl_x, tl_y, br_x, br_y] format. best_score (Tensor): the tracking bbox confidence in range [0,1],

and the score of initial frame is -1.

Return type bbox_pred (Tensor)

simple_test_vot(*img*, *frame_id*, *gt_bboxes*, *img_metas=None*)
Test using VOT test mode.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image.
- **frame_id** (*int*) – the id of current frame in the video.
- **gt_bboxes** (*list[Tensor]*) – list of ground truth bboxes for each image with shape (1, 4) in [tl_x, tl_y, br_x, br_y] format or shape (1, 8) in [x1, y1, x2, y2, x3, y3, x4, y4].
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.

Returns

in [tl_x, tl_y, br_x, br_y] format. best_score (Tensor): the tracking bbox confidence in range [0,1],

and the score of initial frame is -1.

Return type bbox_pred (Tensor)

track(*img*, *bbox*, *z_feat*, *avg_channel*)
Track the box *bbox* of previous frame to current frame *img*.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding original input image.
- **bbox** (*Tensor*) – The bbox in previous frame. The shape of the box is (4,) in [cx, cy, w, h] format.
- **z_feat** (*tuple[Tensor]*) – The multi level feature maps of exemplar image in the first frame.
- **avg_channel** (*Tensor*) – of shape (3,) denoting the padding values.

Returns best_score is a Tensor denoting the score of best_bbox, best_bbox is a Tensor of shape (4,) in [cx, cy, w, h] format, and denotes the best tracked bbox in current frame.

Return type tuple(best_score, best_bbox)

class mmtrack.models.sot.**Stark**(*backbone*, *neck=None*, *head=None*, *init_cfg=None*, *frozen_modules=None*, *train_cfg=None*, *test_cfg=None*)

STARK: Learning Spatio-Temporal Transformer for Visual Tracking.

This single object tracker is the implementation of STARk.

Parameters

- **backbone** (*dict*) – the configuration of backbone network.
- **neck** (*dict, optional*) – the configuration of neck network. Defaults to None.
- **head** (*dict, optional*) – the configuration of head network. Defaults to None.
- **init_cfg** (*dict, optional*) – the configuration of initialization. Defaults to None.
- **frozen_modules** (*str | list | tuple, optional*) – the names of frozen modules. Defaults to None.
- **train_cfg** (*dict, optional*) – the configuratioin of train. Defaults to None.
- **test_cfg** (*dict, optional*) – the configuration of test. Defaults to None.

extract_feat(*img*)

Extract the features of the input image.

Parameters **img** (*Tensor*) – image of shape (N, C, H, W).

Returns

the multi-level feature maps, and each of them is of shape (N, C, H // stride, W // stride).

Return type tuple(Tensor)

forward_train(*img*, *img_metas*, *search_img*, *search_img_metas*, *gt_bboxes*, *padding_mask*, *search_gt_bboxes*, *search_padding_mask*, *search_gt_labels=None*, ***kwargs*)

forward of training.

Parameters

- **img** (*Tensor*) – template images of shape (N, num_templates, C, H, W). Typically, there are 2 template images, and H and W are both equal to 128.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **search_img** (*Tensor*) – of shape (N, 1, C, H, W) encoding input search images. 1 denotes there is only one search image for each template image. Typically H and W are both equal to 320.

- **search_img_metas** (*list[list[dict]]*) – The second list only has one element. The first list contains search image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for template images with shape (N, 4) in [tl_x, tl_y, br_x, br_y] format.
- **padding_mask** (*Tensor*) – padding mask of template images. It's of shape (N, num_templates, H, W). Typically, there are 2 padding masks of template images, and H and W are both equal to that of template images.
- **search_gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for search images with shape (N, 5) in [0., tl_x, tl_y, br_x, br_y] format.
- **search_padding_mask** (*Tensor*) – padding mask of search images. Its of shape (N, 1, H, W). There are 1 padding masks of search image, and H and W are both equal to that of search image.
- **search_gt_labels** (*list[Tensor], optional*) – Ground truth labels for search images with shape (N, 2).

Returns a dictionary of loss components.

Return type dict[str, Tensor]

get_cropped_img(*img*, *target_bbox*, *search_area_factor*, *output_size*)

Crop Image Only used during testing This function mainly contains two steps: 1. Crop *img* based on target_bbox and search_area_factor. If the cropped image/mask is out of boundary of *img*, use 0 to pad. 2. Resize the cropped image/mask to *output_size*.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W)
- **target_bbox** (*list | ndarray*) – in [cx, cy, w, h] format
- **search_area_factor** (*float*) – Ratio of crop size to target size
- **output_size** (*float*) – the size of output cropped image (always square).

Returns

of shape (1, C, output_size, output_size) resize_factor (float): the ratio of original image scale to cropped

image scale.

pdding_mask (Tensor): the padding mask caused by cropping. It's of shape (1, output_size, output_size).

Return type img_crop_padded (Tensor)

init(*img*, *bbox*)

Initialize the single object tracker in the first frame.

Parameters

- **img** (*Tensor*) – input image of shape (1, C, H, W).
- **bbox** (*list | Tensor*) – in [cx, cy, w, h] format.

init_weights()
Initialize the weights of modules in single object tracker.

mapping_bbox_back(*pred_bboxes*, *prev_bbox*, *resize_factor*)
Mapping the *prediction bboxes* from resized cropped image to original image. The coordinate origins of them are both the top left corner.

Parameters

- **pred_bboxes** (*Tensor*) – the predicted bbox of shape (B, Nq, 4), in [tl_x, tl_y, br_x, br_y] format. The coordinates are based in the resized cropped image.
- **prev_bbox** (*Tensor*) – the previous bbox of shape (B, 4), in [cx, cy, w, h] format. The coordinates are based in the original image.
- **resize_factor** (*float*) – the ratio of original image scale to cropped image scale.

Returns in [tl_x, tl_y, br_x, br_y] format.

Return type (Tensor)

simple_test(*img*, *img_metas*, *gt_bboxes*, ***kwargs*)
Test without augmentation.

Parameters

- **img** (*Tensor*) – input image of shape (1, C, H, W).
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **gt_bboxes** (*list[Tensor]*) – list of ground truth bboxes for each image with shape (1, 4) in [tl_x, tl_y, br_x, br_y] format.

Returns ndarray): the tracking results.

Return type dict(str

track(*img*, *bbox*)
Track the box *bbox* of previous frame to current frame *img*.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W).
- **bbox** (*list | Tensor*) – The bbox in previous frame. The shape of the bbox is (4,) in [x, y, w, h] format.

Returns:

update_template(*img*, *bbox*, *conf_score*)
Update the dymanic templates.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W).
- **bbox** (*list | ndarray*) – in [cx, cy, w, h] format.
- **conf_score** (*float*) – the confidence score of the predicted bbox.

28.3 vid

class mmtrack.models.vid.**BaseVideoDetector**(*init_cfg*)

Base class for video object detector.

Parameters **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

aug_test(*imgs*, *img_metas*, ***kwargs*)

Test function with test time augmentation.

forward(*img*, *img_metas*, *ref_img=None*, *ref_img_metas=None*, *return_loss=True*, ***kwargs*)

Calls either *forward_train()* or *forward_test()* depending on whether `return_loss` is `True`.

Note this setting will change the expected inputs. When `return_loss=True`, img and img_meta are single-nested (i.e. Tensor and List[dict]), and when `resturn_loss=False`, img and img_meta should be double nested (i.e. List[Tensor], List[List[dict]]), with the outer list indicating test time augmentations.

forward_test(*imgs*, *img_metas*, *ref_img=None*, *ref_img_metas=None*, ***kwargs*)

Parameters

- **imgs** (*List[Tensor]*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains all images in the batch.
- **img_metas** (*List[List[dict]]*) – the outer list indicates test-time augs (multi-scale, flip, etc.) and the inner list indicates images in a batch.
- **ref_img** (*list[Tensor] | None*) – The list only contains one Tensor of shape (1, N, C, H, W) encoding input reference images. Typically these should be mean centered and std scaled. N denotes the number for reference images. There may be no reference images in some cases.
- **ref_img_metas** (*list[list[list[dict]]] | None*) – The first and second list only has one element. The third list contains image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*. There may be no reference images in some cases.

abstract forward_train(*imgs*, *img_metas*, *ref_img=None*, *ref_img_metas=None*, ***kwargs*)

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_img** (*Tensor*) – of shape (N, R, C, H, W) encoding input images. Typically these should be mean centered and std scaled. R denotes there is #R reference images for each input image.
- **ref_img_metas** (*list[list[dict]]*) – The first list only has one element. The second list contains reference image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.

freeze_module(*module*)
Freeze module during training.

show_result(*img*, *result*, *score_thr=0.3*, *bbox_color='green'*, *text_color='green'*, *thickness=1*, *font_scale=0.5*, *win_name=''*, *show=False*, *wait_time=0*, *out_file=None*)
Draw *result* over *img*.

Parameters

- **img** (*str or Tensor*) – The image to be displayed.
- **result** (*dict*) – The results to draw over *img* det_bboxes or (det_bboxes, det_masks). The value of key 'det_bboxes' is list with length num_classes, and each element in list is ndarray with shape(n, 5) in [tl_x, tl_y, br_x, br_y, score] format.
- **score_thr** (*float, optional*) – Minimum score of bboxes to be shown. Default: 0.3.
- **bbox_color** (str or tuple or Color) – Color of bbox lines.
- **text_color** (str or tuple or Color) – Color of texts.
- **thickness** (*int*) – Thickness of lines.
- **font_scale** (*float*) – Font scales of texts.
- **win_name** (*str*) – The window name.
- **wait_time** (*int*) – Value of waitKey param. Default: 0.
- **show** (*bool*) – Whether to show the image. Default: False.
- **out_file** (*str or None*) – The filename to write the image. Default: None.

Returns Only if not *show* or *out_file*

Return type img (Tensor)

train_step(*data*, *optimizer*)
The iteration step during training.

This method defines an iteration step during training, except for the back propagation and optimizer updating, which are done in an optimizer hook. Note that in some complicated cases or models, the whole process including back propagation and optimizer updating is also defined in this method, such as GAN.

Parameters

- **data** (*dict*) – The output of dataloader.
- **optimizer** (torch.optim.Optimizer | dict) – The optimizer of runner is passed to train_step(). This argument is unused and reserved.

Returns

It should contain at least 3 keys: loss, log_vars, num_samples.

- loss is a tensor for back propagation, which can be a weighted sum of multiple losses.
- log_vars contains all the variables to be sent to the

logger. - num_samples indicates the batch size (when the model is DDP, it means the batch size on each GPU), which is used for averaging the logs.

Return type dict

val_step(*data*, *optimizer*)
The iteration step during validation.

This method shares the same signature as *train_step()*, but used during val epochs. Note that the evaluation after training epochs is not implemented with this method, but an evaluation hook.

property with_aggregator
whether the framework has a aggregator

Type bool

property with_detector
whether the framework has a detector

Type bool

property with_motion
whether the framework has a motion model

Type bool

class mmtrack.models.vid.**DFF**(*detector*, *motion*, *pretrains=None*, *init_cfg=None*, *frozen_modules=None*, *train_cfg=None*, *test_cfg=None*)
Deep Feature Flow for Video Recognition.

This video object detector is the implementation of DFF.

aug_test(*imgs*, *img_metas*, ***kwargs*)
Test function with test time augmentation.

extract_feats(*img*, *img_metas*)
Extract features for *img* during testing.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.

Returns Multi level feature maps of *img*.

Return type list[Tensor]

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *ref_img*, *ref_img_metas*, *ref_gt_bboxes*, *ref_gt_labels*, *gt_instance_ids=None*, *gt_bboxes_ignore=None*, *gt_masks=None*, *proposals=None*, *ref_gt_instance_ids=None*, *ref_gt_bboxes_ignore=None*, *ref_gt_masks=None*, *ref_proposals=None*, ***kwargs*)

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.

- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box.
- **ref_img** (*Tensor*) – of shape (N, 1, C, H, W) encoding input images. Typically these should be mean centered and std scaled. 1 denotes there is only one reference image for each input image.
- **ref_img_metas** (*list[list[dict]]*) – The first list only has one element. The second list contains reference image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_gt_bboxes** (*list[Tensor]*) – The list only has one Tensor. The Tensor contains ground truth bboxes for each reference image with shape (num_all_ref_gts, 5) in [ref_img_id, tl_x, tl_y, br_x, br_y] format. The ref_img_id start from 0, and denotes the id of reference image for each key image.
- **ref_gt_labels** (*list[Tensor]*) – The list only has one Tensor. The Tensor contains class indices corresponding to each reference box with shape (num_all_ref_gts, 2) in [ref_img_id, class_indice].
- **gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bbox.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task.
- **proposals** (*None | Tensor*) – override rpn proposals with custom proposals. Use when *with_rpn* is False.
- **ref_gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bboxes of reference images.
- **ref_gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes of reference images can be ignored when computing the loss.
- **ref_gt_masks** (*None | Tensor*) – True segmentation masks for each box of reference image used if the architecture supports a segmentation task.
- **ref_proposals** (*None | Tensor*) – override rpn proposals with custom proposals of reference images. Use when *with_rpn* is False.

Returns a dictionary of loss components

Return type dict[str, Tensor]

simple_test(*img*, *img_metas*, *ref_img=None*, *ref_img_metas=None*, *proposals=None*, *rescale=False*)
Test without augmentation.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape',

'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.

- **ref_img** (*None*) – Not used in DFF. Only for unifying API interface.
- **ref_img_metas** (*None*) – Not used in DFF. Only for unifying API interface.
- **proposals** (*None | Tensor*) – Override rpn proposals with custom proposals. Use when *with_rpn* is False. Defaults to None.
- **rescale** (*bool*) – If False, then returned bboxes and masks will fit the scale of img, otherwise, returned bboxes and masks will fit the scale of original image shape. Defaults to False.

Returns list(ndarray)]: The detection results.

Return type dict[str

class mmtrack.models.vid.**FGFA**(*detector*, *motion*, *aggregator*, *pretrains=None*, *init_cfg=None*, *frozen_modules=None*, *train_cfg=None*, *test_cfg=None*)

Flow-Guided Feature Aggregation for Video Object Detection.

This video object detector is the implementation of FGFA.

aug_test(*imgs*, *img_metas*, ***kwargs*)

Test function with test time augmentation.

extract_feats(*img*, *img_metas*, *ref_img*, *ref_img_metas*)

Extract features for *img* during testing.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_img** (*Tensor | None*) – of shape (1, N, C, H, W) encoding input reference images. Typically these should be mean centered and std scaled. N denotes the number of reference images. There may be no reference images in some cases.
- **ref_img_metas** (*list[list[dict]] | None*) – The first list only has one element. The second list contains image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*. There may be no reference images in some cases.

Returns Multi level feature maps of *img*.

Return type list[Tensor]

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *ref_img*, *ref_img_metas*, *ref_gt_bboxes*, *ref_gt_labels*, *gt_instance_ids=None*, *gt_bboxes_ignore=None*, *gt_masks=None*, *proposals=None*, *ref_gt_instance_ids=None*, *ref_gt_bboxes_ignore=None*, *ref_gt_masks=None*, *ref_proposals=None*, ***kwargs*)

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box.
- **ref_img** (*Tensor*) – of shape (N, 2, C, H, W) encoding input images. Typically these should be mean centered and std scaled. 2 denotes there is two reference images for each input image.
- **ref_img_metas** (*list[list[dict]]*) – The first list only has one element. The second list contains reference image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_gt_bboxes** (*list[Tensor]*) – The list only has one Tensor. The Tensor contains ground truth bboxes for each reference image with shape (num_all_ref_gts, 5) in [ref_img_id, tl_x, tl_y, br_x, br_y] format. The ref_img_id start from 0, and denotes the id of reference image for each key image.
- **ref_gt_labels** (*list[Tensor]*) – The list only has one Tensor. The Tensor contains class indices corresponding to each reference box with shape (num_all_ref_gts, 2) in [ref_img_id, class_indice].
- **gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bbox.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task.
- **proposals** (*None | Tensor*) – override rpn proposals with custom proposals. Use when *with_rpn* is False.
- **ref_gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bboxes of reference images.
- **ref_gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes of reference images can be ignored when computing the loss.
- **ref_gt_masks** (*None | Tensor*) – True segmentation masks for each box of reference image used if the architecture supports a segmentation task.
- **ref_proposals** (*None | Tensor*) – override rpn proposals with custom proposals of reference images. Use when *with_rpn* is False.

Returns a dictionary of loss components

Return type dict[str, Tensor]

simple_test(*img*, *img_metas*, *ref_img=None*, *ref_img_metas=None*, *proposals=None*, *rescale=False*)
Test without augmentation.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_img** (*list[Tensor] | None*) – The list only contains one Tensor of shape (1, N, C, H, W) encoding input reference images. Typically these should be mean centered and std scaled. N denotes the number for reference images. There may be no reference images in some cases.
- **ref_img_metas** (*list[list[list[dict]]] | None*) – The first and second list only has one element. The third list contains image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*. There may be no reference images in some cases.
- **proposals** (*None | Tensor*) – Override rpn proposals with custom proposals. Use when *with_rpn* is False. Defaults to None.
- **rescale** (*bool*) – If False, then returned bboxes and masks will fit the scale of img, otherwise, returned bboxes and masks will fit the scale of original image shape. Defaults to False.

Returns list(ndarray)]: The detection results.

Return type dict[str

class mmtrack.models.vid.**SELSA**(*detector*, *pretrains=None*, *init_cfg=None*, *frozen_modules=None*, *train_cfg=None*, *test_cfg=None*)

Sequence Level Semantics Aggregation for Video Object Detection.

This video object detector is the implementation of SELSA.

aug_test(*imgs*, *img_metas*, ***kwargs*)

Test function with test time augmentation.

extract_feats(*img*, *img_metas*, *ref_img*, *ref_img_metas*)

Extract features for *img* during testing.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_img** (*Tensor | None*) – of shape (1, N, C, H, W) encoding input reference images. Typically these should be mean centered and std scaled. N denotes the number of reference images. There may be no reference images in some cases.
- **ref_img_metas** (*list[list[dict]] | None*) – The first list only has one element. The second list contains image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape',

'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*. There may be no reference images in some cases.

Returns

x is the multi level feature maps of *img*, ref_x is the multi level feature maps of *ref_img*.

Return type tuple(x, img_metas, ref_x, ref_img_metas)

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *ref_img*, *ref_img_metas*, *ref_gt_bboxes*, *ref_gt_labels*, *gt_instance_ids=None*, *gt_bboxes_ignore=None*, *gt_masks=None*, *proposals=None*, *ref_gt_instance_ids=None*, *ref_gt_bboxes_ignore=None*, *ref_gt_masks=None*, *ref_proposals=None*, ***kwargs*)

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box.
- **ref_img** (*Tensor*) – of shape (N, 2, C, H, W) encoding input images. Typically these should be mean centered and std scaled. 2 denotes there is two reference images for each input image.
- **ref_img_metas** (*list[list[dict]]*) – The first list only has one element. The second list contains reference image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_gt_bboxes** (*list[Tensor]*) – The list only has one Tensor. The Tensor contains ground truth bboxes for each reference image with shape (num_all_ref_gts, 5) in [ref_img_id, tl_x, tl_y, br_x, br_y] format. The ref_img_id start from 0, and denotes the id of reference image for each key image.
- **ref_gt_labels** (*list[Tensor]*) – The list only has one Tensor. The Tensor contains class indices corresponding to each reference box with shape (num_all_ref_gts, 2) in [ref_img_id, class_indice].
- **gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bbox.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task.
- **proposals** (*None | Tensor*) – override rpn proposals with custom proposals. Use when *with_rpn* is False.

- **ref_gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bboxes of reference images.
- **ref_gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes of reference images can be ignored when computing the loss.
- **ref_gt_masks** (*None | Tensor*) – True segmentation masks for each box of reference image used if the architecture supports a segmentation task.
- **ref_proposals** (*None | Tensor*) – override rpn proposals with custom proposals of reference images. Use when *with_rpn* is False.

Returns a dictionary of loss components

Return type dict[str, Tensor]

simple_test(*img*, *img_metas*, *ref_img=None*, *ref_img_metas=None*, *proposals=None*, *ref_proposals=None*, *rescale=False*)

Test without augmentation.

Parameters

- **img** (*Tensor*) – of shape (1, C, H, W) encoding input image. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **ref_img** (*list[Tensor] | None*) – The list only contains one Tensor of shape (1, N, C, H, W) encoding input reference images. Typically these should be mean centered and std scaled. N denotes the number for reference images. There may be no reference images in some cases.
- **ref_img_metas** (*list[list[list[dict]]] | None*) – The first and second list only has one element. The third list contains image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*. There may be no reference images in some cases.
- **proposals** (*None | Tensor*) – Override rpn proposals with custom proposals. Use when *with_rpn* is False. Defaults to None.
- **rescale** (*bool*) – If False, then returned bboxes and masks will fit the scale of img, otherwise, returned bboxes and masks will fit the scale of original image shape. Defaults to False.

Returns list(ndarray)]: The detection results.

Return type dict[str

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

class mmtrack.models.backbones.**ConvVisionTransformer**(*in_chans=3, act_layer=<class 'mmtrack.models.backbones.mixformer_backbone.QuickGELU'>, norm_layer=functools.partial(<class 'mmtrack.models.backbones.mixformer_backbone.LayerNormAutofp32'>, eps=1e-05), init='trunc_norm', num_stages=3, patch_size=[7, 3, 3], patch_stride=[4, 2, 2], patch_padding=[2, 1, 1], dim_embed=[64, 192, 384], num_heads=[1, 3, 6], depth=[1, 4, 16], mlp_channel_ratio=[4, 4, 4], attn_drop_rate=[0.0, 0.0, 0.0], drop_rate=[0.0, 0.0, 0.0], path_drop_probs=[0.0, 0.0, 0.1], qkv_bias=[True, True, True], qkv_proj_method=['dw_bn', 'dw_bn', 'dw_bn'], kernel_qkv=[3, 3, 3], padding_kv=[1, 1, 1], stride_kv=[2, 2, 2], padding_q=[1, 1, 1], stride_q=[1, 1, 1], norm_cfg={'requires_grad': False, 'type': 'BN'}*)

Vision Transformer with support for patch or hybrid CNN input stage.

This backbone refers to the implementation of CvT:.

Parameters

- **in_chans** (*int*) – number of input channels
- **act_layer** (*nn.Module*) – activate function used in FFN
- **norm_layer** (*nn.Module*) – normalization layer used in attention block
- **init** (*str*) – weight init method
- **num_stage** (*int*) – number of backbone stages
- **patch_size** (*List[int]*) – patch size of each stage
- **patch_stride** (*List[int]*) – patch stride of each stage
- **patch_padding** (*List[int]*) – patch padding of each stage
- **dim_embed** (*List[int]*) – embedding dimension of each stage
- **num_heads** (*List[int]*) – number of heads in multi-head
- **operation of each stage** (*attention*) –
- **depth** (*List[int]*) – number of attention blocks of each stage
- **mlp_channel_ratio** (*List[int]*) – hidden dim ratio of FFN of each stage
- **attn_drop_rate** (*List[float]*) – attn drop rate of each stage
- **drop_rate** (*List[float]*) – drop rate of each stage
- **path_drop_probs** (*List[float]*) – drop path of each stage

- **qkv_bias** (*List[bool]*) – qkv bias of each stage
- **qkv_proj_method** (*List[str]*) – qkv project method of each stage
- **kernel_qkv** (*List[int]*) – kernel size for qkv projection of each stage
- **padding_kv/q** (*List[int]*) – padding size for kv/q projection
- **each stage** (*of*) –
- **stride_kv/q** (*List[int]*) – stride for kv/q project of each stage
- **norm_cfg** (*dict*) – normalization layer config

forward(*template*, *online_template*, *search*)

Forward-pass method in train pipeline.

Parameters

- **template** (*online*) – template images of shape (B, C, H, W)
- **template** – online template images
- **shape** (*of*) –
- **search** (*Tensor*) – search images of shape (B, C, H, W)

forward_test(*search*)

Forward-pass method for search image in test pipeline. The model forwarding strategies are different between train and test. In test pipeline, we call `search()` method which only takes in search image when tracker is tracking current frame. This approach reduces computational overhead and thus increases tracking speed.

Parameters **search** (*Tensor*) – search images of shape (B, C, H, W)

set_online(*template*, *online_template*)

Forward-pass method for template image in test pipeline. The model forwarding strategies are different between train and test. In test pipeline, we call `set_online()` method which only takes in template images when tracker is initialized or is updating online template. This approach reduces computational overhead and thus increases tracking speed.

Parameters

- **template** (*online*) – template images of shape (B, C, H, W)
- **template** – online template images
- **shape** (*of*) –

class mmtrack.models.backbones.**SOTResNet**(*depth*, *unfreeze_backbone=True*, ***kwargs*)

ResNet backbone for SOT.

The main difference between ResNet in torch and the SOTResNet is the padding and dilation in the convs of SOTResNet. Please refer to SiamRPN++ for detailed analysis.

Parameters **depth** (*int*) – Depth of resnet, from {50, }.

make_res_layer(***kwargs*)

Pack all blocks in a stage into a `ResLayer`.

28.6 losses

class mmtrack.models.losses.**L2Loss**(*neg_pos_ub=- 1*, *pos_margin=- 1*, *neg_margin=- 1*, *hard_mining=False*, *reduction='mean'*, *loss_weight=1.0*)

L2 loss.

Parameters

- **reduction** (*str, optional*) – The method to reduce the loss. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – The weight of loss.

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*)

Forward function.

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning target of the prediction.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

static random_choice(*gallery*, *num*)

Random select some elements from the gallery.

It seems that Pytorch's implementation is slower than numpy so we use numpy to randperm the indices.

update_weight(*pred*, *target*, *weight*, *avg_factor*)

Update the weight according to targets.

class mmtrack.models.losses.**MultiPosCrossEntropyLoss**(*reduction='mean'*, *loss_weight=1.0*)

multi-positive targets cross entropy loss.

forward(*cls_score*, *label*, *weight=None*, *avg_factor=None*, *reduction_override=None*, ***kwargs*)

Forward function.

Parameters

- **cls_score** (*torch.Tensor*) – The classification score.
- **label** (*torch.Tensor*) – The assigned label of the prediction.
- **weight** (*torch.Tensor*) – The element-wise weight.
- **avg_factor** (*float*) – Average factor when computing the mean of losses.
- **reduction** (*str*) – Same as built-in losses of PyTorch.

Returns Calculated loss

Return type torch.Tensor

multi_pos_cross_entropy(*pred*, *label*, *weight=None*, *reduction='mean'*, *avg_factor=None*)

Parameters

- **pred** (*torch.Tensor*) – The prediction.

- **label** (*torch.Tensor*) – The assigned label of the prediction.
- **weight** (*torch.Tensor*) – The element-wise weight.
- **reduction** (*str*) – Same as built-in losses of PyTorch.
- **avg_factor** (*float*) – Average factor when computing the mean of losses.

Returns Calculated loss

Return type torch.Tensor

class mmtrack.models.losses.**TripletLoss**(*margin=0.3*, *loss_weight=1.0*, *hard_mining=True*)

Triplet loss with hard positive/negative mining.

Reference:

Hermans et al. In Defense of the Triplet Loss for Person Re-Identification. arXiv:1703.07737.

Imported from `<https://github.com/KaiyangZhou/deep-person-reid/blob/` master/torchreid/losses/hard_mine_triplet_loss.py>`_.

Parameters

- **margin** (*float, optional*) – Margin for triplet loss. Default to 0.3.
- **loss_weight** (*float, optional*) – Weight of the loss. Default to 1.0.

forward(*inputs*, *targets*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

hard_mining_triplet_loss_forward(*inputs*, *targets*)

Parameters

- **inputs** (*torch.Tensor*) – feature matrix with shape (batch_size, feat_dim).
- **targets** (*torch.LongTensor*) – ground truth labels with shape (num_classes).

28.7 motion

class mmtrack.models.motion.**CameraMotionCompensation**(*warp_mode='cv2.MOTION_EUCLIDEAN'*, *num_iters=50*, *stop_eps=0.001*)

Camera motion compensation.

Parameters

- **warp_mode** (*str*) – Warp mode in opencv.
- **num_iters** (*int*) – Number of the iterations.
- **stop_eps** (*float*) – Terminate threshold.

get_warp_matrix(*img*, *ref_img*)

Calculate warping matrix between two images.

track(*img*, *ref_img*, *tracks*, *num_samples*, *frame_id*)

Tracking forward.

warp_bboxes(*bboxes*, *warp_matrix*)
Warp bounding boxes according to the warping matrix.

class mmtrack.models.motion.**FlowNetSimple**(*img_scale_factor*, *out_indices=[2, 3, 4, 5, 6]*, *flow_scale_factor=5.0*, *flow_img_norm_std=[255.0, 255.0, 255.0]*, *flow_img_norm_mean=[0.411, 0.432, 0.45]*, *init_cfg=None*)

The simple version of FlowNet.

This FlowNetSimple is the implementation of FlowNetSimple.

Parameters

- **img_scale_factor** (*float*) – Used to upsample/downsample the image.
- **out_indices** (*list*) – The indices of outputting feature maps after each group of conv layers. Defaults to [2, 3, 4, 5, 6].
- **flow_scale_factor** (*float*) – Used to enlarge the values of flow. Defaults to 5.0.
- **flow_img_norm_std** (*list*) – Used to scale the values of image. Defaults to [255.0, 255.0, 255.0].
- **flow_img_norm_mean** (*list*) – Used to center the values of image. Defaults to [0.411, 0.432, 0.450].
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

crop_like(*input*, *target*)
Crop *input* as the size of *target*.

forward(*imgs*, *img_metas*)
Compute the flow of images pairs.

Parameters

- **imgs** (*Tensor*) – of shape (N, 6, H, W) encoding input images pairs. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.

Returns of shape (N, 2, H, W) encoding flow of images pairs.

Return type Tensor

prepare_imgs(*imgs*, *img_metas*)
Preprocess images pairs for computing flow.

Parameters

- **imgs** (*Tensor*) – of shape (N, 6, H, W) encoding input images pairs. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image information dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.

Returns of shape (N, 6, H, W) encoding the input images pairs for FlowNetSimple.

Return type Tensor

class mmtrack.models.motion.**KalmanFilter**(*center_only=False*)

A simple Kalman filter for tracking bounding boxes in image space.

The implementation is referred to https://github.com/nwojke/deep_sort.

gating_distance(*mean*, *covariance*, *measurements*, *only_position=False*)

Compute gating distance between state distribution and measurements.

A suitable distance threshold can be obtained from *chi2inv95*. If *only_position* is False, the chi-square distribution has 4 degrees of freedom, otherwise 2.

Parameters

- **mean** (*ndarray*) – Mean vector over the state distribution (8 dimensional).
- **covariance** (*ndarray*) – Covariance of the state distribution (8x8 dimensional).
- **measurements** (*ndarray*) – An Nx4 dimensional matrix of N measurements, each in format (x, y, a, h) where (x, y) is the bounding box center position, a the aspect ratio, and h the height.
- **only_position** (*bool, optional*) – If True, distance computation is done with respect to the bounding box center position only. Defaults to False.

Returns Returns an array of length N, where the i-th element contains the squared Mahalanobis distance between (mean, covariance) and *measurements[i]*.

Return type ndarray

initiate(*measurement*)

Create track from unassociated measurement.

Parameters

- **measurement** (*ndarray*) – Bounding box coordinates (x, y, a, h) with
- **position** (*center*) –

Returns

Returns the mean vector (8 dimensional) and covariance matrix (8x8 dimensional) of the new track. Unobserved velocities are initialized to 0 mean.

Return type (ndarray, ndarray)

predict(*mean*, *covariance*)

Run Kalman filter prediction step.

Parameters

- **mean** (*ndarray*) – The 8 dimensional mean vector of the object state at the previous time step.
- **covariance** (*ndarray*) – The 8x8 dimensional covariance matrix of the object state at the previous time step.

Returns

Returns the mean vector and covariance matrix of the predicted state. Unobserved velocities are initialized to 0 mean.

Return type (ndarray, ndarray)

project(*mean*, *covariance*)

Project state distribution to measurement space.

Parameters

- **mean** (*ndarray*) – The state's mean vector (8 dimensional array).
- **covariance** (*ndarray*) – The state's covariance matrix (8x8 dimensional).

Returns Returns the projected mean and covariance matrix of the given state estimate.

Return type (ndarray, ndarray)

track(*tracks*, *bboxes*)
Track forward.

Parameters

- **(dict[int** (*tracks*) – dict]): Track buffer.
- **bboxes** (*Tensor*) – Detected bounding boxes.

Returns dict], Tensor): Updated tracks and bboxes.

Return type (dict[int

update(*mean*, *covariance*, *measurement*)
Run Kalman filter correction step.

Parameters

- **mean** (*ndarray*) – The predicted state's mean vector (8 dimensional).
- **covariance** (*ndarray*) – The state's covariance matrix (8x8 dimensional).
- **measurement** (*ndarray*) – The 4 dimensional measurement vector (x, y, a, h), where (x, y) is the center position, a the aspect ratio, and h the height of the bounding box.

Returns Returns the measurement-corrected state distribution.

Return type (ndarray, ndarray)

class mmtrack.models.motion.**LinearMotion**(*num_samples=2*, *center_motion=False*)
Linear motion while tracking.

Parameters

- **num_samples** (*int, optional*) – Number of samples to calculate the velocity. Default to 2.
- **center_motion** (*bool, optional*) – Whether use center location or bounding box location to estimate the velocity. Default to False.

center(*bbox*)
Get the center of the box.

get_velocity(*bboxes*, *num_samples=None*)
Get velocities of the input objects.

step(*bboxes*, *velocity=None*)
Step forward with the velocity.

track(*tracks*, *frame_id*)
Tracking forward.

28.8 reid

class mmtrack.models.reid.**BaseReID**(*backbone*, *neck=None*, *head=None*, *pretrained=None*, *train_cfg=None*, *init_cfg=None*)

Base class for re-identification.

forward_train(*img*, *gt_label*, ***kwargs*)

"Training forward function.

simple_test(*img*, ***kwargs*)

Test without augmentation.

class mmtrack.models.reid.**FcModule**(*in_channels*, *out_channels*, *norm_cfg=None*, *act_cfg={'type': 'ReLU'}*, *inplace=True*, *init_cfg={'layer': 'Linear', 'type': 'Kaiming'}*)

Fully-connected layer module.

Parameters

- **in_channels** (*int*) – Input channels.
- **out_channels** (*int*) – Ourput channels.
- **norm_cfg** (*dict, optional*) – Configuration of normlization method after fc. Defaults to None.
- **act_cfg** (*dict, optional*) – Configuration of activation method after fc. Defaults to dict(type='ReLU').
- **inplace** (*bool, optional*) – Whether inplace the activatation module.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to dict(type='Kaiming', layer='Linear').

forward(*x*, *activate=True*, *norm=True*)

Model forward.

property norm

Normalization.

class mmtrack.models.reid.**GlobalAveragePooling**(*kernel_size=None*, *stride=None*)

Global Average Pooling neck.

Note that we use *view* to remove extra channel after pooling. We do not use *squeeze* as it will also remove the batch dimension when the tensor has a batch dimension of size 1, which can lead to unexpected errors.

class mmtrack.models.reid.**LinearReIDHead**(*num_fcs*, *in_channels*, *fc_channels*, *out_channels*, *norm_cfg=None*, *act_cfg=None*, *num_classes=None*, *loss=None*, *loss_pairwise=None*, *topk=(1)*, *init_cfg={'bias': 0, 'layer': 'Linear', 'mean': 0, 'std': 0.01, 'type': 'Normal'}*)

Linear head for re-identification.

Parameters

- **num_fcs** (*int*) – Number of fcs.
- **in_channels** (*int*) – Number of channels in the input.
- **fc_channels** (*int*) – Number of channels in the fcs.
- **out_channels** (*int*) – Number of channels in the output.
- **norm_cfg** (*dict, optional*) – Configuration of normlization method after fc. Defaults to None.
- **act_cfg** (*dict, optional*) – Configuration of activation method after fc. Defaults to None.

- **num_classes** (*int, optional*) – Number of the identities. Default to None.
- **loss** (*dict, optional*) – Cross entropy loss to train the re-identificaiton module.
- **loss_pairwise** (*dict, optional*) – Triplet loss to train the re-identificaiton module.
- **topk** (*int, optional*) – Calculate topk accuracy. Default to False.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to dict(type='Normal',layer='Linear', mean=0, std=0.01, bias=0).

forward_train(*x*)
Model forward.

loss(*gt_label*, *feats*, *cls_score=None*)
Compute losses.

28.9 roi_heads

class mmtrack.models.roi_heads.**SelsaBBoxHead**(*aggregator*, **args*, ***kwargs*)
Selsa bbox head.

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

Parameters **aggregator** (*dict*) – Configuration of aggregator.

forward(*x*, *ref_x*)
Computing the *cls_score* and *bbox_pred* of the features *x* of key frame proposals.

Parameters

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

Returns The predicted score of classes and the predicted regression offsets.

Return type tuple(cls_score, bbox_pred)

class mmtrack.models.roi_heads.**SelsaRoIHead**(*bbox_roi_extractor=None*, *bbox_head=None*, *mask_roi_extractor=None*, *mask_head=None*, *shared_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
selsa roi head.

forward_train(*x*, *ref_x*, *img_metas*, *proposal_list*, *ref_proposal_list*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*, *gt_masks=None*)

Parameters

- **x** (*list[Tensor]*) – list of multi-level img features.
- **ref_x** (*list[Tensor]*) – list of multi-level ref_img features.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **proposal_list** (*list[Tensors]*) – list of region proposals.
- **ref_proposal_list** (*list[Tensors]*) – list of region proposals from ref_imgs.

- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task.

Returns a dictionary of loss components

Return type dict[str, Tensor]

simple_test(*x*, *ref_x*, *proposals_list*, *ref_proposals_list*, *img_metas*, *proposals=None*, *rescale=False*)
Test without augmentation.

simple_test_bboxes(*x*, *ref_x*, *proposals*, *ref_proposals*, *img_metas*, *rcnn_test_cfg*, *rescale=False*)
Test only det bboxes without augmentation.

class mmtrack.models.roi_heads.**SingleRoIExtractor**(*roi_layer*, *out_channels*, *featmap_strides*, *finest_scale=56*, *init_cfg=None*)

Extract RoI features from a single level feature map.

This Class is the same as *SingleRoIExtractor* from *mmdet.models.roi_heads.roi_extractors* except for using ***kwargs* to accept external arguments.

forward(*feats*, *rois*, *roi_scale_factor=None*, ***kwargs*)
Forward function.

class mmtrack.models.roi_heads.**TemporalRoIAlign**(*num_most_similar_points=2*, *num_temporal_attention_blocks=4*, **args*, ***kwargs*)

Temporal RoI Align module.

This module is proposed in "Temporal ROI Align for Video Object Recognition". TRoI Align.

Parameters

- **num_most_similar_points** (*int*) – Denotes the number of the most similar points in the Most Similar RoI Align. Defaults to 2.
- **num_temporal_attention_blocks** (*int*) – Denotes the number of temporal attention blocks in the Temporal Attentional Feature Aggregation. If the value isn't greater than 0, the averaging operation will be adopted to aggregate the RoI features with the Most Similar RoI features. Defaults to 4.

forward(*feats*, *rois*, *roi_scale_factor=None*, *ref_feats=None*)
Forward function.

most_similar_roi_align(*roi_feats*, *ref_feats*)
Extract the Most Similar RoI features from reference feature maps *ref_feats* based on RoI features *roi_feats*.

The extraction mainly contains three steps: 1. Compute cos similarity maps between *roi_feats* and *ref_feats*. 2. Pick the top K points based on the similarity maps. 3. Project these top K points into reference feature maps *ref_feats*.

Parameters

- **roi_feats** (*Tensor*) – of shape [roi_n, C, roi_h, roi_w]. roi_n, roi_h and roi_w denote the number of key frame proposals, the height of RoI features and the width of RoI features, respectively.

- **ref_feats** (*Tensor*) – of shape [img_n, C, img_h, img_w]. img_n, img_h and img_w denote the number of reference frames, the height of reference frame feature maps and the width of reference frame feature maps, respectively.

Returns

The extracted Most Similar RoI features from reference feature maps with shape [img_n, roi_n, C, roi_h, roi_w].

Return type Tensor

temporal_attentional_feature_aggregation(*x*, *ref_x*)

Aggregate the RoI features *x* with the Most Similar RoI features *ref_x*.

The aggregation mainly contains three steps: 1. Pass through a tiny embed network. 2. Use multi-head attention to computing the weight between *x* and *ref_x*. 3. Use the normlized (i.e. softmax) weight to weightedly sum *x* and *ref_x*.

Parameters

- **x** (*Tensor*) – of shape [1, roi_n, C, roi_h, roi_w]. roi_n, roi_h and roi_w denote the number of key frame proposals, the height of RoI features and the width of RoI features, respectively.
- **ref_x** (*Tensor*) – of shape [img_n, roi_n, C, roi_h, roi_w]. img_n is the number of reference images.

Returns

The aggregated Temporal RoI features of key frame proposals with shape [roi_n, C, roi_h, roi_w].

Return type Tensor

28.10 track_heads

class mmtrack.models.track_heads.**CornerPredictorHead**(*inplanes*, *channel*, *feat_size=20*, *stride=16*)

Corner Predictor head.

Parameters

- **inplanes** (*int*) – input channel
- **channel** (*int*) – the output channel of the first conv block
- **feat_size** (*int*) – the size of feature map
- **stride** (*int*) – the stride of feature map from the backbone

forward(*x*)

Forward pass with input x.

Parameters **x** (*Tensor*) – of shape (bs, C, H, W).

Returns bbox of shape (bs, 4) in (tl_x, tl_y, br_x, br_y) format.

Return type (Tensor)

get_score_map(*x*)

Score map branch.

Parameters **x** (*Tensor*) – of shape (bs, C, H, W).

Returns

of shape (bs, 1, H, W). The score map of top left corner of tracking bbox.

score_map_br (Tensor): of shape (bs, 1, H, W). The score map of bottom right corner of tracking bbox.

Return type score_map_tl (Tensor)

soft_argmax(*score_map*)

Get soft-argmax coordinate for the given score map.

Parameters **score_map** (*self.feat_size, self.feat_size*) – the last score map in bbox_head branch

Returns

of shape (bs, 1). The values are in range [0, self.feat_size * self.stride]

exp_y (Tensor): of shape (bs, 1). The values are in range [0, self.feat_size * self.stride]

Return type exp_x (Tensor)

class mmtrack.models.track_heads.**CorrelationHead**(*in_channels*, *mid_channels*, *out_channels*, *kernel_size=3*, *norm_cfg={'type': 'BN'}*, *act_cfg={'type': 'ReLU'}*, *init_cfg=None*, ***kwargs*)

Correlation head module.

This module is proposed in "SiamRPN++: Evolution of Siamese Visual Tracking with Very Deep Networks. SiamRPN++.

Parameters

- **in_channels** (*int*) – Input channels.
- **mid_channels** (*int*) – Middle channels.
- **out_channels** (*int*) – Output channels.
- **kernel_size** (*int*) – Kernel size of convs. Defaults to 3.
- **norm_cfg** (*dict*) – Configuration of normlization method after each conv. Defaults to dict(type='BN').
- **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(*kernel*, *search*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmtrack.models.track_heads.**MixFormerHead**(*bbox_head=None*, *score_head=None*, *loss_bbox={'loss_weight': 5.0, 'type': 'L1Loss'}*, *loss_iou={'loss_weight': 2.0, 'type': 'GIoULoss'}*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*)

MixFormer head module for bounding box regression and prediction of confidence of tracking bbox.

This module is proposed in "MixFormer: End-to-End Tracking with Iterative Mixed Attention". MixFormer.

forward(*template*, *search*, *run_score_head=True*, *gt_bboxes=None*)

Parameters

- **template** (*Tensor*) – Template features extracted from backbone,
- **shape** (*with*) –
- **search** (*Tensor*) – Search region features extracted from backbone,
- **shape** –

Returns

- **'pred_bboxes': (Tensor) of shape (bs, 1, 4), in** [tl_x, tl_y, br_x, br_y] format
- 'pred_scores': (Tensor) of shape (bs, 1, 1)

Return type (dict)

forward_bbox_head(*search*)

Parameters

- **search** (*Tensor*) – Search region features extracted from backbone,
- **shape** (*with*) –

Returns of shape (bs, 1, 4). The bbox is in [tl_x, tl_y, br_x, by_y] format.

Return type Tensor

loss(*track_results*, *gt_bboxes*, *gt_labels*, *img_size=None*)

compute loss. Not Implemented yet!

Parameters

- **track_results** (*dict*) – it may contains the following keys: - 'pred_bboxes': bboxes of (N, num_query, 4) shape in

  [tl_x, tl_y, br_x, br_y] format.

  - 'pred_scores': scores of (N, num_query, 1) shaoe.

- **gt_bboxes** (*list[Tensor]*) – ground truth bboxes for search image with shape (N, 5) in [0., tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – ground truth labels for search imges with shape (N, 2).
- **img_size** (*tuple, optional*) – the size (h, w) of original search image. Defaults to None.

class mmtrack.models.track_heads.**MixFormerScoreDecoder**(*pool_size=4*, *feat_size=20*, *stride=16*, *num_heads=6*, *hidden_dim=384*, *num_layers=3*)

Score Prediction Module (SPM) proposed in "MixFormer: End-to-End Tracking with Iterative Mixed Attention". MixFormer.

Parameters

- **pool_size** (*int*) – pool size for roi pooling

- **feat_size** (*int*) – search region feature map size
- **stride** (*int*) – ratio between original image size
- **feature map size** (*and*) –
- **num_heads** (*int*) – number of heads of attention
- **hidden_dim** (*int*) – embedding dimension
- **num_layer** (*int*) – number of layers of the mlp

forward(*search_feat*, *template_feat*, *search_box*)

Parameters

- **search_feat** (*Tensor*) – Search region features extracted from
- **with shape** (*backbone*) –
- **template_feat** (*Tensor*) – Template features extracted from
- **with shape** –
- **search_box** (*Tensor*) – of shape (B, 4), in
- **[tl_x** –
- **tl_y** –
- **br_x** –
- **format.** (*br_y]*) –

Returns

Confidence score of the predicted result. of shape (b, 1, 1)

Return type out_score (Tensor)

class mmtrack.models.track_heads.**QuasiDenseEmbedHead**(*embed_channels=256*, *softmax_temp=- 1*, *loss_track={'loss_weight': 0.25, 'type': 'MultiPosCrossEntropyLoss'}*, *loss_track_aux={'hard_mining': True, 'loss_weight': 1.0, 'margin': 0.3, 'sample_ratio': 3, 'type': 'L2Loss'}*, *init_cfg={'bias': 0, 'distribution': 'uniform', 'layer': 'Linear', 'override': {'bias': 0, 'mean': 0, 'name': 'fc_embed', 'std': 0.01, 'type': 'Normal'}, 'type': 'Xavier'}*, **args*, ***kwargs*)

The quasi-dense roi embed head.

Parameters

- **embed_channels** (*int*) – The input channel of embed features. Defaults to 256.
- **softmax_temp** (*int*) – Softmax temperature. Defaults to -1.
- **loss_track** (*dict*) – The loss function for tracking. Defaults to MultiPosCrossEntropyLoss.
- **loss_track_aux** (*dict*) – The auxiliary loss function for tracking. Defaults to L2Loss.

forward(*x*)

Forward the input *x*.

get_targets(*gt_match_indices*, *key_sampling_results*, *ref_sampling_results*)

Calculate the track targets and track weights for all samples in a batch according to the sampling_results.

Parameters

- **(List[obj** (*ref_sampling_results*) – SamplingResults]): Assign results of all images in a batch after sampling.
- **(List[obj** – SamplingResults]): Assign results of all reference images in a batch after sampling.
- **gt_match_indices** (*list(Tensor)*) – Mapping from gt_instance_ids to ref_gt_instance_ids of the same tracklet in a pair of images.

Returns

Association results. Containing the following list of Tensors:

- **track_targets (list[Tensor]): The mapping instance ids from** all positive proposals in the key image to all proposals in the reference image, each tensor in list has shape (len(key_pos_bboxes), len(ref_bboxes)).
- **track_weights (list[Tensor]): Loss weights for all positive** proposals in a batch, each tensor in list has shape (len(key_pos_bboxes),).

Return type Tuple[list[Tensor]]

loss(*dists*, *cos_dists*, *targets*, *weights*)

Calculate the track loss and the auxiliary track loss.

Parameters

- **dists** (*list[Tensor]*) – Dot-product dists between key_embeds and ref_embeds.
- **cos_dists** (*list[Tensor]*) – Cosine dists between key_embeds and ref_embeds.
- **targets** (*list[Tensor]*) – The mapping instance ids from all positive proposals in the key image to all proposals in the reference image, each tensor in list has shape (len(key_pos_bboxes), len(ref_bboxes)).
- **weights** (*list[Tensor]*) – Loss weights for all positive proposals in a batch, each tensor in list has shape (len(key_pos_bboxes),).

Returns

Tensor]: Calculation results. Containing the following list of Tensors:

- loss_track (Tensor): Results of loss_track function.
- loss_track_aux (Tensor): Results of loss_track_aux function.

Return type Dict [str

match(*key_embeds*, *ref_embeds*, *key_sampling_results*, *ref_sampling_results*)

Calculate the dist matrixes for loss measurement.

Parameters

- **key_embeds** (*Tensor*) – Embeds of positive bboxes in sampling results of key image.
- **ref_embeds** (*Tensor*) – Embeds of all bboxes in sampling results of the reference image.
- **(List[obj** (*ref_sampling_results*) – SamplingResults]): Assign results of all images in a batch after sampling.
- **(List[obj** – SamplingResults]): Assign results of all reference images in a batch after sampling.

Returns

Calculation results. Containing the following list of Tensors:

- **dists (list[Tensor]): Dot-product dists between** key_embeds and ref_embeds, each tensor in list has shape (len(key_pos_bboxes), len(ref_bboxes)).
- **cos_dists (list[Tensor]): Cosine dists between** key_embeds and ref_embeds, each tensor in list has shape (len(key_pos_bboxes), len(ref_bboxes)).

Return type Tuple[list[Tensor]]

class mmtrack.models.track_heads.**QuasiDenseTrackHead**(**args*, ***kwargs*)

The quasi-dense track head.

extract_bbox_feats(*x*, *bboxes*)

Extract roi features.

forward_train(*x*, *img_metas*, *proposal_list*, *gt_bboxes*, *gt_labels*, *gt_match_indices*, *ref_x*, *ref_img_metas*, *ref_proposals*, *ref_gt_bboxes*, *ref_gt_labels*, *gt_bboxes_ignore=None*, *gt_masks=None*, *ref_gt_bboxes_ignore=None*, *ref_gt_mask=None*, **args*, ***kwargs*)

Forward function during training.

Args: x (list[Tensor]): list of multi-level image features. img_metas (list[dict]): list of image info dict where each dict

has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.

proposal_list (list[Tensors]): list of region proposals. gt_bboxes (list[Tensor]): Ground truth bboxes of the image,

each item has a shape (num_gts, 4).

gt_labels (list[Tensor]): Ground truth labels of all images. each has a shape (num_gts,).

gt_match_indices (list(Tensor)): Mapping from gt_instance_ids to ref_gt_instance_ids of the same tracklet in a pair of images.

ref_x (list[Tensor]): list of multi-level ref_img features. ref_img_metas (list[dict]): list of reference image info dict where

each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.

ref_proposal_list (list[Tensors]): list of ref_img region proposals.

ref_gt_bboxes (list[Tensor]): Ground truth bboxes of the reference image, each item has a shape (num_gts, 4).

ref_gt_labels (list[Tensor]): Ground truth labels of all reference images, each has a shape (num_gts,).

gt_bboxes_ignore (list[Tensor], None): Ground truth bboxes to be ignored, each item has a shape (num_ignored_gts, 4).

gt_masks (list[Tensor]) [Masks for each bbox, has a shape] (num_gts, h , w).

ref_gt_bboxes_ignore (list[Tensor], None): Ground truth bboxes of reference images to be ignored, each item has a shape (num_ignored_gts, 4).

ref_gt_masks (list[Tensor]) [Masks for each reference bbox,] has a shape (num_gts, h , w).

Returns Tensor]: Track losses.

Return type dict[str

class mmtrack.models.track_heads.**RoIEmbedHead**(*num_convs=0*, *num_fcs=0*, *roi_feat_size=7*, *in_channels=256*, *conv_out_channels=256*, *with_avg_pool=False*, *fc_out_channels=1024*, *conv_cfg=None*, *norm_cfg=None*, *loss_match={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': False}*, *init_cfg=None*, ***kwargs*)

The roi embed head.

This module is used in multi-object tracking methods, such as MaskTrack R-CNN.

Parameters

- **num_convs** (*int*) – The number of convoluational layers to embed roi features. Defaults to 0.
- **num_fcs** (*int*) – The number of fully connection layers to embed roi features. Defaults to 0.
- **roi_feat_size** (*int* | *tuple(int)*) – The spatial size of roi features. Defaults to 7.
- **in_channels** (*int*) – The input channel of roi features. Defaults to 256.
- **conv_out_channels** (*int*) – The output channel of roi features after forwarding convoluational layers. Defaults to 256.
- **with_avg_pool** (*bool*) – Whether use average pooling before passing roi features into fully connection layers. Defaults to False.
- **fc_out_channels** (*int*) – The output channel of roi features after forwarding fully connection layers. Defaults to 1024.
- **conv_cfg** (*dict*) – Config dict for convolution layer. Defaults to None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to None.
- **loss_match** (*dict*) – The loss function. Defaults to dict(type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0)
- **init_cfg** (*dict*) – Configuration of initialization. Defaults to None.

forward(*x*, *ref_x*, *num_x_per_img*, *num_x_per_ref_img*)

Computing the similarity scores between *x* and *ref_x*.

Parameters

- **x** (*Tensor*) – of shape [N, C, H, W]. N is the number of key frame proposals.
- **ref_x** (*Tensor*) – of shape [M, C, H, W]. M is the number of reference frame proposals.
- **num_x_per_img** (*list[int]*) – The *x* contains proposals of multi-images. *num_x_per_img* denotes the number of proposals for each key image.
- **num_x_per_ref_img** (*list[int]*) – The *ref_x* contains proposals of multi-images. *num_x_per_ref_img* denotes the number of proposals for each reference image.

Returns The predicted similarity_logits of each pair of key image and reference image.

Return type list[Tensor]

`get_targets`(*sampling_results*, *gt_instance_ids*, *ref_gt_instance_ids*)
Calculate the ground truth for all samples in a batch according to the sampling_results.

Parameters

- **(List[obj** (*sampling_results*) – SamplingResults]): Assign results of all images in a batch after sampling.
- **gt_instance_ids** (*list[Tensor]*) – The instance ids of gt_bboxes of all images in a batch, each tensor has shape (num_gt,).
- **ref_gt_instance_ids** (*list[Tensor]*) – The instance ids of gt_bboxes of all reference images in a batch, each tensor has shape (num_gt,).

Returns

Ground truth for proposals in a batch. Containing the following list of Tensors:

- track_id_targets (list[Tensor]): The instance ids of Gt_labels for all proposals in a batch, each tensor in list has shape (num_proposals,).
- track_id_weights (list[Tensor]): Labels_weights for all proposals in a batch, each tensor in list has shape (num_proposals,).

Return type Tuple[list[Tensor]]

`loss`(*similarity_logits*, *track_id_targets*, *track_id_weights*, *reduction_override=None*)
Calculate the loss in a batch.

Parameters

- **similarity_logits** (*list[Tensor]*) – The predicted similarity_logits of each pair of key image and reference image.
- **track_id_targets** (*list[Tensor]*) – The instance ids of Gt_labels for all proposals in a batch, each tensor in list has shape (num_proposals,).
- **track_id_weights** (*list[Tensor]*) – Labels_weights for all proposals in a batch, each tensor in list has shape (num_proposals,).
- **reduction_override** (*str, optional*) – The method used to reduce the loss. Options are "none", "mean" and "sum".

Returns a dictionary of loss components.

Return type dict[str, Tensor]

`class mmtrack.models.track_heads.RoITrackHead`(*roi_extractor=None*, *embed_head=None*, *regress_head=None*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, **args*, ***kwargs*)

The roi track head.

This module is used in multi-object tracking methods, such as MaskTrack R-CNN.

Parameters

- **roi_extractor** (*dict*) – Configuration of roi extractor. Defaults to None.
- **embed_head** (*dict*) – Configuration of embed head. Defaults to None.
- **train_cfg** (*dict*) – Configuration when training. Defaults to None.
- **test_cfg** (*dict*) – Configuration when testing. Defaults to None.

- **init_cfg** (*dict*) – Configuration of initialization. Defaults to None.

extract_roi_feats(*x*, *bboxes*)
Extract roi features.

forward_train(*x*, *ref_x*, *img_metas*, *proposal_list*, *gt_bboxes*, *ref_gt_bboxes*, *gt_labels*, *gt_instance_ids*, *ref_gt_instance_ids*, *gt_bboxes_ignore=None*, ***kwargs*)

Parameters

- **x** (*list[Tensor]*) – list of multi-level image features.
- **ref_x** (*list[Tensor]*) – list of multi-level ref_img features.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmtrack/datasets/pipelines/formatting.py:VideoCollect*.
- **proposal_list** (*list[Tensors]*) – list of region proposals.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **ref_gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each reference image with shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box.
- **gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bbox.
- **ref_gt_instance_ids** (*None | list[Tensor]*) – specify the instance id for each ground truth bbox of reference images.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.

Returns a dictionary of loss components

Return type dict[str, Tensor]

init_assigner_sampler()
Initialize assigner and sampler.

init_embed_head(*roi_extractor*, *embed_head*)
Initialize embed_head

simple_test(*roi_feats*, *prev_roi_feats*)
Test without augmentations.

property with_track
whether the mulit-object tracker has a embed head

Type bool

class mmtrack.models.track_heads.**SiameseRPNHead**(*anchor_generator*, *in_channels*, *kernel_size=3*, *norm_cfg={'type': 'BN'}*, *weighted_sum=False*, *bbox_coder={'target_means': [0.0, 0.0, 0.0, 0.0], 'target_stds': [1.0, 1.0, 1.0, 1.0], 'type': 'DeltaXYWHBBoxCoder'}*, *loss_cls={'loss_weight': 1.0, 'reduction': 'sum', 'type': 'CrossEntropyLoss'}*, *loss_bbox={'loss_weight': 1.2, 'reduction': 'sum', 'type': 'L1Loss'}*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, **args*, ***kwargs*)

Siamese RPN head.

This module is proposed in "SiamRPN++: Evolution of Siamese Visual Tracking with Very Deep Networks. SiamRPN++.

Parameters

- **anchor_generator** (*dict*) – Configuration to build anchor generator module.
- **in_channels** (*int*) – Input channels.
- **kernel_size** (*int*) – Kernel size of convs. Defaults to 3.
- **norm_cfg** (*dict*) – Configuration of normlization method after each conv. Defaults to dict(type='BN').
- **weighted_sum** (*bool*) – If True, use learnable weights to weightedly sum the output of multi heads in siamese rpn , otherwise, use averaging. Defaults to False.
- **bbox_coder** (*dict*) – Configuration to build bbox coder. Defaults to dict(type='DeltaXYWHBBoxCoder', target_means=[0., 0., 0., 0.], target_stds=[1., 1., 1., 1.]).
- **loss_cls** (*dict*) – Configuration to build classification loss. Defaults to dict(type='CrossEntropyLoss', reduction='sum', loss_weight=1.0)
- **loss_bbox** (*dict*) – Configuration to build bbox regression loss. Defaults to dict(type='L1Loss', reduction='sum', loss_weight=1.2).
- **train_cfg** (*Dict*) – Training setting. Defaults to None.
- **test_cfg** (*Dict*) – Testing setting. Defaults to None.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*z_feats*, *x_feats*)

Forward with features *z_feats* of exemplar images and features *x_feats* of search images.

Parameters

- **z_feats** (*tuple[Tensor]*) – Tuple of Tensor with shape (N, C, H, W) denoting the multi level feature maps of exemplar images. Typically H and W equal to 7.
- **x_feats** (*tuple[Tensor]*) – Tuple of Tensor with shape (N, C, H, W) denoting the multi level feature maps of search images. Typically H and W equal to 31.

Returns cls_score is a Tensor with shape (N, 2 * num_base_anchors, H, W), bbox_pred is a Tensor with shape (N, 4 * num_base_anchors, H, W), Typically H and W equal to 25.

Return type tuple(cls_score, bbox_pred)

get_bbox(*cls_score*, *bbox_pred*, *prev_bbox*, *scale_factor*)

Track *prev_bbox* to current frame based on the output of network.

Parameters

- **cls_score** (*Tensor*) – of shape (1, 2 * num_base_anchors, H, W).
- **bbox_pred** (*Tensor*) – of shape (1, 4 * num_base_anchors, H, W).
- **prev_bbox** (*Tensor*) – of shape (4,) in [cx, cy, w, h] format.
- **scale_factor** (*Tensr*) – scale factor.

Returns best_score is a Tensor denoting the score of *best_bbox*, best_bbox is a Tensor of shape (4,) with [cx, cy, w, h] format, which denotes the best tracked bbox in current frame.

Return type tuple(best_score, best_bbox)

get_targets(*gt_bboxes*, *score_maps_size*, *is_positive_pairs*)

Generate the training targets for exemplar image and search image pairs.

Parameters

- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes of each search image with shape (1, 5) in [0.0, tl_x, tl_y, br_x, br_y] format.
- **score_maps_size** (*torch.size*) – denoting the output size (height, width) of the network.
- **is_positive_pairs** (*bool*) – list of bool denoting whether each ground truth bbox in *gt_bboxes* is positive.

Returns tuple(all_labels, all_labels_weights, all_bbox_targets, all_bbox_weights): the shape is (N, H * W * num_base_anchors), (N, H * W * num_base_anchors), (N, H * W * num_base_anchors, 4), (N, H * W * num_base_anchors, 4), respectively. All of them are Tensor.

loss(*cls_score*, *bbox_pred*, *labels*, *labels_weights*, *bbox_targets*, *bbox_weights*)

Compute loss.

Parameters

- **cls_score** (*Tensor*) – of shape (N, 2 * num_base_anchors, H, W).
- **bbox_pred** (*Tensor*) – of shape (N, 4 * num_base_anchors, H, W).
- **labels** (*Tensor*) – of shape (N, H * W * num_base_anchors).
- **labels_weights** (*Tensor*) – of shape (N, H * W * num_base_anchors).
- **bbox_targets** (*Tensor*) – of shape (N, H * W * num_base_anchors, 4).
- **bbox_weights** (*Tensor*) – of shape (N, H * W * num_base_anchors, 4).

Returns a dictionary of loss components

Return type dict[str, Tensor]

class mmtrack.models.track_heads.**StarkHead**(*num_query=1*, *transformer=None*, *positional_encoding={'normalize': True, 'num_feats': 128, 'type': 'SinePositionalEncoding'}*, *bbox_head=None*, *cls_head=None*, *loss_cls={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': False}*, *loss_bbox={'loss_weight': 5.0, 'type': 'L1Loss'}*, *loss_iou={'loss_weight': 2.0, 'type': 'GIoULoss'}*, *train_cfg=None*, *test_cfg=None*, *init_cfg=None*, *frozen_modules=None*, ***kwargs*)

STARK head module for bounding box regression and prediction of confidence score of tracking bbox.

This module is proposed in "Learning Spatio-Temporal Transformer for Visual Tracking". STARK.

Parameters

- **num_query** (*int*) – Number of query in transformer.
- **(obj** (*test_cfg*) – `mmcv.ConfigDict`|dict): Config for transformer. Default: None.
- **(obj** – `mmcv.ConfigDict`|dict): Config for position encoding.
- **(obj** – `mmcv.ConfigDict`|dict, optional): Config for bbox head. Defaults to None.
- **(obj** – `mmcv.ConfigDict`|dict, optional): Config for classification head. Defaults to None.
- **(obj** – *mmcv.ConfigDict`|dict): Config of the classification loss. Default `CrossEntropyLoss*.
- **(obj** – *mmcv.ConfigDict`|dict): Config of the bbox regression loss. Default `L1Loss*.
- **(obj** – *mmcv.ConfigDict`|dict): Config of the bbox regression iou loss. Default `GIoULoss*.
- **(obj** – `mmcv.ConfigDict`|dict): Training config of transformer head.
- **(obj** – `mmcv.ConfigDict`|dict): Testing config of transformer head.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None

forward(*inputs*)

" :param inputs: The list contains the

multi-level features and masks of template or search images.

- **'feat': (tuple(Tensor)), the Tensor is of shape** (bs, c, h//stride, w//stride).
- 'mask': (Tensor), of shape (bs, h, w).

Here, *h* and *w* denote the height and width of input image respectively. *stride* is the stride of feature map.

Returns

- **'pred_bboxes': (Tensor) of shape (bs, num_query, 4), in** [tl_x, tl_y, br_x, br_y] format
- 'pred_logit': (Tensor) of shape (bs, num_query, 1)

Return type (dict)

forward_bbox_head(*feat*, *enc_mem*)

Parameters

- **feat** – output embeddings of decoder, with shape (1, bs, num_query, c).
- **enc_mem** – output embeddings of encoder, with shape (feats_flatten_len, bs, C)

  Here, 'feats_flatten_len' = z_feat_h*z_feat_w*2 + x_feat_h*x_feat_w. 'z_feat_h' and 'z_feat_w' denote the height and width of the template features respectively. 'x_feat_h' and 'x_feat_w' denote the height and width of search features respectively.

Returns

of shape (bs, num_query, 4). The bbox is in [tl_x, tl_y, br_x, br_y] format.

Return type Tensor

init_weights()
Parameters initialization.

loss(*track_results*, *gt_bboxes*, *gt_labels*, *img_size=None*)
Compute loss.

Parameters

- **track_results** (*dict*) – it may contains the following keys: - 'pred_bboxes': bboxes of (N, num_query, 4) shape in

  [tl_x, tl_y, br_x, br_y] format.

  - 'pred_logits': bboxes of (N, num_query, 1) shape.

- **gt_bboxes** (*list[Tensor]*) – ground truth bboxes for search images with shape (N, 5) in [0., tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – ground truth labels for search images with shape (N, 2).
- **img_size** (*tuple, optional*) – the size (h, w) of original search image. Defaults to None.

Returns a dictionary of loss components.

Return type dict[str, Tensor]

28.11 builder

mmtrack.models.**build_aggregator**(*cfg*)
Build aggregator model.

mmtrack.models.**build_model**(*cfg*, *train_cfg=None*, *test_cfg=None*)
Build model.

mmtrack.models.**build_motion**(*cfg*)
Build motion model.

mmtrack.models.**build_reid**(*cfg*)
Build reid model.

mmtrack.models.**build_tracker**(*cfg*)
Build tracker.

CHAPTER

TWENTYNINE

MMTRACK.UTILS

mmtrack.utils.**build_ddp**(*model*, *device='cuda'*, **args*, ***kwargs*)

Build DistributedDataParallel module by device type. If device is cuda, return a MMDistributedDataParallel model; if device is npu, return a NPUDistributedDataParallel model. :param model: module to be parallelized. :type model: `nn.Module` :param device: device type, npu or cuda. :type device: str

Returns the module to be parallelized

Return type `nn.Module`

References

mmtrack.utils.**build_dp**(*model*, *device='cuda'*, *dim=0*, **args*, ***kwargs*)

build DataParallel module by device type.

if device is cuda, return a MMDataParallel model; if device is npu, return a NPUDataParallel model. :param model: model to be parallelized. :type model: `nn.Module` :param device: device type, cuda, cpu or npu. Defaults to cuda. :type device: str :param dim: Dimension used to scatter the data. Defaults to 0. :type dim: int

Returns the model to be parallelized.

Return type nn.Module

mmtrack.utils.**collect_env**()

Collect the information of the running environments.

mmtrack.utils.**get_device**()

Returns an available device, cpu, cuda or npu.

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

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y