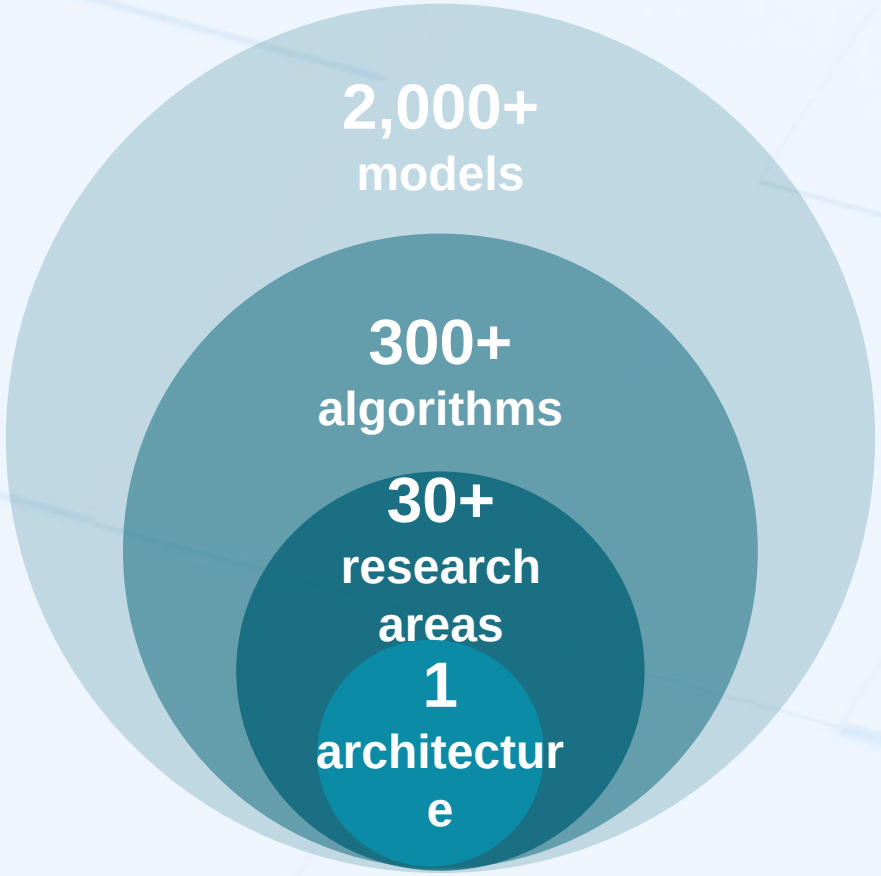
OpenMMLab: A Foundational Platform for Computer Vision Research and Production

Ruohui Wang
Shanghai AI Laboratory

Overview of OpenMMLab

Open-source computer vision algorithm platform



1

Architecture

- unified architecture for training and evaluation

30+

Research Areas

- cover a wide range of research areas

300+

Algorithms

- classic and state-of-the art algorithms

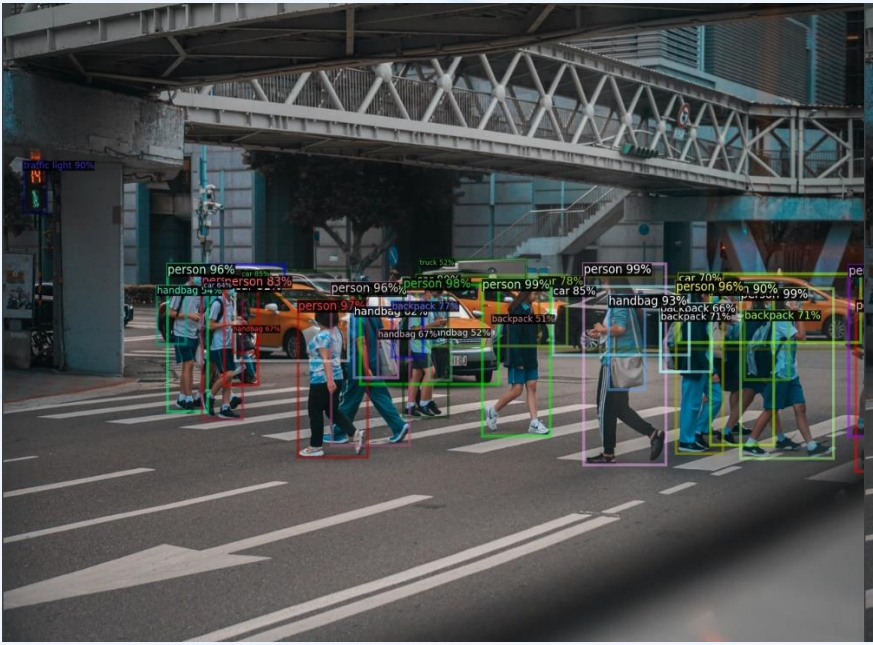
2000+

Pretrained Models

- fair benchmarks and out-of-the-box tools

History of development

MMDetection

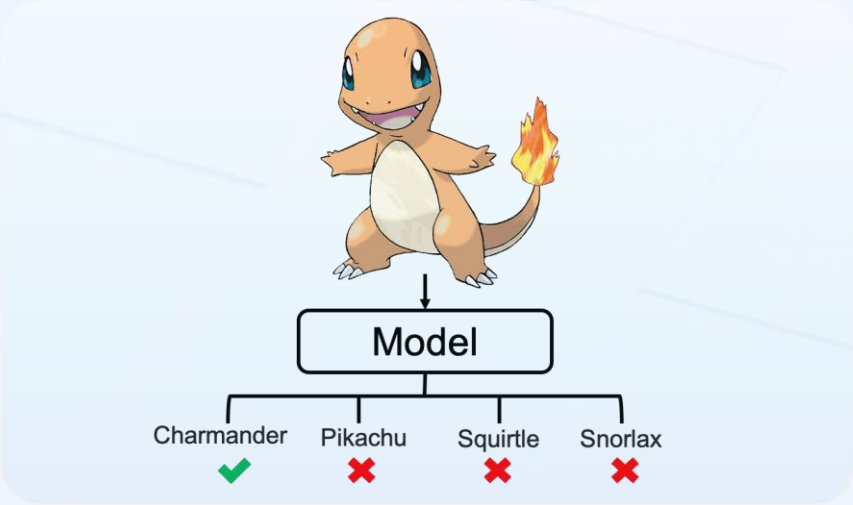


Object Detection

Instance Segmentation

Panoptic Segmentation

MMClassification



Representative models

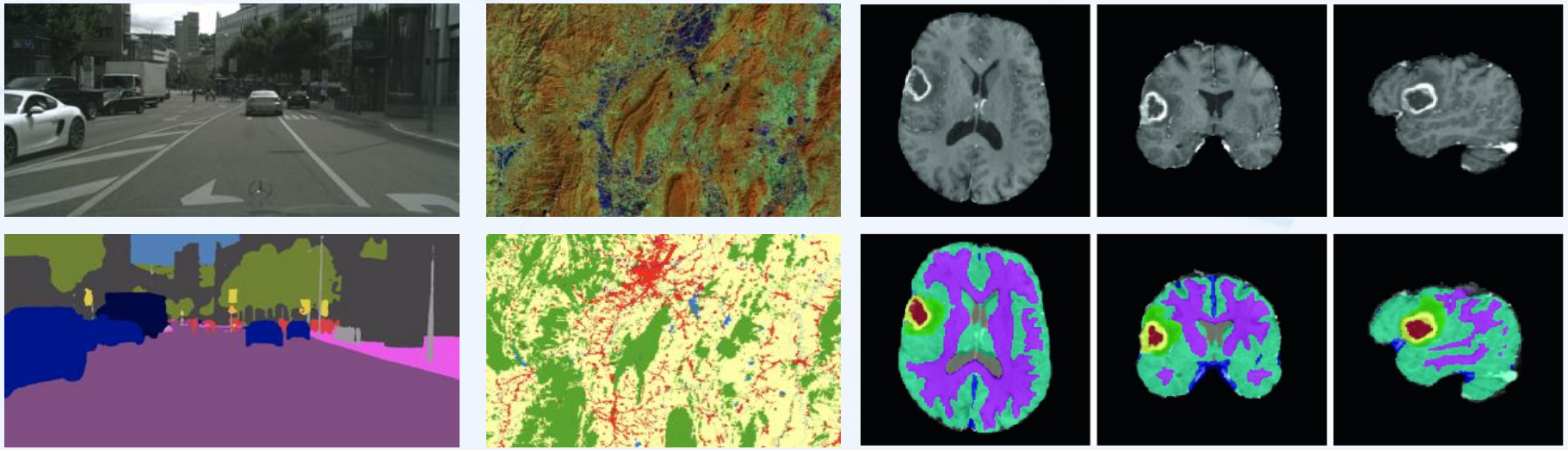
VGG	ResNet	ResNeXt ShuffleNet	ResNeSt RegNet	ViT Swin Transformer	ConvNeXt MobileOne
2014	2015	2017	2020	2021	2022

40+ backbones

mainstream datasets

rich training recipes

MMSegmentation



Self-driving | Remote sensing | Medical image analysis

Model zoo

- **400+** models
- **30+** algorithms

Modular design

- configurable
- extensible

Standard benchmark

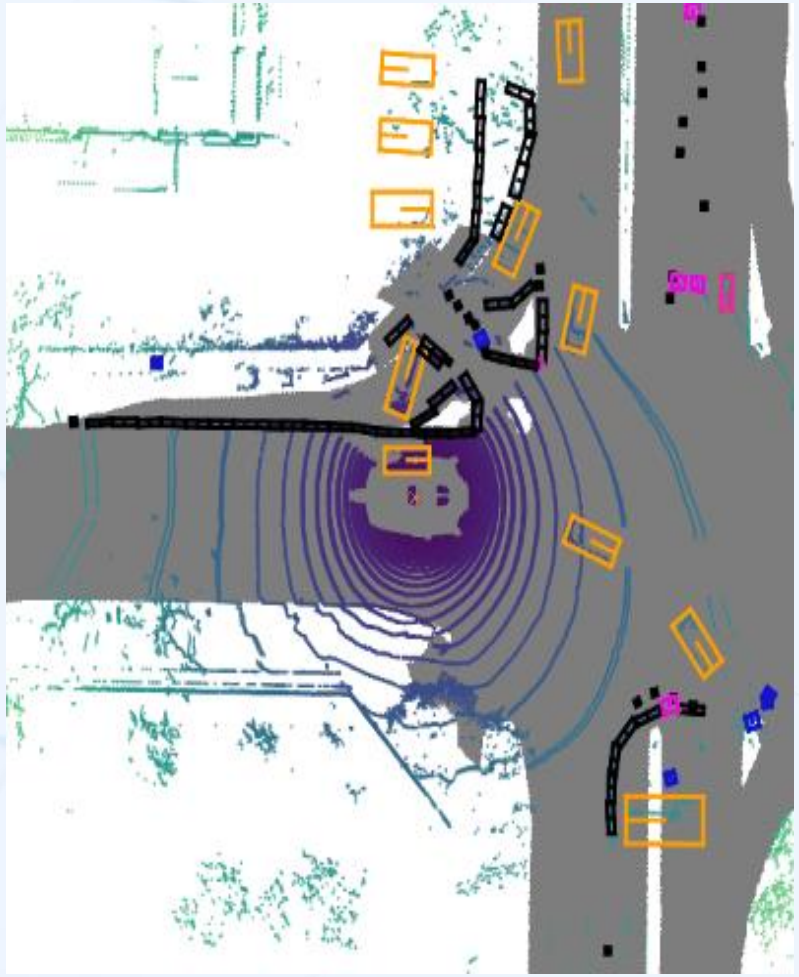
- ablation study
- fair comparison

Usage

- training
- testing
- inference API

MMDetection3D

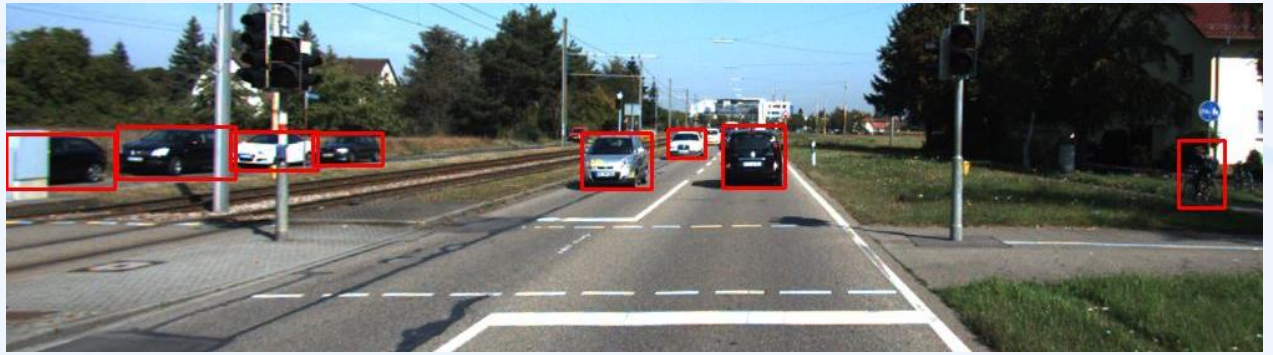
OpenMMLab

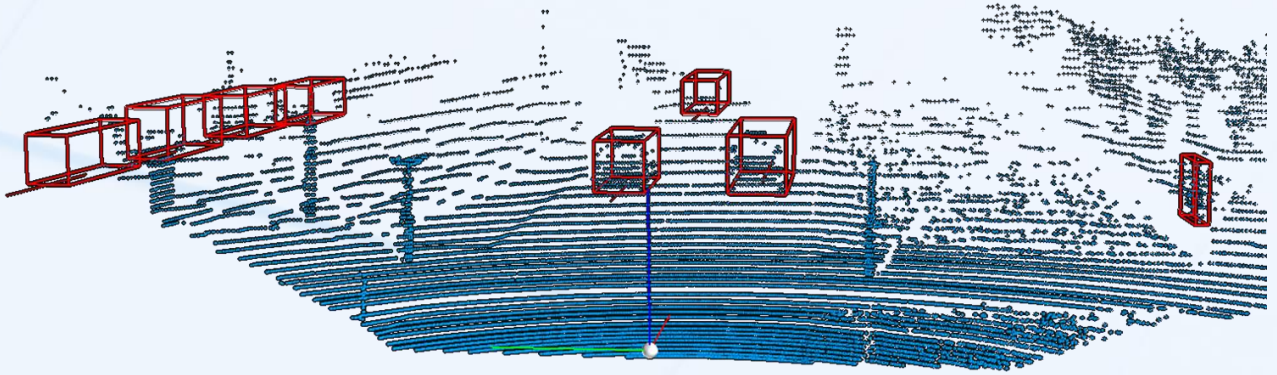


7 datasets

17 algorithms

80+ models





MMOCR

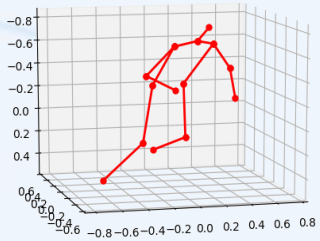
Text detection

Text recognition

Key information extraction

MMPose

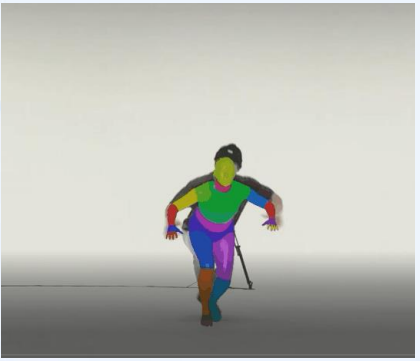
OpenMMLab



image/video

body/face/hand

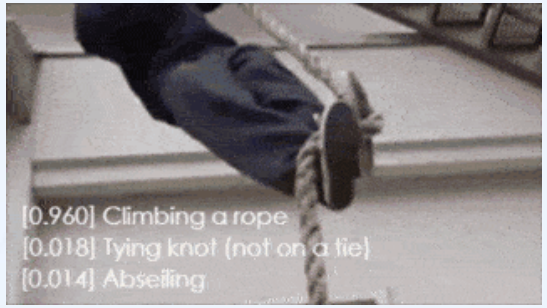
dim | **input** | **output** | **part**



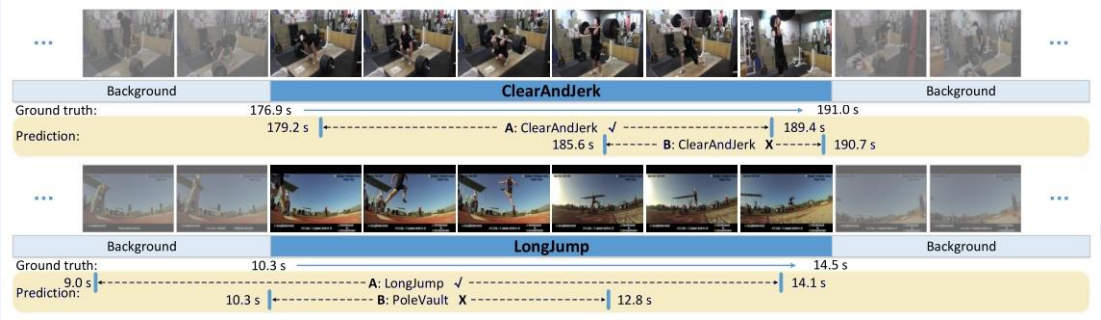
2D/3D

Mesh/keypoints

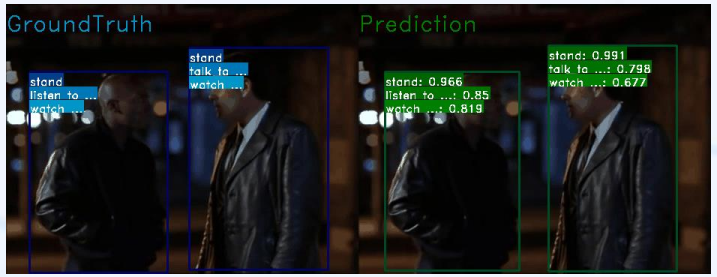
MMAction2



Action recognition



Action localization



Spatio-temporal action detection

Tasks

- recognition
- localization
- spatio-temporal

Algorithms

- **200+ models**
- 20+ algorithms

Performance

- faster training
- higher accuracy

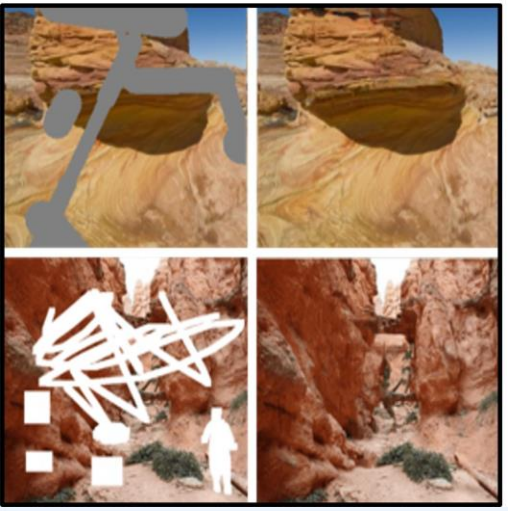
Functionality

- training
- testing
- inference api
- multiple IO backends

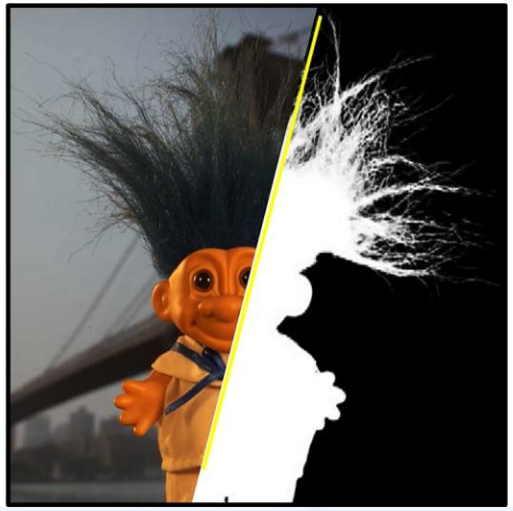
MMEditing

OpenMMLab

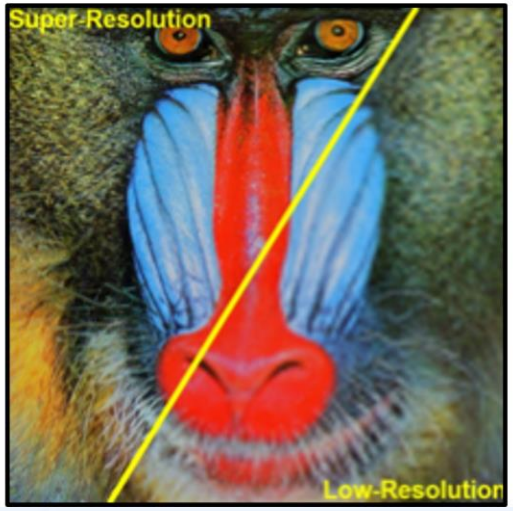
Inpainting



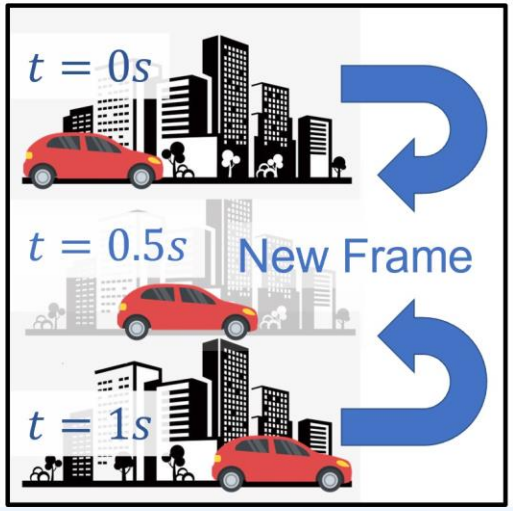
Matting



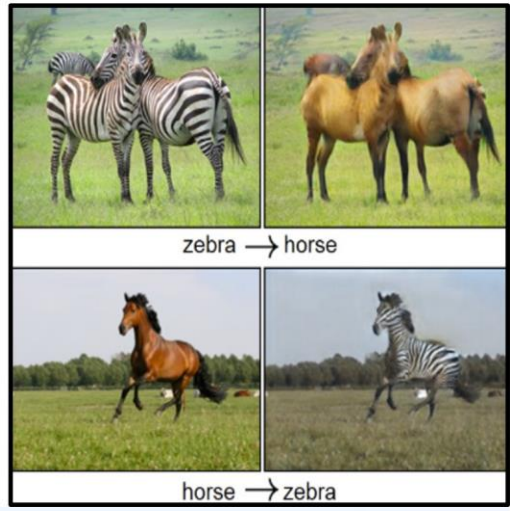
Super Resolution



Frame Interpolation



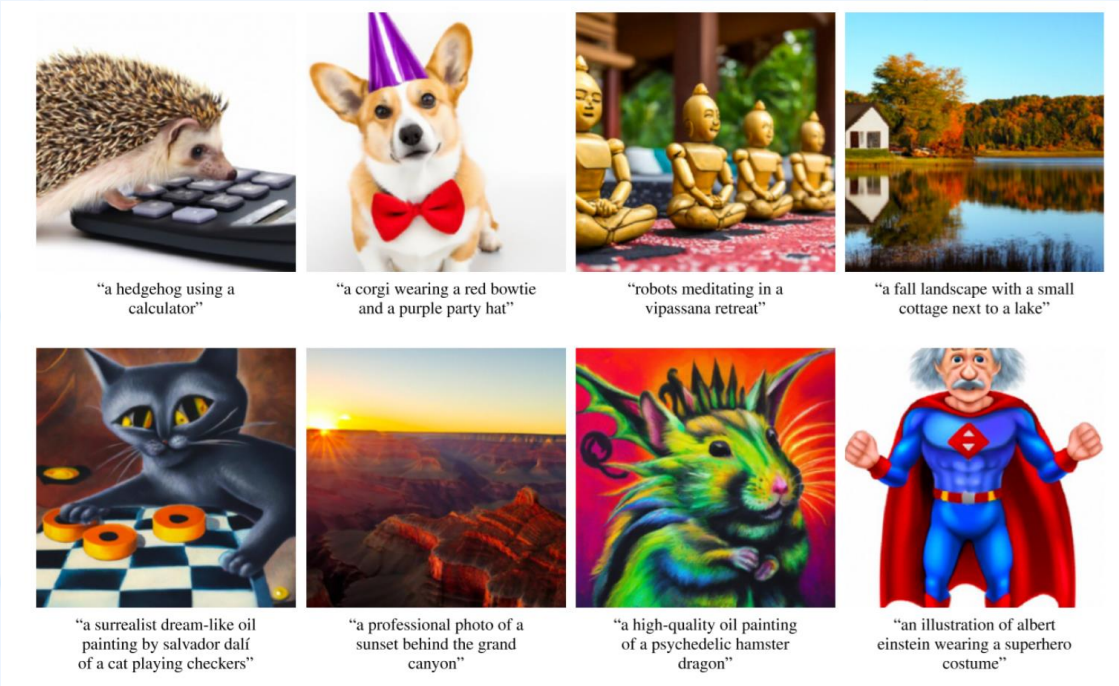
Translation



MMEditing

OpenMMLab

Text to image



3D aware generation

From Research to Production

OpenMMLab

Training Framework

Intermediate Representation (IR)

Inference Engine

PyTorch

ONNX

TorchScript

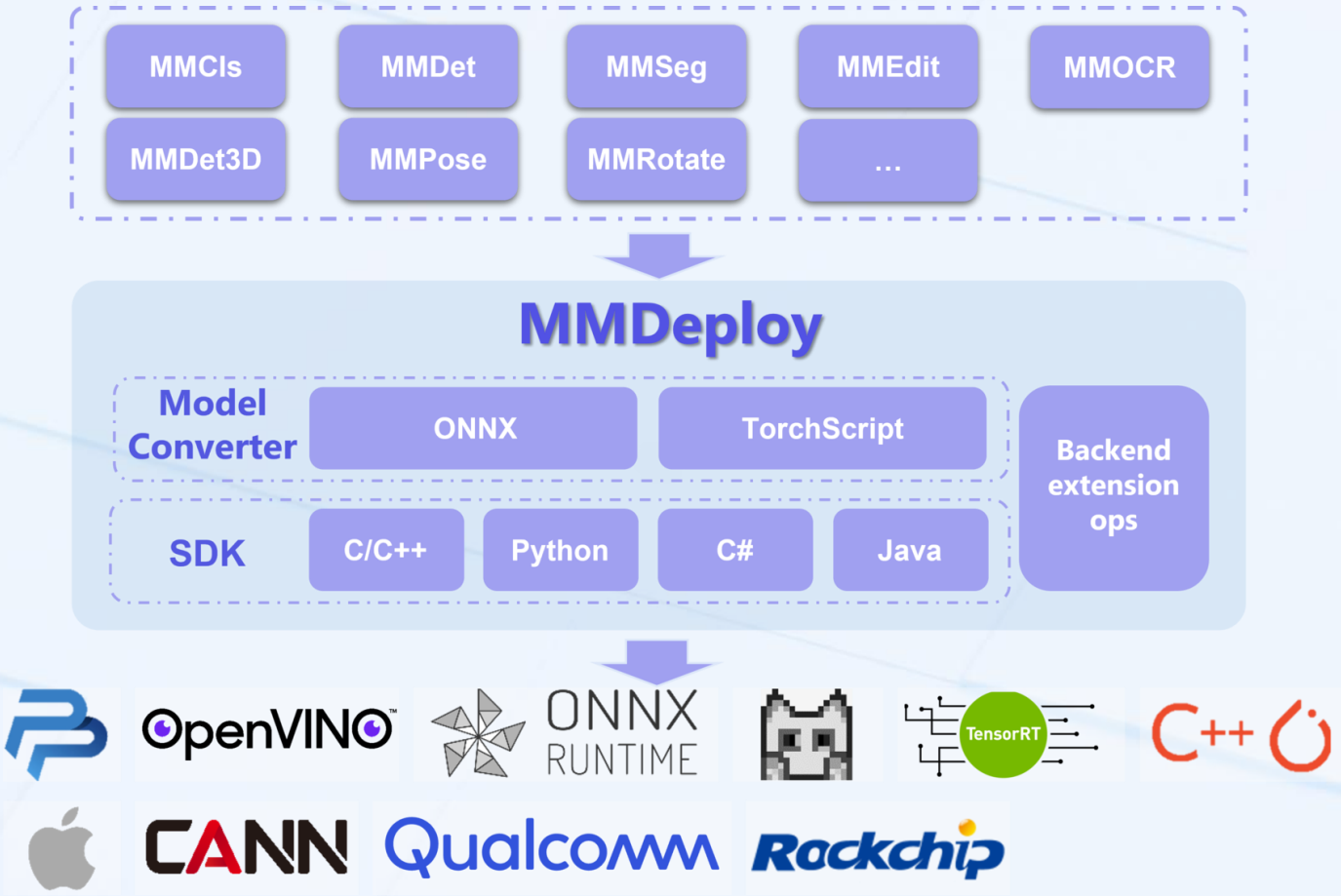
TensorRT

ONNX RUNTIME

OpenVINO™

...

MMDeploy



Various inference engines

TensorRT、ONNXRuntime、OpenVINO、ncnn、libtorch、PPL.NN

Multiple platforms

Linux、Windows、Android、macOS

Multiple language support

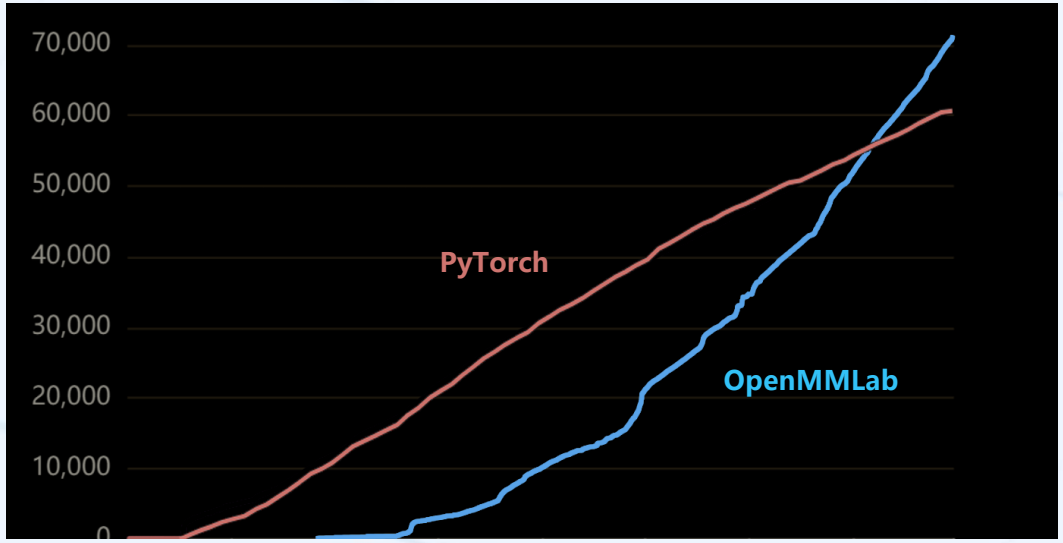
C/C++、Python、Java、C#

Flexible integration to production system

IR models、Inference engine models、MMDeploy SDK

Community Impact

Users



- 70000+ GitHub stars (overall)
- 1000+ contributors from 40+ countries

Academic

1000+ papers adopted OpenMMLab

- Swin Transformer（ICCV 2021 best paper）
- EPro-PnP（CVPR 2022 best student paper）
- BeiT（ICLR 2022, 600+ citations）
- ConvNeXt（CVPR 2022, 700+ citations）
- SegFormer（NeurIPS 2021, 800+ citations）

20+ challenge winners adopted OpenMMLab

- COCO 2018/2019
- LVIS 2021
- Waymo 2022
- NTIRE 2021/2022

Benefits of adopting OpenMMLab

Unified architecture	Learn once, use everywhere; implement once, use everywhere
Unified benchmark	Provide fair baselines for academic research
Modular design	Fast to develop and try new components
High-quality Implementation	Efficient, high performance, good code style

Basic Usage

Contents

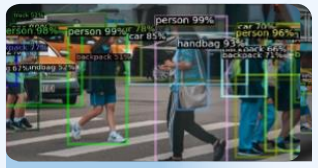
- Installation
- Inference with Pre-trained models
- Model training, testing and deployment
- Understand config files and internal mechanism
- Customization

Installation

Structures of OpenMMLab Tool System

Computer Vision Toolboxes

MMClassification



MMDetection

MMSegmentation

MMPose

MMEditing

30+ Computer Vision Toolboxes

Common Base Libraries

MMEngine — Train / Eval Engine — Module Management

MMCV — Data Augmentation — Extended NN Operators

DL Framework

PyTorch

E.g. Installing MMDetection require installing MMEngine and MMCV (also PyTorch) as dependencies.

Pure Python Packages

With C/CUDA extensions

Must match version & CUDA version of PyTorch

Compatibility of MMCV

C/CUDA extension of MMCV should be **compiled** with the **same CUDA version** of as PyTorch

To simplify this process:

1. We provides pre-built packages of MMCV
2. We developed `mim` (**M**IM **I**nstall Open**M**MLab) to detect and install the correct version of MMCV

```
# Suppose PyTorch is correctly setup
# install openmim to install other OpenMMLab packages
pip install openmim
# mim will detect the version of PyTorch and CUDA
# and download corresponding pre-built MMCV to install
mim install "mmcv>=2.0.0rc0"

# pip is unaware of the CUDA version of PyTorch
# will trigger compilation and takes lots of time
# (NOT recommended)
pip install "mmcv>=2.0.0rc0"
```

- Make sure specifying `>=2.0.0rc0` to install the new version
- MMEngine will be installed as dependencies 😉

Install Other Packages

MIM can be used to install other OpenMMLab packages and solve internal dependencies as well

```
# Option 1
# Install directly using MIM
# More convenient if you don't plan to modify the source codes
mim install "mmcls>=1.0.0rc0" "mmdet>=3.0.0rc"
```

We also recommend installing from the source

```
# Option 2
# Install from source codes
# Modifying source codes are easier
# Make sure to checkout to the version 2.0 architecture branch
git clone -b 1.x https://github.com/open-mmlab/mmclassification.git

cd mmclassification
mim install -e .          # -e: editable mode
                          # enable in-place code modification without reinstallation
```

Verify Installation

```python
import mmcv
import mmcv.ops

print(mmcv.__version__)
# Output 2.0.0rc3
print(mmcv.ops.get_compiling_cuda_version())
# Output: 11.6
# Should match the CUDA version of pytorch, otherwise cause runtime error

import mmcls
print(mmcls.__version__)
# Output: 1.0.0rc5

import mmdet
print(mmdet.__version__)
# Output: 3.0.0rc5
```

Note on the Version

Note: Up to now, OpenMMLab 2.0 toolboxes are all release candidates

Need explicitly specifying the version number, otherwise old versions will be installed

Note: OpenMMLab 2.0 is an overall name, it does NOT mean the exact version of every toolbox is 2.0

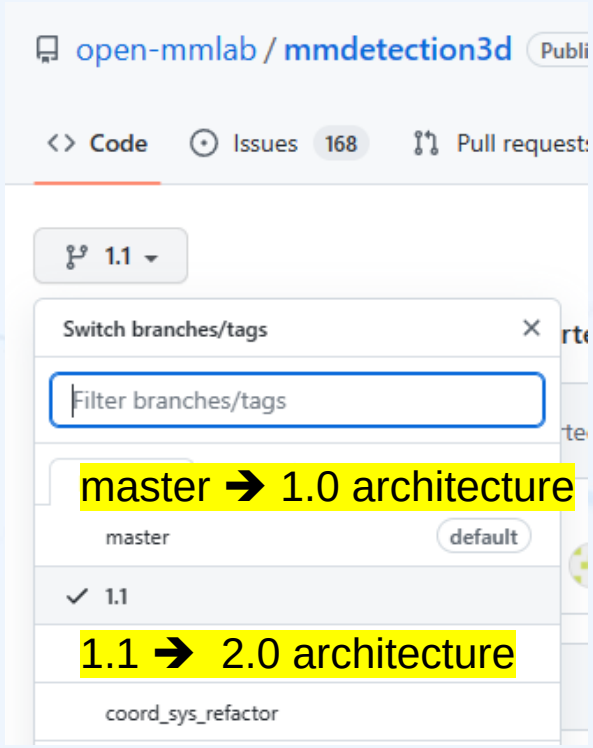
Exact version of commonly used packages are:

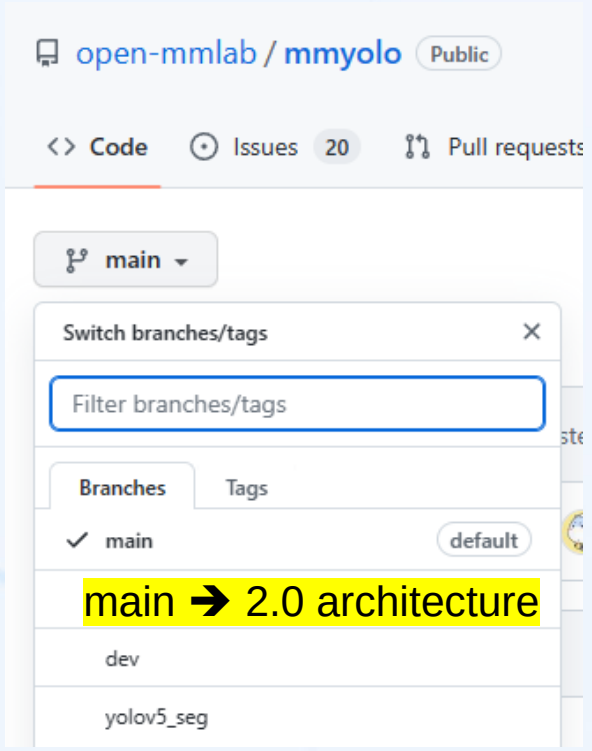
	OpenMMLab 1.0	OpenMMLab 2.0
MMEngine	N/A	mmengine
MMCV	mmcv-full	"mmcv>=2.0.0rc0"
MMClassification	mmcls	"mmcls>=1.0.0rc0"
MMDetection	mmdet	"mmdet>=3.0.0rc0"

Note on the Version

If install from source, check the correct version or branch on GitHub

- Existing toolboxes has both 1.0 and 2.0 variant
- Recently developed toolboxes has only 2.0 variant





Inference

Inference with Pre-trained Models

OpenMMLab implements 300+ CV models and provides 2300+ pre-trained models weights with SOTA performance.

4 steps for inference

1. Select a model from the **model zoo**
2. Download the **config file** and the **checkpoint file**
3. Write some Python codes to **initialize the model** and **perform inference**
4. (Optional) visualize the results

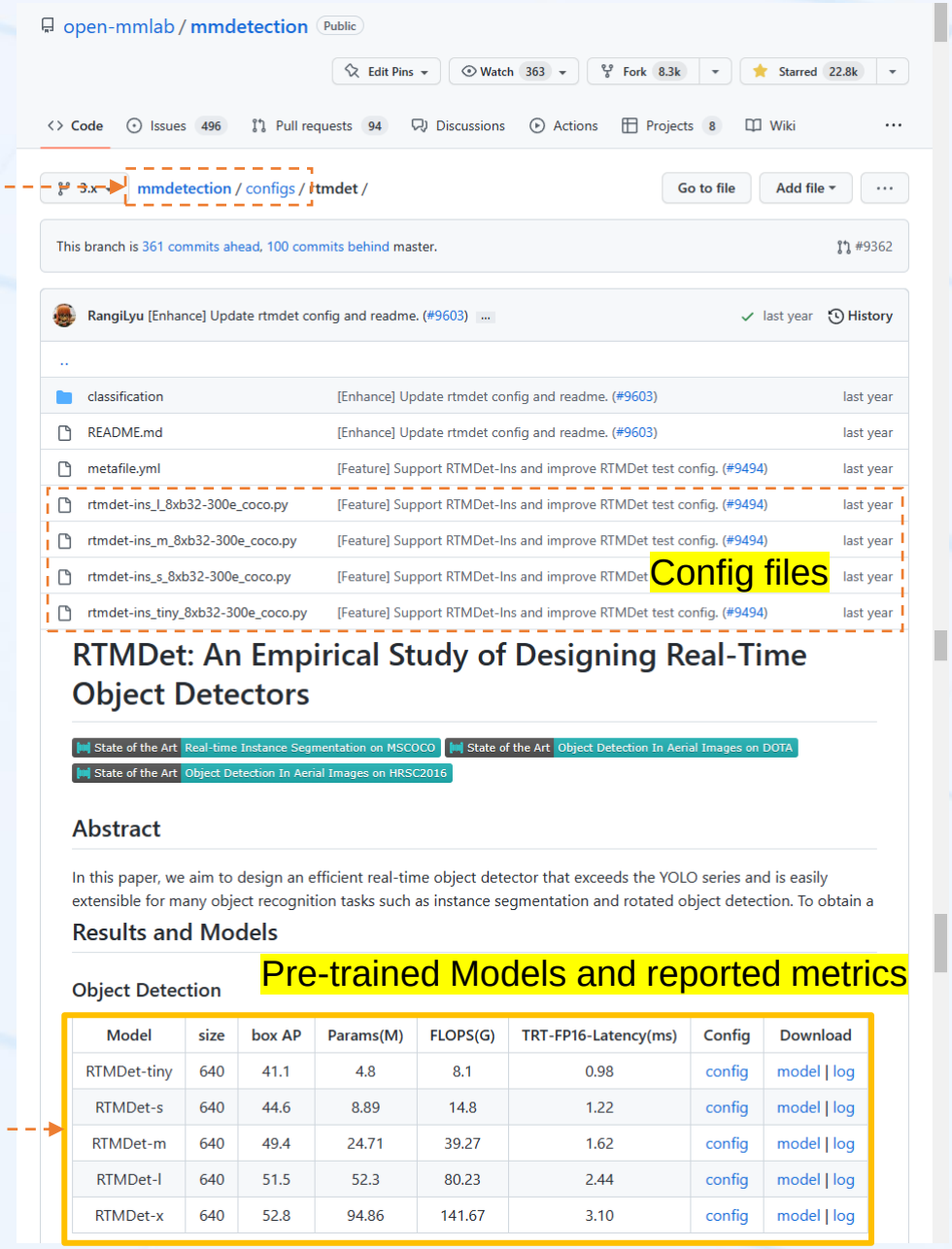
1. Select from the Model Zoo

Full list of supported models are organized under the `config` directory of each repository.

A bunch of pre-trained weights for each model

Every pre-trained weights is associated with:

- A **config file**: includes the definition of model structure
- A **checkpoint file**: storing learned weights
- Metrics and training log for reference



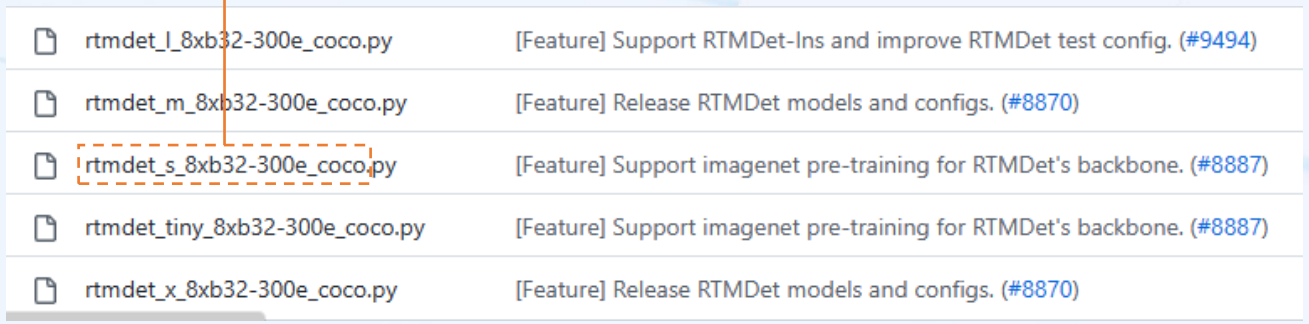
Source: https://github.com/open-mmlab/mmdetection/tree/3.x/configs/rtmdet

2. Download the Config file and Checkpoint file

Option 1: Download with MIM

```
$ mim download mmdet --config rtmdet_s_8xb32-300e_coco --dest .
```

Downloading




```
$ ls .
rtmdet_s_8xb32-300e_coco_20220905_161602-387a891e.pth
rtmdet_s_8xb32-300e_coco.py
```

Option 2: Manually from hyperlinks

Results and Models

Object Detection

Model	size	box AP	Params(M)	FLOPS(G)	TRT-FP16-Latency(ms)	Config	Download
RTMDet-tiny	640	41.1	4.8	8.1	0.98	config	model | log
RTMDet-s	640	44.6	8.89	14.8	1.22	config	model | log
RTMDet-m	640	49.4	24.71	39.27	1.62	config	model | log
RTMDet-l	640	51.5	52.3	80.23	2.44	config	model | log
RTMDet-x	640	52.8	94.86	141.67	3.10	config	model | log

3. Python API for Inference

Some Python codes to perform inference

```python
from mmdet.utils import register_all_modules
from mmdet.apis import init_detector, inference_detector

# Necessary for registries to work, will introduce details later
register_all_modules()

# model is an nn.Module object
model = init_detector('rtmdet_s_8xb32-300e_coco.py',
                      'rtmdet_s_8xb32-300e_coco_20220905_161602-387a891e.pth',
                      device='cuda:0')

# Load Image -> Preprocess -> Forward the model
# Images as numpy.array are also acceptable
result = inference_detector(model, 'bench.jpg')
```

`result` is a data structure containing all detection boxes

Applies to ALL OpenMMLab toolboxes with possible difference on function names, e.g. `init_model`.

References: https://mmdetection.readthedocs.io/en/3.x/api.html

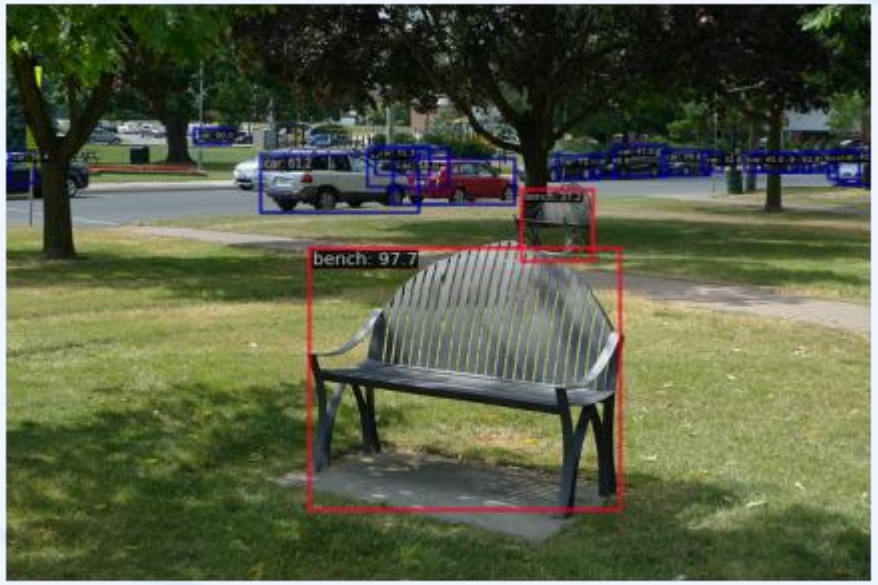
4. Visualize the result

Some toolboxes also provide utilities to visualize the results.

```
from mmdet.registry import VISUALIZERS
import mmcv

# init the visualizer(execute this only once)
visualizer = VISUALIZERS.build(model.cfg.visualizer)
visualizer.dataset_meta = model.dataset_meta
# the dataset_meta is loaded from the checkpoint and
# then pass to the model in init_detector

img = mmcv.imread('bench.jpg', channel_order='rgb')
# show the results
visualizer.add_datasample(
    'result',
    img,
    data_sample=result,
    draw_gt=False,
    wait_time=0,
)
visualizer.show()
```



Full notebook for inference detectors: https://github.com/open-mmlab/mmdetection/blob/3.x/demo/inference_demo.ipynb
More on visualization can be found on the documentation site: https://mmdetection.readthedocs.io/en/3.x/user_guides/useful_tools.html#visualization

Training

Reproduce Training

Most of pre-trained models are trained by OpenMMLab toolboxes themselves.

Only a small fraction of models are converted from other source and support inference only.

To reproduce training, one need to:

1. Prepare the dataset
2. Launch the training with the config file

E.g. ImageNet for classification, COCO for detection, small dataset such as MNIST and CIFAR can be downloaded automatically.

```
# Option 1
python tools/train.py configs/lenet/lenet5_mnist.py

# Option 2
mim train mmcls configs/lenet/lenet5_mnist.py
```

The same config we used in inference

It defines everything (not only the model) required to launch training, including the model, dataset, learning algorithm, etc.

Every toolbox provides a doc page describing how to prepare common academic datasets, e.g. for detection see https://mmdetection.readthedocs.io/en/latest/1_exist_data_model.html#prepare-datasets

Collect Results

During training, **logs** and **checkpoints** are automatically saved under a ***working directory***.

It is by default `work_dirs` under the current working directory.

Training logs & Visualization

Checkpoints

A copy of the config

```
work_dirs/lenet5_mnist
├── 20230130_092053
│   ├── 20230130_092053.log
│   └── vis_data
│       ├── 20230130_092053.json
│       ├── config.py
│       └── scalars.json
├── epoch_1.pth
├── epoch_2.pth
├── epoch_3.pth
├── epoch_4.pth
├── epoch_5.pth
├── last_checkpoint
└── lenet5_mnist.py
```

Note: If train multiple times, checkpoints are **overwitted** to save space.
Logs are separately saved with data time prepend.

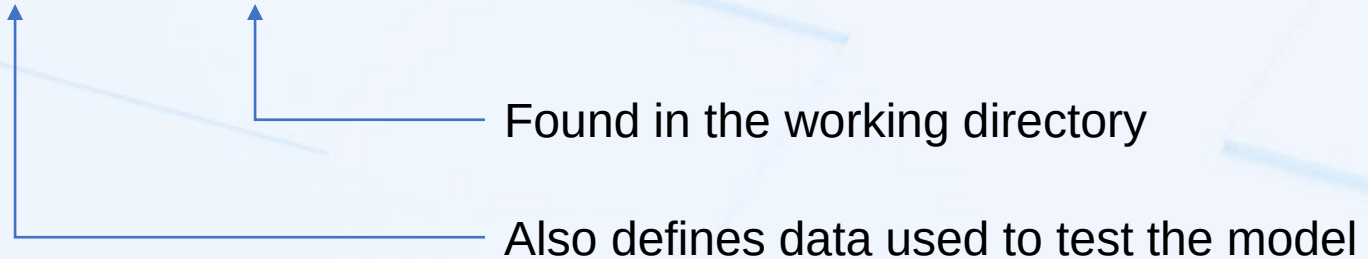
Manually specify working directories

```
python tools/train.py ${CONFIG_FILE} --work-dir ${NEW_WORK_DIR}
```

Evaluation / Testing

Once get trained model, we can evaluate the model and compute performance metrics.

```
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--out ${RESULT_FILE}] [--show]
```



Alternative: test with MIM

```
mim test mmcls ${CONFIG_FILE} --checkpoint ${CHECKPOINT_FILE} [other arguments]
```

See `python tools/test.py -h` for more available options for each toolbox

Distributed Training / Testing

Distributed training / testing

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

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

Distributed training with Slurm task manager

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

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

Reference:
https://github.com/open-mmlab/mmdetection/blob/3.x/tools/dist_train.sh
https://github.com/open-mmlab/mmdetection/blob/3.x/tools/slurm_train.sh

Distributed Training

Alternative: using MIM

```
mim train mmcls [arguments passes to tools/train.py] --gpus 4 --launcher pytorch
```

Alternative: using MIM with slurm

```
mim train mmcls [arguments passes to tools/train.py] --gpus 8 \
    --gpus-per-node 8 --partition partition_name --launcher slurm
```

Reference: https://github.com/open-mmlab/mim#command

Learning Rate Auto Scaling

The Linear Scaling Rule:

To use difference number of GPUs, one need to scale learning rate proportional to total batch size

[Total batch size] = [Batch size on single GPU] × [Number of GPUs]

Most provided configs are tuned on 8 GPU servers.

Need to tune the learning rate based on actual number of GPUs.

We support automatic learning rate scaling the with one additional argument

```
python tools/train.py ${CONFIG_FILE} --auto-scale-lr [optional arguments]
```

Reference
https://mmdetection.readthedocs.io/en/3.x/user_guides/train.html?highlight=learning%20rate%20auto%20scaling#learning-rate-auto-scaling

More Options for Training

Resume training from existing checkpoints

```
python tools/train.py ${CONFIG_FILE} --resume work_dirs/lenet5_mnist/epoch_4.pth
```

Automatic mixed precision training

```
python tools/train.py ${CONFIG_FILE} --amp
```

Check `python tools/train.py -h` for more available options for each toolbox

Dataset Customization

It is very common to train models based on users own dataset.

We offer two options to accomplish this:

Option 1: Convert the dataset offline

- Convert the dataset to a standard format, e.g. ImageNet format for classification or COCO format for detection.
- Easier, existing codes work on organized data
- No modification to source codes, only some modification on dataset meta (e.g. class names) in config files.

Detailed example on Option 1: https://mmdetection.readthedocs.io/en/3.x/user_guides/train.html#train-with-customized-datasets

Dataset Customization

It is very common to train models based on users own dataset.

We offer two options to accomplish this:

Option 2: Implement a new dataset class

- Requires to write some codes to read and parse your dataset.
- More modification to the config file.
- Generally more efficient, no requirement on offline storage.

More on Option 2: https://mmdetection.readthedocs.io/en/3.x/advanced_guides/customize_dataset.html

Model Deployment

For production purpose, deploy model to target device to make computation more efficient

MMDeploy toolbox supports convert model to different backends.

Example: The following command convert a YOLO v3 model to TensorRT backend.

```
python ./tools/deploy.py \
    configs/mmdet/detection/detection_tensorrt_dynamic-320x320-1344x1344.py \
    $PATH_TO_MMDET/configs/yolo/yolov3_d53_mstrain-608_273e_coco.py \
    $PATH_TO_MMDET/checkpoints/yolo/yolov3_d53_mstrain-608_273e_coco.pth \
    $PATH_TO_MMDET/demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cuda:0
```

A deployment config

Config and checkpoints of a model

Reference: https://mmdeploy.readthedocs.io/en/latest/02-how-to-run/convert_model.html

MMDeploy Usage Example

```
python tools/deploy.py \
    mmdeploy/configs/mmdet/detection/detection_tensorrt_dynamic-320x320-1344x1344.py \
    mmdetection/configs/retinanet/retinanet_r50_fpn_1x_coco.py \
    retinanet_r50_fpn_1x_coco_20200130-c2398f9e.pth \
    mmdetection/demo/demo.jpg \
    --work-dir retinanet/tensorrt \
    --device cuda:0 \
    --dump-info
```

RetinaNet


```
retinanet/tensorrt
├── deploy.json
├── detail.json
├── end2end.engine
├── end2end.onnx
├── output_pytorch.jpg
├── output_tensorrt.jpg
└── pipeline.json
```

```python
from mmdeploy_python import Detector
import cv2

model_path='retinanet/tensorrt'
image_path='mmdetection/demo/demo.jpg'
img = cv2.imread(image_path)
detector = Detector(model_path=model_path,
                    device_name='cuda',
                    device_id=0)
bboxes, labels, _ = detector(img)
```

Config Files

Components of a Config File

A **Config file** contains a bunch of key-value pairs, organized in python format.

It defines all items required to train and/or test a model.

4 major parts:

- **Model** structure
- **Dataset**, data augmentation and dataloader
- **Optimizer** and learning rate policies
- **Runtime** configs
  - Working directory, logs, checkpoints, visualizations
  - Multi-processing / distributed training environment
  - Random seeds

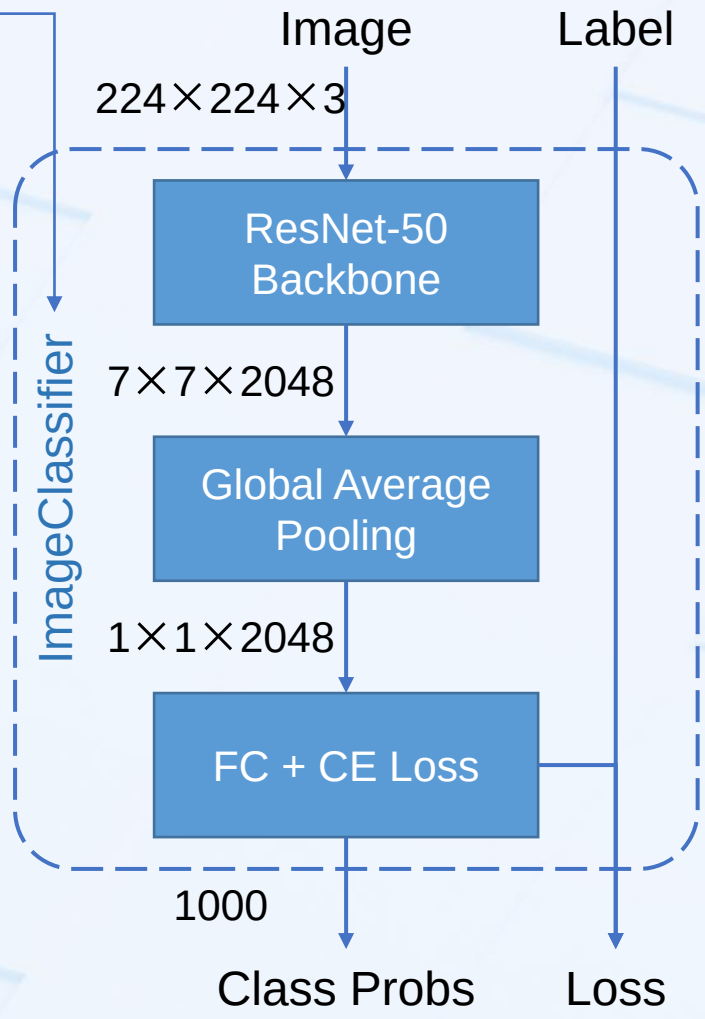
Config files in OpenMMLab are usually in `py` format, while `json/yaml` are also supported.

The Model

```
model = dict(
    type='ImageClassifier',
    backbone=dict(
        type='ResNet',
        depth=50,
        num_stages=4,
        out_indices=(3, ),
        style='pytorch'),
    neck=dict(type='GlobalAveragePooling'),
    head=dict(
        type='LinearClsHead',
        num_classes=1000,
        in_channels=2048,
        loss=dict(type='CrossEntropyLoss', loss_weight=1.0),
        topk=(1, 5),
    ))
```

The **type** key specify the type of the model

Nested key-value pairs define submodules in the model.



Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/models/resnet50.py

Dataset and Dataloader

```python
train_dataloader = dict(
    batch_size=32,
    num_workers=5,
    dataset=dict(
        type='ImageNet',
        data_root='data/imagenet',
        ann_file='meta/train.txt',
        data_prefix='train',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='RandomResizedCrop', scale=224),
            dict(type='RandomFlip', prob=0.5, direction='horizontal'),
            dict(type='PackClsInputs'),
        ]),
    sampler=dict(type='DefaultSampler', shuffle=True),
)
```

Also validation and test loaders

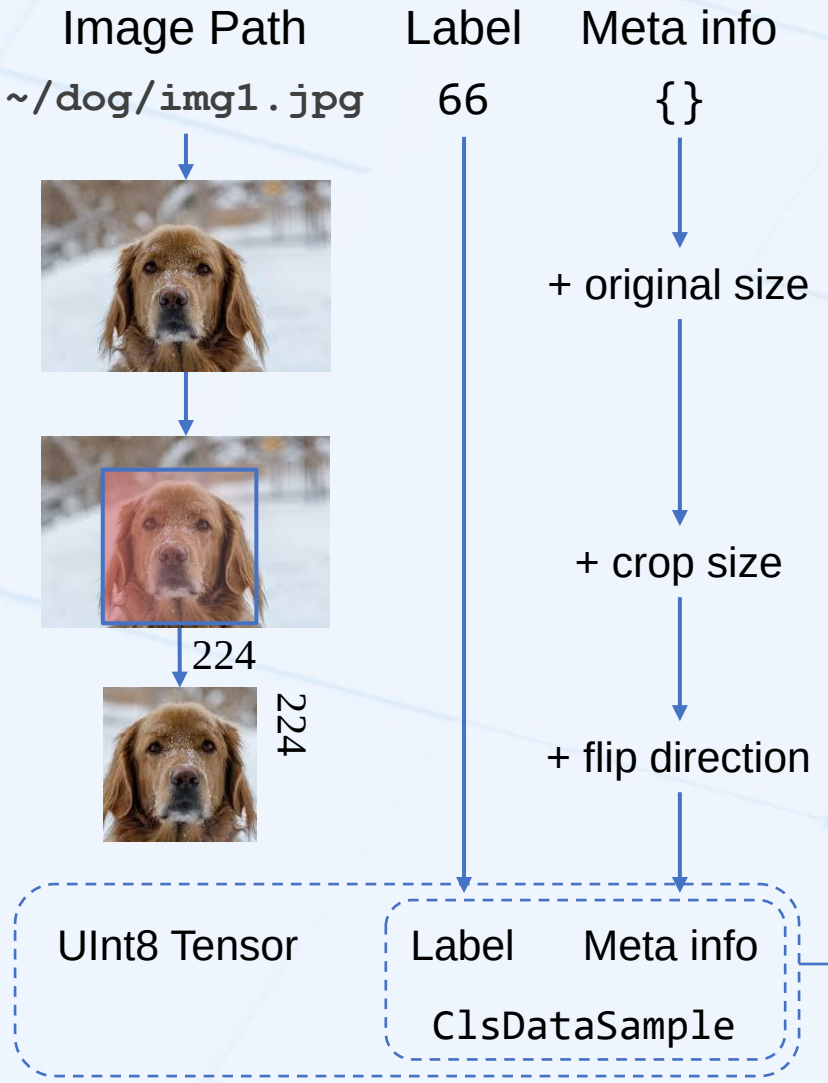
Dataset subkey specifies data paths and a pre-processing pipeline

```
mmclassification
├── mmcls
├── tools
├── configs
├── docs
├── data
│   ├── imagenet
│   │   ├── meta
│   │   ├── train
│   │   ├── val
```

Arguments are aligned with PyTorch

Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/datasets/imagenet_bs32.py
For dataset preparation, check the doc site: https://mmclassification.readthedocs.io/en/1.x/user_guides/dataset_prepare.html

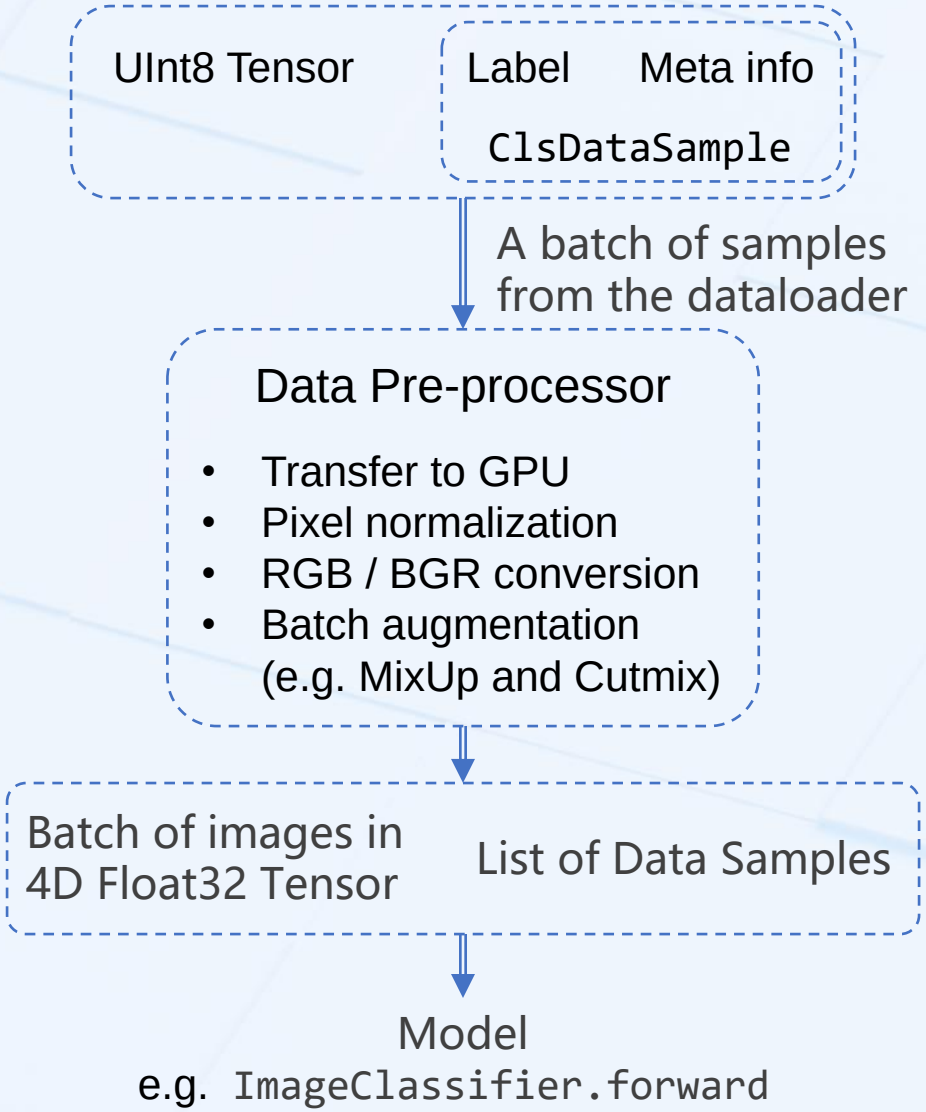
Data Pre-processing Pipeline (within dataloader)




```python
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='RandomResizedCrop', scale=224),
    dict(type='RandomFlip', prob=0.5, direction='horizontal'),
    dict(type='PackClsInputs'),
]
```

Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/datasets/imagenet_bs32.py

Data Pre-processing Pipeline (from dataloader to model)




```python
data_preprocessor = dict(
    num_classes=1000,
    # RGB format normalization parameters
    mean=[123.675, 116.28, 103.53],
    std=[58.395, 57.12, 57.375],
    # convert image from BGR to RGB
    to_rgb=True,
)
```

Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/datasets/imagenet_bs32.py

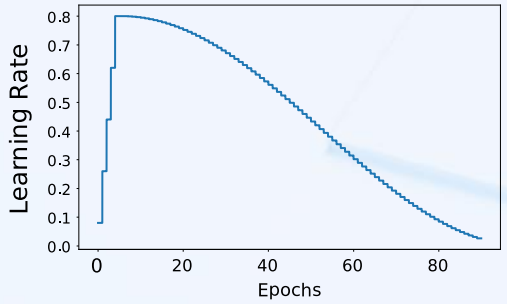
Optimizer and Learning Rate Schedulers

```python
# optimizer
optim_wrapper = dict(
    optimizer=dict(type='SGD', lr=0.8, momentum=0.9, weight_decay=5e-5))

# learning policy
param_scheduler = [
    dict(type='LinearLR', start_factor=0.1, by_epoch=True, begin=0, end=5),
    dict(type='CosineAnnealingLR', T_max=95, by_epoch=True, begin=5, end=100)
]

# train, val, test setting
train_cfg = dict(by_epoch=True, max_epochs=100, val_interval=1)
val_cfg = dict()
test_cfg = dict()

# NOTE: `auto_scale_lr` is for automatically scaling LR,
# based on the actual training batch size.
auto_scale_lr = dict(base_batch_size=1024)
```



Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/schedules/imagenet_bs1024_coslr.py

Runtime Configs

```python
# configure default hooks
default_hooks = dict(
    # record the time of every iteration.
    timer=dict(type='IterTimerHook'),

    # print log every 100 iterations.
    logger=dict(type='LoggerHook', interval=100),

    # enable the parameter scheduler.
    param_scheduler=dict(type='ParamSchedulerHook'),

    # save checkpoint per epoch.
    checkpoint=dict(type='CheckpointHook', interval=1),

    # set sampler seed in distributed evrionment.
    sampler_seed=dict(type='DistSamplerSeedHook'),

    # validation results visualization, set True to enable it.
    visualization=dict(type='VisualizationHook', enable=False),
)
```

Interval of outputting log and saving checkpoints

The unit of interval can be epoch or iteration, by setting `by_epoch=True/False`

We will cover more on hooks later.

Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/default_runtime.py

Runtime Configs

```python
# set visualizer
vis_backends = [dict(type='LocalVisBackend')]
visualizer = dict(type='ClsVisualizer', vis_backends=vis_backends)

# set log level
log_level = 'INFO'

# load from which checkpoint
load_from = None

# whether to resume training from the loaded checkpoint
resume = False

# Defaults to use random seed and disable `deterministic`
randomness = dict(seed=None, deterministic=False)
```

Initialize model from other pre-trained weights
No optimizer states

Resume from checkpoints (e.g. interrupted accidentally)
Recover optimizer state too

Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/default_runtime.py

Runtime Configs

```python
# configure environment
env_cfg = dict(
    # whether to enable cudnn benchmark
    cudnn_benchmark=False,

    # set multi process parameters
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),

    # set distributed parameters
    dist_cfg=dict(backend='nccl'),
)
```

Items configuring distributed environment

Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/default_runtime.py

Summary: 4 major parts of a config file

A Config file contains all items required to define a specific **experiment**, including 4 major parts:

- The structure of the model
- Data
  - Dataset used for training / validating / testing the model
  - Data pre-processing pipelines and data pre-processor
  - Dataloader parameters such as batch size, prefetch workers, etc.
- Optimizer and learning rate policies
- Runtime configs
  - Logs, checkpoints, visualizations
  - multi-processing / distributed training parameters
  - Random seeds

Using Intermediate Variable in Configs

```python
train_dataloader = dict(
    batch_size=32,
    num_workers=5,
    dataset=dict(
        type='ImageNet',
        data_root='data/imagenet',
        ann_file='meta/train.txt',
        data_prefix='train',
        pipeline=train_pipeline),
    sampler=dict(type='DefaultSampler', shuffle=True),
)
```

When written in **Python**, config files support reusing **variable** just like writing Python scripts.

- More clear file structure
- Reuse same items

```python
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='RandomResizedCrop', scale=224),
    dict(type='RandomFlip', prob=0.5, direction='horizontal'),
    dict(type='PackClsInputs'),
]
```

Source: https://github.com/open-mmlab/mmclassification/blob/1.x/configs/_base_/datasets/imagenet_bs32.py
For dataset preparation, check the doc site: https://mmclassification.readthedocs.io/en/1.x/user_guides/dataset_prepare.html

Inheritance of Config

Imagine training the same model using different optimizers.

The model part shares, but rewriting every item is obviously redundant.

A minimal example

resnet50.py:

```
model = dict(type='ResNet', depth=50)
optimizer = dict(type='SGD', lr=0.01)
```

optimizer_cfg.py

```
_base_ = ['resnet50.py']
optimizer = dict(type='Adam')
```

Equivalent to a merged config

Items already in _base_ will be updated

optimizer_cfg.py

```
optimizer = dict(type='Adam', lr=0.02)
model = dict(type='ResNet', depth=50)
```

Reference: https://mmengine.readthedocs.io/en/latest/advanced_tutorials/config.html

Inheritance of Config

With inheritance, configs in OpenMMLab toolboxes are usually organized as a series of **base configs** and experiment configs inherited from these base configs.

Base Configs

```
mmclassification/configs/_base_
├──datasets
├──models
├──schedules
└──default_funtime.py
```

Config for specific experiment

mmclassification/configs/resnet/resnet50_8xb32_in1k.py

```
_base_ = [
    '../_base_/models/resnet50.py',
    '../_base_/datasets/imagenet_bs32.py',
    '../_base_/schedules/imagenet_bs256.py',
    '../_base_/default_runtime.py'
]

(If any) additional items
(If any) modification of base configs
```

Reference: https://github.com/open-mmlab/mmclassification/tree/1.x/configs

Summary

A Config file contains all items required to define an **experiment**, including 4 major parts:

- The structure of the model
- Data
- Optimizer and learning rate policies
- Runtime configs

- Config files in Python support intermediate variables
- Config files support inheritance, toolboxes organize base configs under `config/_base_` directory

Internal Mechanism

How Config Works

The config

```python
model = dict(
    type='ImageClassifier',
    backbone=dict(
        type='ResNet',
        depth=50,
        num_stages=4,
        out_indices=(3, ),
        style='pytorch'),
    neck=dict(type='GlobalAveragePooling'),
    head=dict(
        type='LinearClsHead',
        num_classes=1000,
        in_channels=2048,
        loss=dict(type='CrossEntropyLoss', loss_weight=1.0),
        topk=(1, 5),
    ))
```

from str to type

pass as arguments

The actual implementation in codebase

mmcls/models/classifiers/image.py

```python
class ImageClassifier(BaseClassifier):
    def __init__(self,
                 backbone: dict,
                 neck: Optional[dict] = None,
                 head: Optional[dict] = None,
                 pretrained: Optional[str] = None,
                 train_cfg: Optional[dict] = None,
                 data_preprocessor: Optional[dict] = None,
                 init_cfg: Optional[dict] = None):
```

Nested modules will be recursively instantiated within `ImageClassifier`.

How Config Works

```python
model = dict(
    type='ImageClassifier',
    backbone=dict(
        type='ResNet',
        depth=50,
        num_stages=4,
        out_indices=(3, ),
        style='pytorch'),
    neck=dict(type='GlobalAveragePooling'),
    head=dict(
        type='LinearClsHead',
        num_classes=1000,
        in_channels=2048,
        loss=dict(type='CrossEntropyLoss', loss_weight=1.0),
        topk=(1, 5),
    ))
```

Instantiate recursively in `ImageClassifier`

```python
from mmcls.models.resnet import ResNet

backbone = ResNet(
        depth=50,
        num_stages=4,
        out_indices=(3, ),
        style='pytorch'
)
```

Config and Registry

`Registry` is essentially a mapping from class name to the class itself.

`MODEL` is a specific registry containing all models. There are also other registries.

Model definition in codebase

```
mmcls/models/resnet.py

@MODELS.register_module
class ResNet(nn.module):
    def __init__(self, depth,**kwargs)
…
```

Registered in the Registry

```
MODELS Registry: Mapping[str, type]

{
   "ResNet": mmcls.models.resnet.ResNet
   "mobilenet":…
}
```

Fetch the class by type name
Instantiate based on arguments

```
from mmengine.registry import MODELS

model = MODELS.build({
   "type": "ResNet"
   "depth": 50})
# This is equivalent to
model = ResNet(depth=50)
```

Note: To make the registry work, we need to register all classes into the registry. This is why we need to call `register_all_modules()` in inference part.

Full list of registries in MMEngine: https://mmengine.readthedocs.io/en/latest/advanced_tutorials/registry.html#advanced-usage

Use Component from Other Toolboxes

Registry allows using modules from other toolboxes.

A config in MMDetection

```
model = dict(
    backbone=dict(
        _delete_=True,
        type='mmcls.ConvNeXt',
        arch='tiny',
        out_indices=[0, 1, 2, 3],
        drop_path_rate=0.4,
        layer_scale_init_value=1.0,
        gap_before_final_norm=False,
        init_cfg=dict(
            type='Pretrained', checkpoint=checkpoint_file,
            prefix='backbone.')),
    neck=dict(in_channels=[96, 192, 384, 768]))
```

This backbone actually implemented in MMClassification

configs/convnext/mask-rcnn_convnext-t-p4-w7_fpn_amp-ms-crop-3x_coco.py

The Runner ---- The Skeleton of the Program

A typical PyTorch program

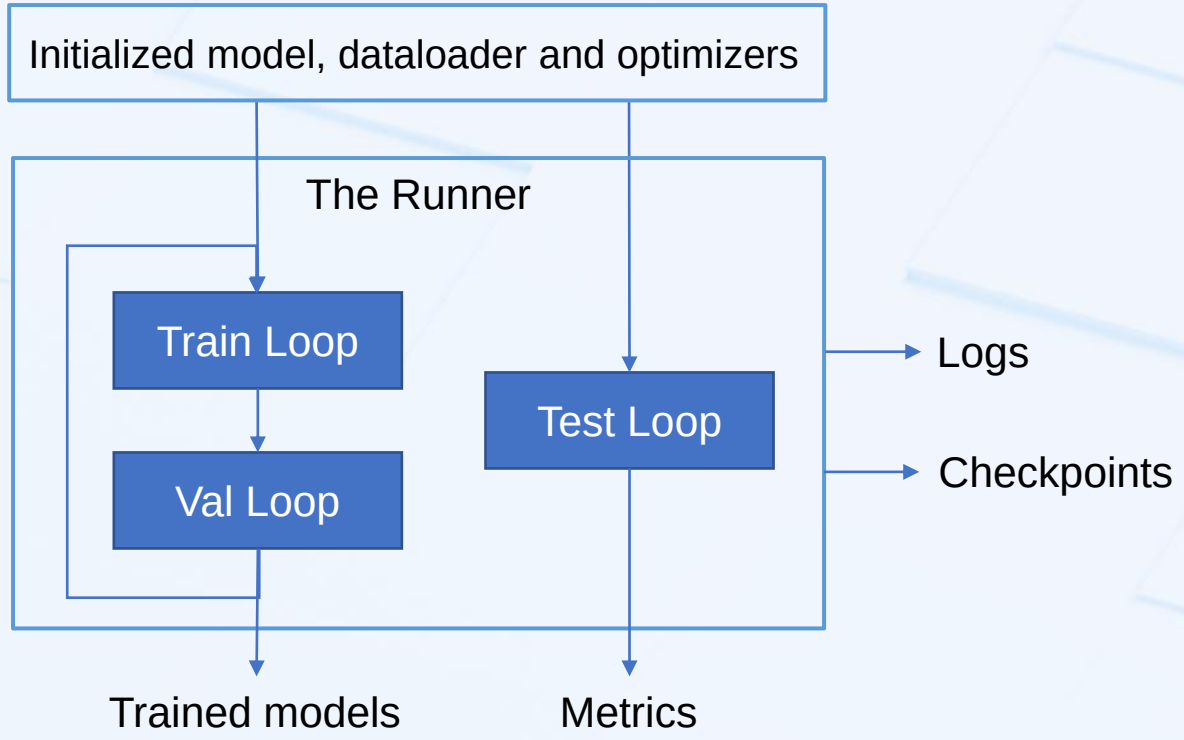
```
# First constructed model,
# dataloaders and optimizer
for _ in range(epochs):
    for data in train_loader:
        loss = model(*data)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    lr_scheduler.step()

    with torch.no_grad():
        for data in val_loader:
            pred = model(*data)
        metric = some_stats_of_all_preds

with torch.no_grad():
    for data in test_loader:
        pred = model(*data)
        metric = some_stats_of_all_preds
```

Abstract

The Runner in MMEngine



Decouple the framework and specific models and datasets:

- MMEngine implements the Runner
- Downstream toolboxes focus on models, data, etc.

More on Runner:
https://mmengine.readthedocs.io/en/latest/design/runner.html
https://github.com/open-mmlab/mmengine/blob/main/mmengine/runner/runner.py

The Runner ---- The Skeleton of the Program

- The runner is essentially the *main* function of the program
- The config file specify the whole argument list to initialize the runner

The config file

```python
class Runner:
    def __init__(
        self,
        model: Union[nn.Module, Dict],
        work_dir: str,
        train_dataloader: Optional[Union[DataLoader, Dict]] = None,
        val_dataloader: Optional[Union[DataLoader, Dict]] = None,
        test_dataloader: Optional[Union[DataLoader, Dict]] = None,
        train_cfg: Optional[Dict] = None,
        val_cfg: Optional[Dict] = None,
        test_cfg: Optional[Dict] = None,
        auto_scale_lr: Optional[Dict] = None,
        optim_wrapper: Optional[Union[OptimWrapper, Dict]] = None,
        param_scheduler: Optional[Union[_ParamScheduler, Dict, List]] = None,
        val_evaluator: Optional[Union[Evaluator, Dict, List]] = None,
        test_evaluator: Optional[Union[Evaluator, Dict, List]] = None,
        default_hooks: Optional[Dict[str, Union[Hook, Dict]]] = None,
        custom_hooks: Optional[List[Union[Hook, Dict]]] = None,
        data_preprocessor: Union[nn.Module, Dict, None] = None,
        load_from: Optional[str] = None,
        resume: bool = False,
        launcher: str = 'none',
        env_cfg: Dict = dict(dist_cfg=dict(backend='nccl')),
        log_processor: Optional[Dict] = None,
        log_level: str = 'INFO',
        visualizer: Optional[Union[Visualizer, Dict]] = None,
        default_scope: str = 'mmengine',
        randomness: Dict = dict(seed=None),
        experiment_name: Optional[str] = None,
        cfg: Optional[ConfigType] = None,
    ):
```

More on Runner:
https://mmengine.readthedocs.io/en/latest/design/runner.html
https://github.com/open-mmlab/mmengine/blob/main/mmengine/runner/runner.py

Hooks

Hook = insert functions at specific locations of a program

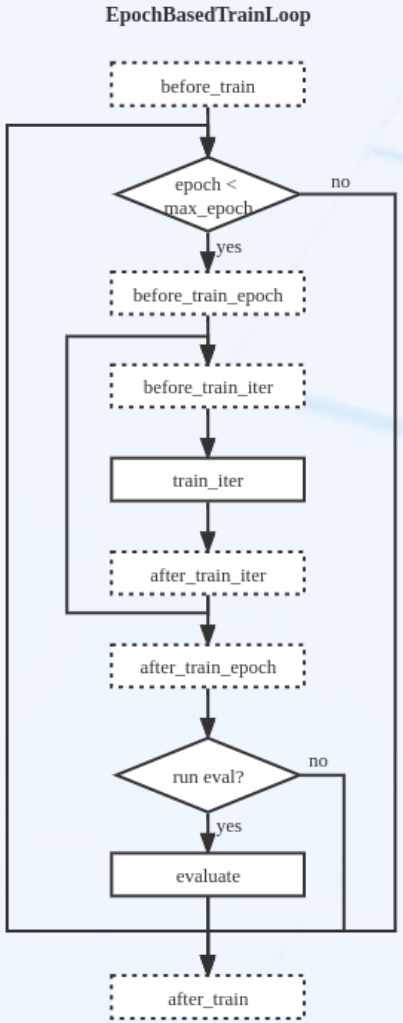
The runner support 20+ hook locations for users to customize.

Hook functions take the whole runner as argument, thus is very flexible.

MMEngine implements some default hooks, **for example**:

- `CheckpointHook` will be executed at `after_train_epoch` location. It fetch the model and the oprimizer from the runner and save their `state_dict` to disks.

User can also define their own hooks and set them into configs to make them work



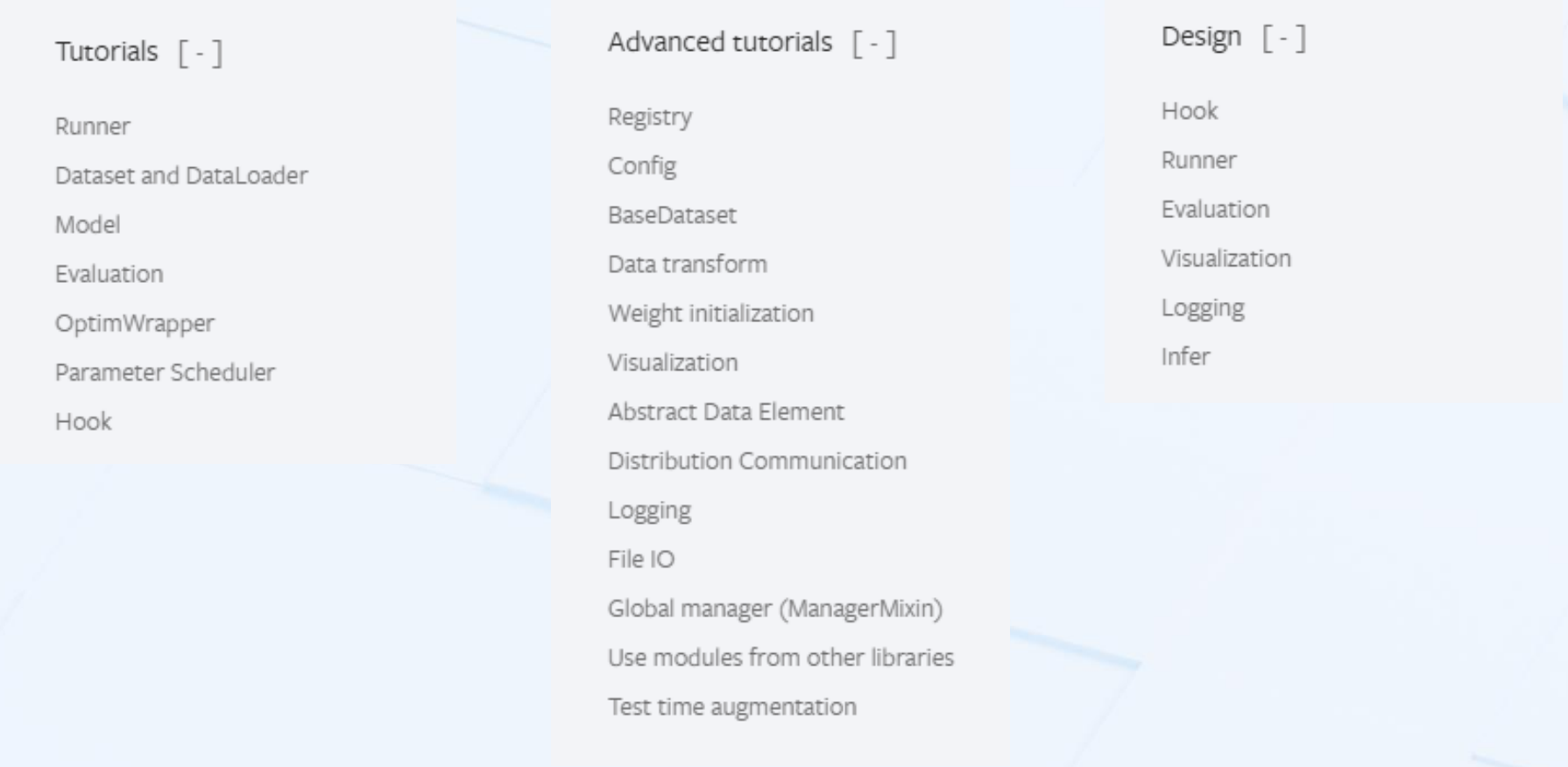
Hook points at EpochBaseTrainLoop

Reference: https://mmengine.readthedocs.io/en/latest/tutorials/hook.html
https://mmdetection.readthedocs.io/en/3.x/user_guides/useful_hooks.html?highlight=hook#how-to-implement-a-custom-hook

Learn More

Learn more internal design at MMEngine's documentation site:

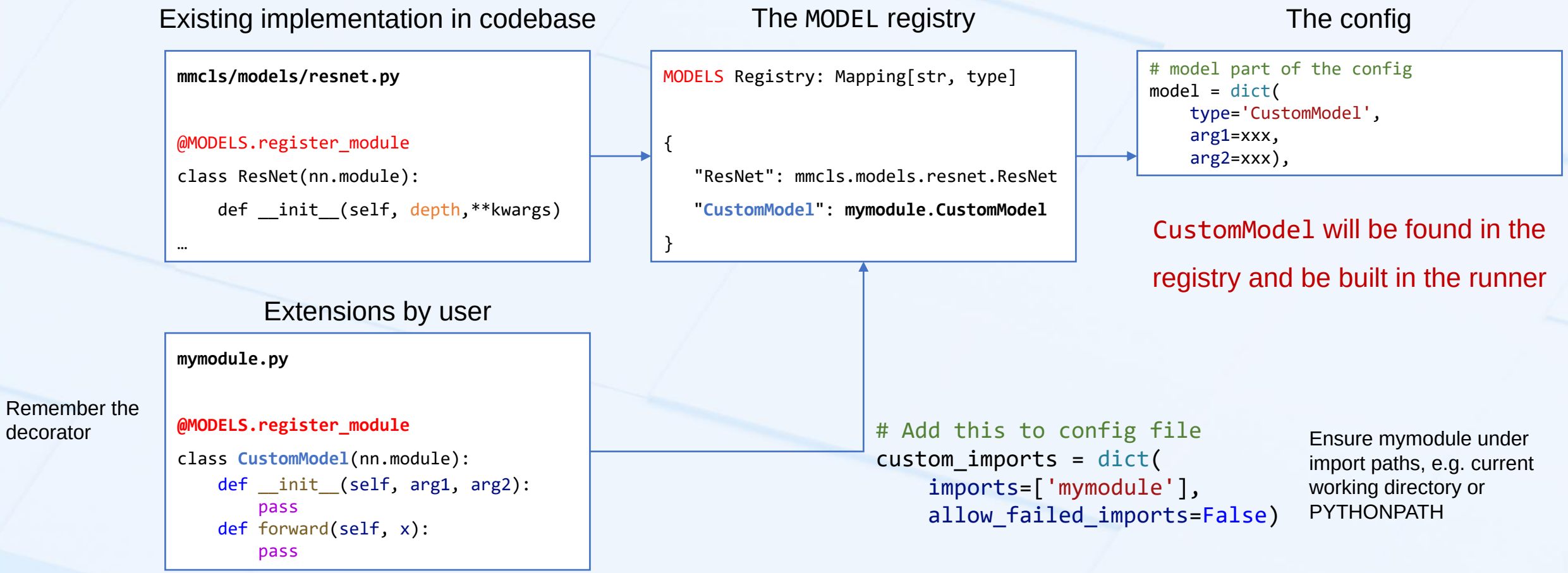
https://mmengine.readthedocs.io/en/latest/index.html



Customization

General Idea

With registry, we can implement customized modules without modifying the original code base.



Detailed Examples

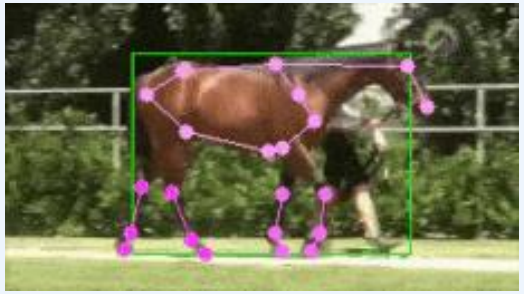
Check our documentation site for more concrete example on customizing model, dataset, pipelines, optimizers, hooks and everything!

https://mmdetection.readthedocs.io/en/3.x/advanced_guides/index.html#component-customization

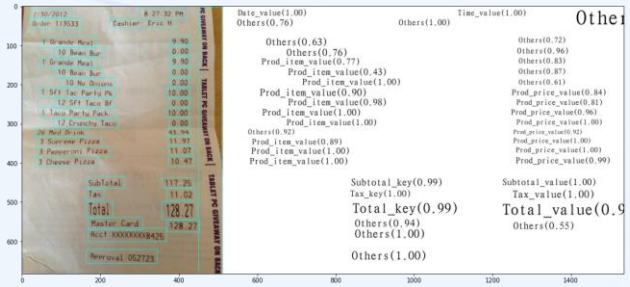
More Examples and Tutorials

Check the `demo` folders of each toolbox for more examples!

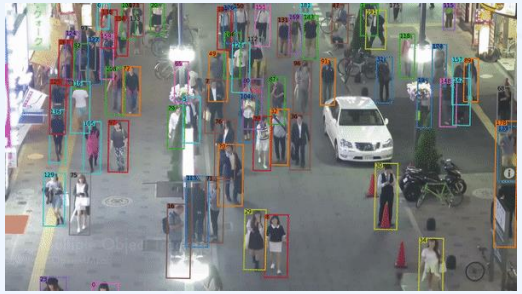
Remember to checkout to the new branch



Animal pose

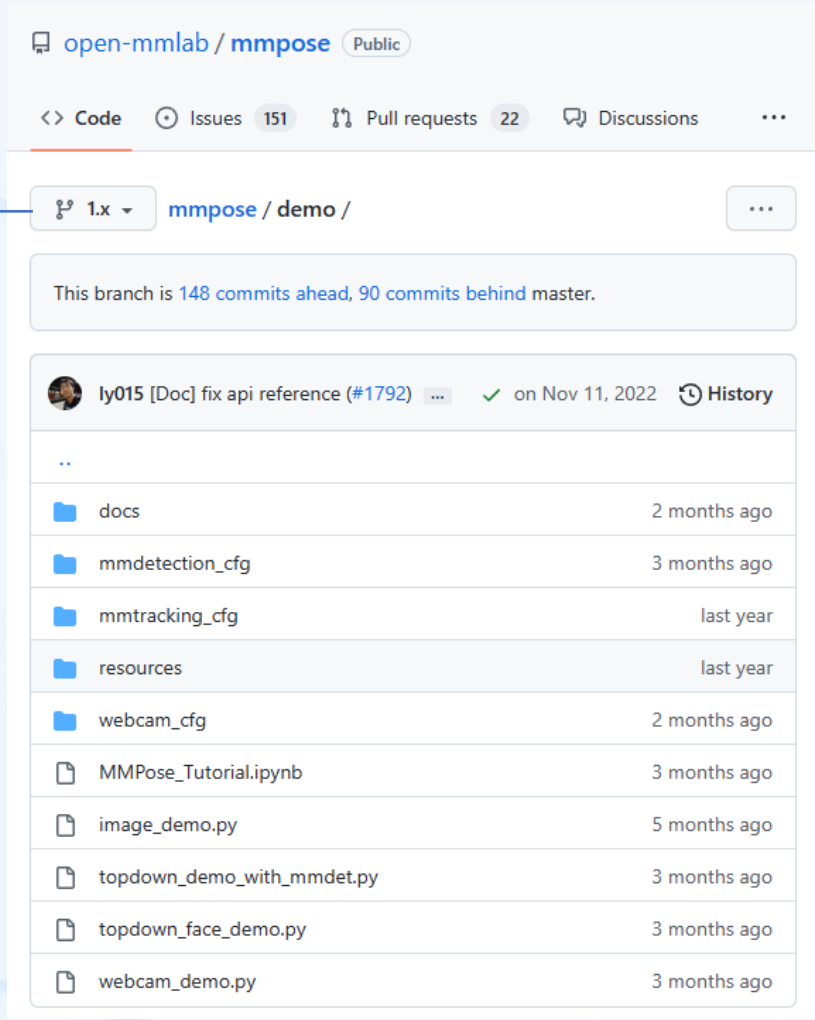


Key information extraction



Tracking

Video super resolution



Demos from MMPose, MMTracking, MMOCR and MMEditing

Thank you! Q&A