MMYOLO

Release 0.6.0

MMYOLO Authors

Aug 24, 2023

GET STARTED

You can switch between Chinese and English documents in the top-right corner of the layout.

CHAPTER

ONE

OVERVIEW

1.1 MMYOLO Introduction

MMYOLO is an open-source algorithms toolkit of YOLO based on PyTorch and MMDetection, part of the OpenMM-Lab project. MMYOLO is positioned as a popular open-source library of YOLO series and core library of industrial applications. Its vision diagram is shown as follows:

The following tasks are currently supported:

- Object detection
- Rotated object detection

The YOLO series of algorithms currently supported are as follows:

- YOLOv5
- YOLOX
- RTMDet
- RTMDet-Rotated
- YOLOv6
- YOLOv7
- PPYOLOE
- YOLOv8

The datasets currently supported are as follows:

- COCO Dataset
- VOC Dataset
- CrowdHuman Dataset
- DOTA 1.0 Dataset

MMYOLO runs on Linux, Windows, macOS, and supports PyTorch 1.7 or later. It has the following three characteristics:

- **Unified and convenient algorithm evaluation**

  MMYOLO unifies various YOLO algorithm modules and provides a unified evaluation process, so that users can compare and analyze fairly and conveniently.

- **Extensive documentation for started and advanced**

  MMYOLO provides a series of documents, including getting started, deployment, advanced practice and algorithm analysis, which is convenient for different users to get started and expand.

- **Modular Design**

  MMYOLO disentangled the framework into modular components, and users can easily build custom models by combining different modules and training and testing strategies.

1.2 User guide for this documentation

MMYOLO divides the document structure into 6 parts, corresponding to different user needs.

- **Get started with MMYOLO**. This part is must read for first-time MMYOLO users, so please read it carefully.
- **Recommend Topics**. This part is the essence documentation provided in MMYOLO by topics, including lots of MMYOLO features, etc. Highly recommended reading for all MMYOLO users.
- **Common functions**. This part provides a list of common features that you will use during the training and testing process, so you can refer back to them when you need.
- **Useful tools**. This part is useful tools summary under `tools`, so that you can quickly and happily use the various scripts provided in MMYOLO.
- **Basic and advanced tutorials**. This part introduces some basic concepts and advanced tutorials in MMYOLO. It is suitable for users who want to understand the design idea and structure design of MMYOLO in detail.
- **Others**. The rest includes model repositories, specifications and interface documentation, etc.

Users with different needs can choose your favorite content to read. If you have any questions about this documentation or a better idea to improve it, welcome to post a Pull Request to MMYOLO ~. Please refer to *How to Contribute to MMYOLO*

CHAPTER

TWO

PREREQUISITES

Compatible MMEngine, MMCV and MMDetection versions are shown as below. Please install the correct version to avoid installation issues.

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

MMDetection works on Linux, Windows, and macOS. It requires:

- Python 3.7+
- PyTorch 1.7+
- CUDA 9.2+
- GCC 5.4+

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

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

Step 1. Create a conda environment and activate it.

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

Step 2. Install PyTorch following official commands, e.g.

On GPU platforms:

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

On CPU platforms:

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

Step 3. Verify PyTorch installation

```
python -c "import torch; print(torch.__version__); print(torch.cuda.is_available())"
```

If the GPU is used, the version information and `True` are printed; otherwise, the version information and `False` are printed.

CHAPTER

THREE

INSTALLATION

3.1 Best Practices

Step 0. Install MMEngine and MMCV using MIM.

```
pip install -U openmim
mim install "mmengine>=0.6.0"
mim install "mmcv>=2.0.0rc4,<2.1.0"
mim install "mmdet>=3.0.0,<4.0.0"
```

If you are currently in the mmyolo project directory, you can use the following simplified commands

```
cd mmyolo
pip install -U openmim
mim install -r requirements/mminstall.txt
```

Note:

a. In MMCV-v2.x, `mmcv-full` is rename to `mmcv`, if you want to install `mmcv` without CUDA ops, you can use `mim install "mmcv-lite>=2.0.0rc1"` to install the lite version.

b. If you would like to use `albumentations`, we suggest using `pip install -r requirements/albu.txt` or `pip install -U albumentations --no-binary qudida,albumentations`. If you simply use `pip install albumentations==1.0.1`, it will install `opencv-python-headless` simultaneously (even though you have already installed `opencv-python`). We recommended checking the environment after installing albumentation to ensure that `opencv-python` and `opencv-python-headless` are not installed at the same time, because it might cause unexpected issues if they both installed. Please refer to official documentation for more details.

Step 1. Install MMYOLO.

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

```
git clone https://github.com/open-mmlab/mmyolo.git
cd mmyolo
# Install albumentations
pip install -r requirements/albu.txt
# Install MMYOLO
mim install -v -e .
# "-v" means verbose, or more output
# "-e" means installing a project in editable mode,
# thus any local modifications made to the code will take effect without reinstallation.
```

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

```
mim install "mmyolo"
```

3.2 Verify the installation

To verify whether MMYOLO is installed correctly, we provide an inference demo.

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

```
mim download mmyolo --config yolov5_s-v61_syncbn_fast_8xb16-300e_coco --dest .
```

The downloading will take several seconds or more, depending on your network environment. When it is done, you will find two files `yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py` and `yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth` in your current folder.

Step 2. Verify the inference demo.

Option (a). If you install MMYOLO from source, just run the following command.

```
python demo/image_demo.py demo/demo.jpg \
                          yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
                          yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth

# Optional parameters
# --out-dir ./output *The detection results are output to the specified directory. When␣
↪args have action --show, the script do not save results. Default: ./output
# --device cuda:0    *The computing resources used, including cuda and cpu. Default:␣
↪cuda:0
# --show             *Display the results on the screen. Default: False
# --score-thr 0.3    *Confidence threshold. Default: 0.3
```

You will see a new image on your `output` folder, where bounding boxes are plotted.

Supported input types:

- Single image, include `jpg`, `jpeg`, `png`, `ppm`, `bmp`, `pgm`, `tif`, `tiff`, `webp`.
- Folder, all image files in the folder will be traversed and the corresponding results will be output.
- URL, will automatically download from the URL and the corresponding results will be output.

Option (b). If you install MMYOLO with MIM, open your python interpreter and copy&paste the following codes.

```
from mmdet.apis import init_detector, inference_detector

config_file = 'yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py'
checkpoint_file = 'yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth'
model = init_detector(config_file, checkpoint_file, device='cpu')  # or device='cuda:0'
inference_detector(model, 'demo/demo.jpg')
```

You will see a list of `DetDataSample`, and the predictions are in the `pred_instance`, indicating the detected bounding boxes, labels, and scores.

3.3 Using MMYOLO with Docker

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

Reminder: If you find out that your download speed is very slow, we suggest canceling the comments in the last two lines of `Optional` in the Dockerfile to obtain a rocket like download speed:

```
# (Optional)
RUN sed -i 's/http:\/\/archive.ubuntu.com\/ubuntu\//http:\/\/mirrors.aliyun.com\/ubuntu\/
↪/g' /etc/apt/sources.list && \
    pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

Build Command

```
# build an image with PyTorch 1.9, CUDA 11.1
# If you prefer other versions, just modified the Dockerfile
docker build -t mmyolo docker/
```

Run it with:

```
export DATA_DIR=/path/to/your/dataset
docker run --gpus all --shm-size=8g -it -v ${DATA_DIR}:/mmyolo/data mmyolo
```

For other customized inatallation, see *Customized Installation*

3.4 Troubleshooting

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

CHAPTER

FOUR

15 MINUTES TO GET STARTED WITH MMYOLO OBJECT DETECTION

Object detection task refers to that given a picture, the network predicts all the categories of objects included in the picture and the corresponding boundary boxes

Take the small dataset of cat as an example, you can easily learn MMYOLO object detection in 15 minutes. The whole process consists of the following steps:

- *Installation*
- *Dataset*
- *Config*
- *Training*
- *Testing*
- *EasyDeploy*

In this tutorial, we take YOLOv5-s as an example. For the rest of the YOLO series algorithms, please see the corresponding algorithm configuration folder.

4.1 Installation

Assuming you've already installed Conda in advance, then install PyTorch using the following commands.

Note: Note: Since this repo uses OpenMMLab 2.0, it is better to create a new conda virtual environment to prevent conflicts with the repo installed in OpenMMLab 1.0.

```
conda create -n mmyolo python=3.8 -y
conda activate mmyolo
# If you have GPU
conda install pytorch torchvision -c pytorch
# If you only have CPU
# conda install pytorch torchvision cpuonly -c pytorch
```

Install MMYOLO and dependency libraries using the following commands.

```
git clone https://github.com/open-mmlab/mmyolo.git
cd mmyolo
pip install -U openmim
mim install -r requirements/mminstall.txt
```

(continues on next page)

(continued from previous page)

```
# Install albumentations
mim install -r requirements/albu.txt
# Install MMYOLO
mim install -v -e .
# "-v" means verbose, or more output
# "-e" means installing a project in editable mode,
# thus any local modifications made to the code will take effect without reinstallation.
```

For details about how to configure the environment, see *Installation and verification*.

4.2 Dataset

The Cat dataset is a single-category dataset consisting of 144 pictures (the original pictures are provided by @RangeKing, and cleaned by @PeterH0323), which contains the annotation information required for training. The sample image is shown below:

You can download and use it directly by the following command:

```
python tools/misc/download_dataset.py --dataset-name cat --save-dir ./data/cat --unzip --
↪delete
```

This dataset is automatically downloaded to the `./data/cat` dir with the following directory structure:

The cat dataset is located in the mmyolo project dir, and `data/cat/annotations` stores annotations in COCO format, and `data/cat/images` stores all images

4.3 Config

Taking YOLOv5 algorithm as an example, considering the limited GPU memory of users, we need to modify some default training parameters to make them run smoothly. The key parameters to be modified are as follows:

- YOLOv5 is an Anchor-Based algorithm, and different datasets need to calculate suitable anchors adaptively
- The default config uses 8 GPUs with a batch size of 16 per GPU. Now change it to a single GPU with a batch size of 12.
- The default training epoch is 300. Change it to 40 epoch
- Given the small size of the dataset, we opted to use fixed backbone weights
- In principle, the learning rate should be linearly scaled accordingly when the batch size is changed, but actual measurements have found that this is not necessary

Create a `yolov5_s-v61_fast_1xb12-40e_cat.py` config file in the `configs/yolov5` folder (we have provided this config for you to use directly) and copy the following into the config file.

```
# Inherit and overwrite part of the config based on this config
_base_ = 'yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py'

data_root = './data/cat/' # dataset root
class_name = ('cat', ) # dataset category name
num_classes = len(class_name) # dataset category number
# metainfo is a configuration that must be passed to the dataloader, otherwise it is␣
↪invalid
```

(continues on next page)

(continued from previous page)

```
# palette is a display color for category at visualization
# The palette length must be greater than or equal to the length of the classes
metainfo = dict(classes=class_name, palette=[(20, 220, 60)])

# Adaptive anchor based on tools/analysis_tools/optimize_anchors.py
anchors = [
    [(68, 69), (154, 91), (143, 162)],  # P3/8
    [(242, 160), (189, 287), (391, 207)],  # P4/16
    [(353, 337), (539, 341), (443, 432)]  # P5/32
]
# Max training 40 epoch
max_epochs = 40
# Set batch size to 12
train_batch_size_per_gpu = 12
# dataloader num workers
train_num_workers = 4

# load COCO pre-trained weight
load_from = 'https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_
↪8xb16-300e_coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth'␣
↪ # noqa

model = dict(
    # Fixed the weight of the entire backbone without training
    backbone=dict(frozen_stages=4),
    bbox_head=dict(
        head_module=dict(num_classes=num_classes),
        prior_generator=dict(base_sizes=anchors)
    ))

train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        data_root=data_root,
        metainfo=metainfo,
        # Dataset annotation file of json path
        ann_file='annotations/trainval.json',
        # Dataset prefix
        data_prefix=dict(img='images/')))

val_dataloader = dict(
    dataset=dict(
        metainfo=metainfo,
        data_root=data_root,
        ann_file='annotations/test.json',
        data_prefix=dict(img='images/')))

test_dataloader = val_dataloader

_base_.optim_wrapper.optimizer.batch_size_per_gpu = train_batch_size_per_gpu
```

(continues on next page)

(continued from previous page)

```
val_evaluator = dict(ann_file=data_root + 'annotations/test.json')
test_evaluator = val_evaluator

default_hooks = dict(
    # Save weights every 10 epochs and a maximum of two weights can be saved.
    # The best model is saved automatically during model evaluation
    checkpoint=dict(interval=10, max_keep_ckpts=2, save_best='auto'),
    # The warmup_mim_iter parameter is critical.
    # The default value is 1000 which is not suitable for cat datasets.
    param_scheduler=dict(max_epochs=max_epochs, warmup_mim_iter=10),
    # The log printing interval is 5
    logger=dict(type='LoggerHook', interval=5))
# The evaluation interval is 10
train_cfg = dict(max_epochs=max_epochs, val_interval=10)
```

The above config is inherited from `yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py`. According to the characteristics of cat dataset updated `data_root`, `metainfo`, `train_dataloader`, `val_dataloader`, `num_classes` and other config.

4.4 Training

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py
```

Run the above training command, `work_dirs/yolov5_s-v61_fast_1xb12-40e_cat` folder will be automatically generated, the checkpoint file and the training config file will be saved in this folder. On a low-end 1660 GPU, the entire training process takes about eight minutes.

The performance on `test.json` is as follows:

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.631
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.909
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.747
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.631
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.627
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.703
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.703
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.703
```

The above properties are printed via the COCO API, where -1 indicates that no object exists for the scale. According to the rules defined by COCO, the Cat dataset contains all large sized objects, and there are no small or medium-sized objects.

4.4.1 Some Notes

Two key warnings are printed during training:

- You are using `YOLOv5Head` with num_classes == 1. The loss_cls will be 0. This is a normal phenomenon.
- The model and loaded state dict do not match exactly

Neither of these warnings will have any impact on performance. The first warning is because the `num_classes` currently trained is 1, the loss of the classification branch is always 0 according to the community of the YOLOv5 algorithm, which is a normal phenomenon. The second warning is because we are currently training in fine-tuning mode, we load the COCO pre-trained weights for 80 classes, This will lead to the final Head module convolution channel number does not correspond, resulting in this part of the weight can not be loaded, which is also a normal phenomenon.

4.4.2 Training is resumed after the interruption

If you stop training, you can add `--resume` to the end of the training command and the program will automatically resume training with the latest weights file from `work_dirs`.

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py --resume
```

4.4.3 Save GPU memory strategy

The above config requires about 3G RAM, so if you don't have enough, consider turning on mixed-precision training

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py --amp
```

4.4.4 Training visualization

MMYOLO currently supports local, TensorBoard, WandB and other back-end visualization. The default is to use local visualization, and you can switch to WandB and other real-time visualization of various indicators in the training process.

1 WandB

WandB visualization need registered in website, and in the https://wandb.ai/settings for wandb API Keys.

```
pip install wandb
# After running wandb login, enter the API Keys obtained above, and the login is␣
↪successful.
wandb login
```

Add the wandb config at the end of config file we just created: `configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py`.

```
visualizer = dict(vis_backends = [dict(type='LocalVisBackend'), dict(type=
↪'WandbVisBackend')])
```

Running the training command and you will see the loss, learning rate, and coco/bbox_mAP visualizations in the link.

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py
```

2 Tensorboard

Install Tensorboard package:

```
pip install tensorboard
```

Add the `tensorboard` config at the end of config file we just created: `configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py`.

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

After re-running the training command, Tensorboard file will be generated in the visualization folder `work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/{timestamp}/vis_data`. We can use Tensorboard to view the loss, learning rate, and coco/bbox_mAP visualizations from a web link by running the following command:

```
tensorboard --logdir=work_dirs/yolov5_s-v61_fast_1xb12-40e_cat
```

4.5 Testing

```
python tools/test.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                     work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
                     --show-dir show_results
```

Run the above test command, you can not only get the AP performance printed in the **Training** section, You can also automatically save the result images to the `work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/{timestamp}/show_results` folder. Below is one of the result images, the left image is the actual annotation, and the right image is the inference result of the model.

You can also visualize model inference results in a browser window if you use 'WandbVisBackend' or 'TensorboardVisBackend'.

4.6 Feature map visualization

MMYOLO provides visualization scripts for feature map to analyze the current model training. Please refer to *Feature Map Visualization*

Due to the bias of direct visualization of `test_pipeline`, we need to modify the `test_pipeline` of `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py`

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
```

(continues on next page)

(continued from previous page)

```
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

to the following config:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # modify the␣
↪LetterResize to mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

Let's choose the `data/cat/images/IMG_20221020_112705.jpg` image as an example to visualize the output feature maps of YOLOv5 backbone and neck layers.

1. Visualize the three channels of YOLOv5 backbone

```
python demo/featmap_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                                configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                                work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
                                --target-layers backbone \
                                --channel-reduction squeeze_mean
```

The result will be saved to the output folder in current path. Three output feature maps plotted in the above figure correspond to small, medium and large output feature maps. As the backbone of this training is not actually involved in training, it can be seen from the above figure that the big object cat is predicted on the small feature map, which is in line with the idea of hierarchical detection of object detection.

2. Visualize the three channels of YOLOv5 neck

```
python demo/featmap_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                                configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                                work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
                                --target-layers neck \
                                --channel-reduction squeeze_mean
```

As can be seen from the above figure, because neck is involved in training, and we also reset anchor, the three output feature maps are forced to simulate the same scale object, resulting in the three output maps of neck are similar, which destroys the original pre-training distribution of backbone. At the same time, it can also be seen that 40 epochs are not enough to train the above dataset, and the feature maps do not perform well.

3. Grad-Based CAM visualization

Based on the above feature map visualization, we can analyze Grad CAM at the feature layer of bbox level.

Install `grad-cam` package:

```
pip install "grad-cam"
```

(a) View Grad CAM of the minimum output feature map of the neck

```
python demo/boxam_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                              configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                              work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
                              --target-layer neck.out_layers[2]
```

(b) View Grad CAM of the medium output feature map of the neck

```
python demo/boxam_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                              configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                              work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
                              --target-layer neck.out_layers[1]
```

(c) View Grad CAM of the maximum output feature map of the neck

```
python demo/boxam_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                              configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                              work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
                              --target-layer neck.out_layers[0]
```

4.7 EasyDeploy deployment

Here we'll use MMYOLO's EasyDeploy to demonstrate the transformation deployment and basic inference of model.

First you need to follow EasyDeploy's basic documentation controls own equipment installed for each library.

```
pip install onnx
pip install onnx-simplifier # Install if you want to use simplify
pip install tensorrt        # If you have GPU environment and need to output TensorRT␣
↪model you need to continue execution
```

Once installed, you can use the following command to transform and deploy the trained model on the cat dataset with one click. The current ONNX version is 1.13.0 and TensorRT version is 8.5.3.1, so keep the `--opset` value of 11. The remaining parameters need to be adjusted according to the config used. Here we export the CPU version of ONNX with the `--backend` set to 1.

```
python projects/easydeploy/tools/export.py \
        configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
        work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
        --work-dir work_dirs/yolov5_s-v61_fast_1xb12-40e_cat \
    --img-size 640 640 \
    --batch 1 \
    --device cpu \
    --simplify \
        --opset 11 \
        --backend 1 \
        --pre-topk 1000 \
        --keep-topk 100 \
```

(continues on next page)

(continued from previous page)

```
        --iou-threshold 0.65 \
        --score-threshold 0.25
```

On success, you will get the converted ONNX model under `work-dir`, which is named `end2end.onnx` by default.

Let's use `end2end.onnx` model to perform a basic image inference:

```
python projects/easydeploy/tools/image-demo.py \
    data/cat/images/IMG_20210728_205312.jpg \
    configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/end2end.onnx \
    --device cpu
```

After successful inference, the result image will be generated in the `output` folder of the default MMYOLO root directory. If you want to see the result without saving it, you can add `--show` to the end of the above command. For convenience, the following is the generated result.

Let's go on to convert the engine file for TensorRT, because TensorRT needs to be specific to the current environment and deployment version, so make sure to export the parameters, here we export the TensorRT8 file, the `--backend` is 2.

```
python projects/easydeploy/tools/export.py \
    configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
    --work-dir work_dirs/yolov5_s-v61_fast_1xb12-40e_cat \
    --img-size 640 640 \
    --batch 1 \
    --device cuda:0 \
    --simplify \
    --opset 11 \
    --backend 2 \
    --pre-topk 1000 \
    --keep-topk 100 \
    --iou-threshold 0.65 \
    --score-threshold 0.25
```

The resulting `end2end.onnx` is the ONNX file for the TensorRT8 deployment, which we will use to complete the TensorRT engine transformation.

```
python projects/easydeploy/tools/build_engine.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/end2end.onnx \
    --img-size 640 640 \
    --device cuda:0
```

Successful execution will generate the `end2end.engine` file under `work-dir`:

```
work_dirs/yolov5_s-v61_fast_1xb12-40e_cat
├── 202302XX_XXXXXX
│   ├── 202302XX_XXXXXX.log
│   └── vis_data
│       ├── 202302XX_XXXXXX.json
│       ├── config.py
│       └── scalars.json
├── best_coco
```

(continues on next page)

(continued from previous page)

```
    └── bbox_mAP_epoch_40.pth
├── end2end.engine
├── end2end.onnx
├── epoch_30.pth
├── epoch_40.pth
├── last_checkpoint
└── yolov5_s-v61_fast_1xb12-40e_cat.py
```

Let's continue use `image-demo.py` for image inference:

```
python projects/easydeploy/tools/image-demo.py \
    data/cat/images/IMG_20210728_205312.jpg \
    configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/end2end.engine \
    --device cuda:0
```

Here we choose to save the inference results under `output` instead of displaying them directly. The following shows the inference results.

This completes the transformation deployment of the trained model and checks the inference results. This is the end of the tutorial.

The full content above can be viewed in 15_minutes_object_detection.ipynb. If you encounter problems during training or testing, please check the *common troubleshooting steps* first and feel free to open an issue if you still can't solve it.

CHAPTER

FIVE

15 MINUTES TO GET STARTED WITH MMYOLO ROTATED OBJECT DETECTION

TODO

CHAPTER

SIX

15 MINUTES TO GET STARTED WITH MMYOLO INSTANCE SEGMENTATION

Instance segmentation is a task in computer vision that aims to segment each object in an image and assign each object a unique identifier.

Unlike semantic segmentation, instance segmentation not only segments out different categories in an image, but also separates different instances of the same category.

Taking the downloadable balloon dataset as an example, I will guide you through a 15-minute easy introduction to MMYOLO instance segmentation. The entire process includes the following steps:

- *Installation*
- *Dataset*
- *Config*
- *Training*
- *Testing*
- *EasyDeploy*

In this tutorial, we will use YOLOv5-s as an example. For the demo configuration of the balloon dataset with other YOLO series algorithms, please refer to the corresponding algorithm configuration folder.

6.1 Installation

Assuming you've already installed Conda in advance, then install PyTorch using the following commands.

Note: Note: Since this repo uses OpenMMLab 2.0, it is better to create a new conda virtual environment to prevent conflicts with the repo installed in OpenMMLab 1.0.

```
conda create -n mmyolo python=3.8 -y
conda activate mmyolo
# If you have GPU
conda install pytorch torchvision -c pytorch
# If you only have CPU
# conda install pytorch torchvision cpuonly -c pytorch
```

Install MMYOLO and dependency libraries using the following commands.

```
git clone https://github.com/open-mmlab/mmyolo.git
cd mmyolo
pip install -U openmim
mim install -r requirements/mminstall.txt
# Install albumentations
mim install -r requirements/albu.txt
# Install MMYOLO
mim install -v -e .
# "-v" means verbose, or more output
# "-e" means installing a project in editable mode,
# thus any local modifications made to the code will take effect without reinstallation.
```

For details about how to configure the environment, see *Installation and verification*.

6.2 Dataset

The Balloon dataset is a single-class dataset that consists of 74 images and includes annotated information required for training. Here is an example image from the dataset:

You can download and use it directly by the following command:

```
python tools/misc/download_dataset.py --dataset-name balloon --save-dir ./data/balloon --
↪unzip --delete
python ./tools/dataset_converters/balloon2coco.py
```

The data for the MMYOLO project is located in the MMYOLO project directory. The `train.json` and `val.json` files store the annotations in COCO format, while the `data/balloon/train` and `data/balloon/val` directories contain all the images for the dataset.

6.3 Config

Taking YOLOv5 algorithm as an example, considering the limited GPU memory of users, we need to modify some default training parameters to make them run smoothly. The key parameters to be modified are as follows:

- YOLOv5 is an Anchor-Based algorithm, and different datasets need to calculate suitable anchors adaptively.
- The default config uses 8 GPUs with a batch size of 16 per GPU. Now change it to a single GPU with a batch size of 12.
- In principle, the learning rate should be linearly scaled accordingly when the batch size is changed, but actual measurements have found that this is not necessary.

To perform the specific operation, create a new configuration file named `yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.py` in the `configs/yolov5/ins_seg` folder. For convenience, we have already provided this configuration file. Copy the following contents into the configuration file.

```
_base_ = './yolov5_ins_s-v61_syncbn_fast_8xb16-300e_coco_instance.py'  # noqa

data_root = 'data/balloon/' # dataset root
# Training set annotation file of json path
train_ann_file = 'train.json'
```

(continues on next page)

(continued from previous page)

```
train_data_prefix = 'train/'  # Dataset prefix
# Validation set annotation file of json path
val_ann_file = 'val.json'
val_data_prefix = 'val/'
metainfo = {
    'classes': ('balloon', ), # dataset category name
    'palette': [
        (220, 20, 60),
    ]
}
num_classes = 1
# Set batch size to 4
train_batch_size_per_gpu = 4
# dataloader num workers
train_num_workers = 2
log_interval = 1
######################
train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        data_root=data_root,
        metainfo=metainfo,
        data_prefix=dict(img=train_data_prefix),
        ann_file=train_ann_file))
val_dataloader = dict(
    dataset=dict(
        data_root=data_root,
        metainfo=metainfo,
        data_prefix=dict(img=val_data_prefix),
        ann_file=val_ann_file))
test_dataloader = val_dataloader
val_evaluator = dict(ann_file=data_root + val_ann_file)
test_evaluator = val_evaluator
default_hooks = dict(logger=dict(interval=log_interval))
######################

model = dict(bbox_head=dict(head_module=dict(num_classes=num_classes)))
```

The above configuration inherits from `yolov5_ins_s-v61_syncbn_fast_8xb16-300e_coco_instance.py` and updates configurations such as `data_root`, `metainfo`, `train_dataloader`, `val_dataloader`, `num_classes`, etc., based on the characteristics of the balloon dataset.

6.4 Training

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py
```

After running the training command mentioned above, the folder `work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance` will be automatically generated. The weight files and the training configuration file for this session will be saved in this folder. On a lower-end GPU like the GTX 1660, the entire training process will take approximately 30 minutes.

The performance on `val.json` is as follows:

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.330
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.509
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.317
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.103
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.417
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.150
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.396
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.454
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.317
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.525
```

The above performance is obtained by printing using the COCO API, where -1 indicates the absence of objects of that scale.

6.4.1 Some Notes

The key warnings are printed during training:

- You are using `YOLOv5Head` with num_classes == 1. The loss_cls will be 0. This is a normal phenomenon.

The warning is because the `num_classes` currently trained is 1, the loss of the classification branch is always 0 according to the community of the YOLOv5 algorithm, which is a normal phenomenon.

6.4.2 Training is resumed after the interruption

If you stop training, you can add `--resume` to the end of the training command and the program will automatically resume training with the latest weights file from `work_dirs`.

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py --resume
```

6.4.3 Save GPU memory strategy

The above config requires about 3G RAM, so if you don't have enough, consider turning on mixed-precision training

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py --amp
```

6.4.4 Training visualization

MMYOLO currently supports local, TensorBoard, WandB and other back-end visualization. The default is to use local visualization, and you can switch to WandB and other real-time visualization of various indicators in the training process.

1 WandB

WandB visualization need registered in website, and in the https://wandb.ai/settings for wandb API Keys.

```
pip install wandb
# After running wandb login, enter the API Keys obtained above, and the login is␣
↪successful.
wandb login
```

Add the wandb config at the end of config file we just created: `configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.py`.

```
visualizer = dict(vis_backends = [dict(type='LocalVisBackend'), dict(type=
↪'WandbVisBackend')])
```

Running the training command and you will see the loss, learning rate, and coco/bbox_mAP visualizations in the link.

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py
```

2 Tensorboard

Install Tensorboard package using the following command:

```
pip install tensorboard
```

Add the `tensorboard` config at the end of config file we just created: `configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.py`.

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

After re-running the training command, Tensorboard file will be generated in the visualization folder `work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/{timestamp}/vis_data`. We can use Tensorboard to view the loss, learning rate, and coco/bbox_mAP visualizations from a web link by running the following command:

```
tensorboard --logdir=work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance
```

6.5 Testing

```
python tools/test.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py \
                      work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/
↪best_coco_bbox_mAP_epoch_300.pth \
                      --show-dir show_results
```

Run the above test command, you can not only get the AP performance printed in the **Training** section, You can also automatically save the result images to the `work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/{timestamp}/show_results` folder. Below is one of the result images, the left image is the actual annotation, and the right image is the inference result of the model.

You can also visualize model inference results in a browser window if you use `WandbVisBackend` or `TensorboardVisBackend`.

6.6 Feature map visualization

MMYOLO provides visualization scripts for feature map to analyze the current model training. Please refer to *Feature Map Visualization*

Due to the bias of direct visualization of `test_pipeline`, we need to modify the `test_pipeline` of `configs/yolov5/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.py`

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

to the following config:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # modify the␣
↪LetterResize to mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
```

(continues on next page)

(continued from previous page)

```
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

Let's choose the `data/balloon/train/3927754171_9011487133_b.jpg` image as an example to visualize the output feature maps of YOLOv5 backbone and neck layers.

1. Visualize the three channels of YOLOv5s backbone

```
python demo/featmap_vis_demo.py data/balloon/train/3927754171_9011487133_b.jpg \
    configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.py \
    work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/best_coco_bbox_
↪mAP_epoch_300.pth \ --target-layers backbone \
    --channel-reduction squeeze_mean
```

The result will be saved to the output folder in current path. Three output feature maps plotted in the above figure correspond to small, medium and large output feature maps.

2. Visualize the three channels of YOLOv5 neck

```
python demo/featmap_vis_demo.py data/balloon/train/3927754171_9011487133_b.jpg \
    configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.py \
    work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/best_coco_bbox_
↪mAP_epoch_300.pth \ --target-layers neck \
    --channel-reduction squeeze_mean
```

TODO

6.7 EasyDeploy deployment

TODO

The full content above can be viewed in 15_minutes_object_detection.ipynb. This is the end of the tutorial. If you encounter problems during training or testing, please check the *common troubleshooting steps* first and feel free to open an issue if you still can't solve it.

CHAPTER

SEVEN

CONTRIBUTING TO OPENMMLAB

Welcome to the MMYOLO community, we are committed to building a cutting-edge computer vision foundational library, and all kinds of contributions are welcomed, including but not limited to

Fix bug

You can directly post a Pull Request to fix typos in code or documents

The steps to fix the bug of code implementation are as follows.

1. If the modification involves significant changes, you should create an issue first and describe the error information and how to trigger the bug. Other developers will discuss it with you and propose a proper solution.
2. Posting a pull request after fixing the bug and adding the corresponding unit test.

New Feature or Enhancement

1. If the modification involves significant changes, you should create an issue to discuss with our developers to propose a proper design.
2. Post a Pull Request after implementing the new feature or enhancement and add the corresponding unit test.

Document

You can directly post a pull request to fix documents. If you want to add a document, you should first create an issue to check if it is reasonable.

7.1 Preparation

The commands for processing pull requests are implemented using Git, and this chapter details `Git Configuration` and `associated GitHub`.

7.1.1 1. Git Configuration

First, make sure you have Git installed on your computer. For Linux systems and macOS systems, Git is generally installed by default. If it is not installed, it can be downloaded at Git-Downloads.

```
# view the Git version
git --version
```

Second, check your `Git Config`

```
# view the Git config
git config --global --list
```

If `user.name` and `user.email` are empty, run the command.

```
git config --global user.name "Change your username here"
git config --global user.email "Change your useremail here"
```

Finally, run the command in `git bash` or `terminal` to generate the key file. After the generation is successful, a `.ssh` file will appear in the user directory, and `id_rsa.pub` is the public key file.

```
# useremail is GitHub's email address
ssh-keygen -t rsa -C "useremail"
```

7.1.2 2. Associated GitHub

First, open `id_rsa.pub` and copy the entire contents.

Second, log in to your GitHub account to set it up.

Click `New SSH key` to add a new SSH keys, and paste the copied content into Key.

Finally, verify that SSH matches the GitHub account by running the command in `git bash` or `terminal`. If it matches, enter `yes` to succeed.

```
ssh -T git@github.com
```

7.2 Pull Request Workflow

If you're not familiar with Pull Request, don't worry! The following guidance will tell you how to create a Pull Request step by step. If you want to dive into the development mode of Pull Request, you can refer to the official documents

7.2.1 1. Fork and clone

If you are posting a pull request for the first time, you should fork the OpenMMLab repositories by clicking the **Fork** button in the top right corner of the GitHub page, and the forked repositories will appear under your GitHub profile.

Then, you can clone the repositories to local:

```
git clone git@github.com:{username}/mmyolo.git
```

After that, you should get into the project folder and add official repository as the upstream repository.

```
cd mmyolo
git remote add upstream git@github.com:open-mmlab/mmyolo
```

Check whether the remote repository has been added successfully by `git remote -v`

```
origin        git@github.com:{username}/mmyolo.git (fetch)
origin        git@github.com:{username}/mmyolo.git (push)
upstream        git@github.com:open-mmlab/mmyolo (fetch)
upstream        git@github.com:open-mmlab/mmyolo (push)
```

Note: Here's a brief introduction to the origin and upstream. When we use "git clone", we create an "origin" remote by default, which points to the repository cloned from. As for "upstream", we add it ourselves to point to the target

repository. Of course, if you don't like the name "upstream", you could name it as you wish. Usually, we'll push the code to "origin". If the pushed code conflicts with the latest code in official("upstream"), we should pull the latest code from upstream to resolve the conflicts, and then push to "origin" again. The posted Pull Request will be updated automatically.

7.2.2 2. Configure pre-commit

You should configure pre-commit in the local development environment to make sure the code style matches that of OpenMMLab. **Note**: The following code should be executed under the MMYOLO directory.

```
pip install -U pre-commit
pre-commit install
```

Check that pre-commit is configured successfully, and install the hooks defined in `.pre-commit-config.yaml`.

```
pre-commit run --all-files
```

Note: Chinese users may fail to download the pre-commit hooks due to the network issue. In this case, you could download these hooks from gitee by setting the .pre-commit-config-zh-cn.yaml

pre-commit install -c .pre-commit-config-zh-cn.yaml pre-commit run –all-files -c .pre-commit-config-zh-cn.yaml

If the installation process is interrupted, you can repeatedly run `pre-commit run ...` to continue the installation.

If the code does not conform to the code style specification, pre-commit will raise a warning and fixes some of the errors automatically.

If we want to commit our code bypassing the pre-commit hook, we can use the `--no-verify` option(**only for temporarily commit**).

```
git commit -m "xxx" --no-verify
```

7.2.3 3. Create a development branch

After configuring the pre-commit, we should create a branch based on the dev branch to develop the new feature or fix the bug. The proposed branch name is `username/pr_name`

```
git checkout -b yhc/refactor_contributing_doc
```

In subsequent development, if the dev branch of the local repository is behind the dev branch of "upstream", we need to pull the upstream for synchronization, and then execute the above command:

```
git pull upstream dev
```

7.2.4 4. Commit the code and pass the unit test

- MMYOLO introduces mypy to do static type checking to increase the robustness of the code. Therefore, we need to add Type Hints to our code and pass the mypy check. If you are not familiar with Type Hints, you can refer to this tutorial.
- The committed code should pass through the unit test

```
# Pass all unit tests
pytest tests

# Pass the unit test of yolov5_coco dataset
pytest tests/test_datasets/test_yolov5_coco.py
```

  If the unit test fails for lack of dependencies, you can install the dependencies referring to the *guidance*

- If the documents are modified/added, we should check the rendering result referring to *guidance*

7.2.5 5. Push the code to remote

We could push the local commits to remote after passing through the check of unit test and pre-commit. You can associate the local branch with remote branch by adding `-u` option.

```
git push -u origin {branch_name}
```

This will allow you to use the `git push` command to push code directly next time, without having to specify a branch or the remote repository.

7.2.6 6. Create a Pull Request

(1) Create a pull request in GitHub's Pull request interface

(2) Modify the PR description according to the guidelines so that other developers can better understand your changes.

Note: The *base* branch should be modified to *dev* branch.

Find more details about Pull Request description in *pull request guidelines*.

note

(a) The Pull Request description should contain the reason for the change, the content of the change, and the impact of the change, and be associated with the relevant Issue (see documentation)

(b) If it is your first contribution, please sign the CLA

(c) Check whether the Pull Request pass through the CI

MMYOLO will run unit test for the posted Pull Request on Linux, based on different versions of Python, and PyTorch to make sure the code is correct. We can see the specific test information by clicking `Details` in the above image so that we can modify the code.

(3) If the Pull Request passes the CI, then you can wait for the review from other developers. You'll modify the code based on the reviewer's comments, and repeat the steps *4*-*5* until all reviewers approve it. Then, we will merge it ASAP.

7.2.7 7. Resolve conflicts

If your local branch conflicts with the latest dev branch of "upstream", you'll need to resolove them. There are two ways to do this:

```
git fetch --all --prune
git rebase upstream/dev
```

or

```
git fetch --all --prune
git merge upstream/dev
```

If you are very good at handling conflicts, then you can use rebase to resolve conflicts, as this will keep your commit logs tidy. If you are unfamiliar with `rebase`, you can use `merge` to resolve conflicts.

7.3 Guidance

7.3.1 Unit test

We should also make sure the committed code will not decrease the coverage of unit test, we could run the following command to check the coverage of unit test:

```
python -m coverage run -m pytest /path/to/test_file
python -m coverage html
# check file in htmlcov/index.html
```

7.3.2 Document rendering

If the documents are modified/added, we should check the rendering result. We could install the dependencies and run the following command to render the documents and check the results:

```
pip install -r requirements/docs.txt
cd docs/zh_cn/
# or docs/en
make html
# check file in ./docs/zh_cn/_build/html/index.html
```

7.4 Code style

7.4.1 Python

We adopt PEP8 as the preferred code style.

We use the following tools for linting and formatting:

- flake8: A wrapper around some linter tools.
- isort: A Python utility to sort imports.
- yapf: A formatter for Python files.

- codespell: A Python utility to fix common misspellings in text files.
- mdformat: Mdformat is an opinionated Markdown formatter that can be used to enforce a consistent style in Markdown files.
- docformatter: A formatter to format docstring.

Style configurations of yapf and isort can be found in setup.cfg.

We use pre-commit hook that checks and formats for `flake8`, `yapf`, `isort`, `trailing whitespaces`, `markdown files`, fixes `end-of-files`, `double-quoted-strings`, `python-encoding-pragma`, `mixed-line-ending`, sorts `requirments.txt` automatically on every commit. The config for a pre-commit hook is stored in .pre-commit-config.

7.4.2 C++ and CUDA

We follow the Google C++ Style Guide.

7.5 PR Specs

1. Use pre-commit hook to avoid issues of code style
2. One short-time branch should be matched with only one PR
3. Accomplish a detailed change in one PR. Avoid large PR
    - Bad: Support Faster R-CNN
    - Acceptable: Add a box head to Faster R-CNN
    - Good: Add a parameter to box head to support custom conv-layer number
4. Provide clear and significant commit message
5. Provide clear and meaningful PR description
    - Task name should be clarified in title. The general format is: [Prefix] Short description of the PR (Suffix)
    - Prefix: add new feature [Feature], fix bug [Fix], related to documents [Docs], in developing [WIP] (which will not be reviewed temporarily)
    - Introduce main changes, results and influences on other modules in short description
    - Associate related issues and pull requests with a milestone

CHAPTER

EIGHT

TRAINING TESTING TRICKS

MMYOLO has already supported most of the YOLO series object detection related algorithms. Different algorithms may involve some practical tricks. This section will describe in detail the commonly used training and testing tricks supported by MMYOLO based on the implemented object detection algorithms.

8.1 Training tricks

8.1.1 Improve performance of detection

1. Multi-scale training

In the field of object detection, multi-scale training is a very common trick. However, in YOLO, most of the models are trained with a single-scale input of 640x640. There are two reasons for this:

1. Single-scale training is faster than multi-scale training. When the training epoch is at 300 or 500, training efficiency is a major concern for users. Multi-scale training will be slower.
2. Multi-scale augmentation is implied in the training pipeline, which is equivalent to the application of multi-scale training, such as the 'Mosaic', 'RandomAffine' and 'Resize', so there is no need to introduce the multi-scale training of model input again.

Through experiments on the COCO dataset, it is founded that the multi-scale training is introduced directly after the output of YOLOv5's DataLoader, the actual performance improvement is very small. If you want to start multi-scale training for YOLO series algorithms in MMYOLO, you can refer to *ms_training_testing*, however, this does not mean that there are no significant gains in user-defined dataset fine-tuning mode

2 Use Mask annotation to optimize object detection performance

When the dataset annotation is complete, such as boundary box annotation and instance segmentation annotation exist at the same time, but only part of the annotation is required for the task, the task can be trained with complete data annotation to improve the performance. In object detection, we can also learn from instance segmentation annotation to improve the performance of object detection. The following is the detection result of additional instance segmentation annotation optimization introduced by YOLOv8. The performance gains are shown below:

As shown in the figure, different scale models have different degrees of performance improvement. It is important to note that 'Mask Refine' only functions in the data enhancement phase and does not require any changes to other training parts of the model and does not affect the speed of training. The details are as follows:

The above-mentioned Mask represents a data augmentation transformation in which instance segmentation annotations play a key role. The application of this technique to other YOLO series has varying degrees of increase.

3 Turn off strong augmentation in the later stage of training to improve detection performance

This strategy is proposed for the first time in YOLOX algorithm and can greatly improve the detection performance. The paper points out that Mosaic+MixUp can greatly improve the target detection performance, but the training pictures are far from the real distribution of natural pictures, and Mosaic's large number of cropping operations will bring many inaccurate label boxes, therefore, YOLOX proposes to turn off the strong enhancement in the last 15 epochs and use the weaker enhancement instead, so that the detector can avoid the influence of inaccurate labeled boxes and complete the final convergence under the data distribution of the natural picture.

This strategy has been applied to most YOLO algorithms. Taking YOLOv8 as an example, its data augmentation pipeline is shown as follows:

However, when to turn off the strong augmentation is a hyper-parameter. If you turn off the strong augmentation too early, it may not give full play to Mosaic and other strong augmentation effects. If you turn off the strong enhancement too late, it will have no gain because it has been overfitted before. This phenomenon can be observed in YOLOv8 experiment

As can be seen from the above table:

- Large models trained on COCO dataset for 500 epochs are prone to overfitting, and disabling strong augmentations such as Mosaic may not be effective in reducing overfitting in such cases.
- Using Mask annotations can alleviate overfitting and improve performance

4 Add pure background images to suppress false positives

For non-open-world datasets in object detection, both training and testing are conducted on a fixed set of classes, and there is a possibility of producing false positives when applied to images with classes that have not been trained. A common mitigation strategy is to add a certain proportion of pure background images. In most YOLO series, the function of suppressing false positives by adding pure background images is enabled by default. Users only need to set train_dataloader.dataset.filter_cfg.filter_empty_gt to False, indicating that pure background images should not be filtered out during training.

5 Maybe the AdamW works wonders

YOLOv5, YOLOv6, YOLOv7 and YOLOv8 all adopt the SGD optimizer, which is strict about parameter settings, while AdamW is on the contrary, which is not so sensitive to learning rate. If user fine-tune a custom-dataset can try to select the AdamW optimizer. We did a simple trial in YOLOX and found that replacing the optimizer with AdamW on the tiny, s, and m scale models all had some improvement.

More details can be found in configs/yolox/README.md.

6 Consider ignore scenarios to avoid uncertain annotations

Take CrowdHuman as an example, a crowded pedestrian detection dataset. Here's a typical image:

The image is sourced from detectron2 issue. The area marked with a yellow cross indicates the `iscrowd` label. There are two reasons for this:

- This area is not a real person, such as the person on the poster
- The area is too crowded to mark

In this scenario, you cannot simply delete such annotations, because once you delete them, it means treating them as background areas during training. However, they are different from the background. Firstly, the people on the posters are very similar to real people, and there are indeed people in crowded areas that are difficult to annotate. If you simply

train them as background, it will cause false negatives. The best approach is to treat the crowded area as an ignored region, where any output in this area is directly ignored, with no loss calculated and no model fitting enforced.

MMYOLO quickly and easily verifies the function of 'iscrowd' annotation on YOLOv5. The performance is as follows:

`ignore_iof_thr` set to -1 indicates that the ignored labels are not considered, and it can be seen that the performance is improved to a certain extent, more details can be found in CrowdHuman results. If you encounter similar situations in your custom dataset, it is recommended that you consider using `ignore` labels to avoid uncertain annotations.

7 Use knowledge distillation

Knowledge distillation is a widely used technique that can transfer the performance of a large model to a smaller model, thereby improving the detection performance of the smaller model. Currently, MMYOLO and MMRazor have supported this feature and conducted initial verification on RTMDet.

* indicates the result of using the large model distillation, more details can be found in Distill RTMDet.

8 Stronger augmentation parameters are used for larger models

If you have modified the model based on the default configuration or replaced the backbone network, it is recommended to scale the data augmentation parameters based on the current model size. Generally, larger models require stronger augmentation parameters, otherwise they may not fully leverage the benefits of large models. Conversely, if strong augmentations are applied to small models, it may result in underfitting. Taking RTMDet as an example, we can observe the data augmentation parameters for different model sizes.

`random_resize_ratio_range` represents the random scaling range of `RandomResize`, and `mosaic_max_cached_images/mixup_max_cached_images` represents the number of cached images during `Mosaic/MixUp` augmentation, which can be used to adjust the strength of augmentation. The YOLO series models all follow the same set of parameter settings principles.

8.1.2 Accelerate training speed

1 Enable cudnn_benchmark for single-scale training

Most of the input image sizes in the YOLO series algorithms are fixed, which is single-scale training. In this case, you can turn on cudnn_benchmark to accelerate the training speed. This parameter is mainly set for PyTorch's cuDNN underlying library, and setting this flag can allow the built-in cuDNN to automatically find the most efficient algorithm that is best suited for the current configuration to optimize the running efficiency. If this flag is turned on in multi-scale mode, it will continuously search for the optimal algorithm, which may slow down the training speed instead.

To enable `cudnn_benchmark` in MMYOLO, you can set `env_cfg = dict(cudnn_benchmark=True)` in the configuration.

2 Use Mosaic and MixUp with caching

If you have applied Mosaic and MixUp in your data augmentation, and after investigating the training bottleneck, it is found that the random image reading is causing the issue, then it is recommended to replace the regular Mosaic and MixUp with the cache-enabled versions proposed in RTMDet.

Mosaic and MixUp involve mixing multiple images, and their time consumption is K times that of ordinary data augmentation (K is the number of images mixed). For example, in YOLOv5, when doing Mosaic each time, the information of 4 images needs to be reloaded from the hard disk. However, the cached version of Mosaic and MixUp only needs to reload the current image, while the remaining images involved in the mixed augmentation are obtained from the cache queue, greatly improving efficiency by sacrificing a certain amount of memory space.

As shown in the figure, N preloaded images and label data are stored in the cache queue. In each training step, only one new image and its label data need to be loaded and updated in the cache queue. (Images in the cache queue can be duplicated, as shown in the figure with img3 appearing twice.) If the length of the cache queue exceeds the preset length, a random image will be popped out. When it is necessary to perform mixed data augmentation, only the required images need to be randomly selected from the cache for concatenation or other processing, without the need to load all images from the hard disk, thus saving image loading time.

8.1.3 Reduce the number of hyperparameter

YOLOv5 provides some practical methods for reducing the number of hyperparameter, which are described below.

1 Adaptive loss weighting, reducing one hyperparameter

In general, it can be challenging to set hyperparameters specifically for different tasks or categories. YOLOv5 proposes some adaptive methods for scaling loss weights based on the number of classes and the number of detection output layers have been proposed based on practical experience, as shown below:

```
# scaled based on number of detection layers
loss_cls=dict(
    type='mmdet.CrossEntropyLoss',
    use_sigmoid=True,
    reduction='mean',
    loss_weight=loss_cls_weight *
    (num_classes / 80 * 3 / num_det_layers)),
loss_bbox=dict(
    type='IoULoss',
    iou_mode='ciou',
    bbox_format='xywh',
    eps=1e-7,
    reduction='mean',
    loss_weight=loss_bbox_weight * (3 / num_det_layer
    return_iou=True),
loss_obj=dict(
    type='mmdet.CrossEntropyLoss',
    use_sigmoid=True,
    reduction='mean',
    loss_weight=loss_obj_weight *
    ((img_scale[0] / 640)**2 * 3 / num_det_layers)),
```

`loss_cls` can adaptively scale `loss_weight` based on the custom number of classes and the number of detection layers, `loss_bbox` can adaptively calculate based on the number of detection layers, and `loss_obj` can adaptively scale based on the input image size and the number of detection layers. This strategy allows users to avoid setting Loss weight hyperparameters. It should be noted that this is only an empirical principle and not necessarily the optimal setting combination, it should be used as a reference.

2 Adaptive Weight Decay and Loss output values base on Batch Size, reducing two hyperparameters

In general,when training on different `Batch Size`, it is necessary to follow the rule of automatic learning rate scaling. However, validation on various datasets shows that YOLOv5 can achieve good results without scaling the learning rate when changing the Batch Size, and sometimes scaling can even lead to worse results. The reason lies in the technique of `Weight Decay` and Loss output based on `Batch Size` adaptation in the code. In YOLOv5, `Weight Decay` and Loss output values will be scaled based on the total `Batch Size` being trained. The corresponding code is:

```
# https://github.com/open-mmlab/mmyolo/blob/dev/mmyolo/engine/optimizers/yolov5_optim_
↪constructor.py#L86
if 'batch_size_per_gpu' in optimizer_cfg:
    batch_size_per_gpu = optimizer_cfg.pop('batch_size_per_gpu')
    # No scaling if total_batch_size is less than
    # base_total_batch_size, otherwise linear scaling.
    total_batch_size = get_world_size() * batch_size_per_gpu
    accumulate = max(
        round(self.base_total_batch_size / total_batch_size), 1)
    scale_factor = total_batch_size * \
        accumulate / self.base_total_batch_size
    if scale_factor != 1:
        weight_decay *= scale_factor
        print_log(f'Scaled weight_decay to {weight_decay}', 'current')
```

```
# https://github.com/open-mmlab/mmyolo/blob/dev/mmyolo/models/dense_heads/yolov5_head.py
↪#L635
 _, world_size = get_dist_info()
 return dict(
     loss_cls=loss_cls * batch_size * world_size,
     loss_obj=loss_obj * batch_size * world_size,
     loss_bbox=loss_box * batch_size * world_size)
```

The weight of Loss varies in different Batch Sizes, and generally, the larger Batch Size means most larger the Loss and gradient. I personally speculate that this can be equivalent to a scenario of linearly increasing learning rate when Batch Size increases. In fact, from the YOLOv5 Study: mAP vs Batch-Size of YOLOv5, it can be found that it is desirable for users to achieve similar performance without modifying other parameters when modifying the Batch Size. The above two strategies are very good training techniques.

8.1.4 Save memory on GPU

How to reduce training memory usage is a frequently discussed issue, and there are many techniques involved. The training executor of MMYOLO comes from MMEngine, so you can refer to the MMEngine documentation for how to reduce training memory usage. Currently, MMEngine supports gradient accumulation, gradient checkpointing, and large model training techniques, details of which can be found in the SAVE MEMORY ON GPU.

8.2 Testing trick

8.2.1 Balance between inference speed and testing accuracy

During model performance testing, we generally require a higher mAP, but in practical applications or inference, we want the model to perform faster while maintaining low false positive and false negative rates. In other words, the testing only focuses on mAP while ignoring post-processing and evaluation speed, while in practical applications, a balance between speed and accuracy is pursued. In the YOLO series, it is possible to achieve a balance between speed and accuracy by controlling certain parameters. In this example, we will describe this in detail using YOLOv5.

1 Avoiding multiple class outputs for a single detection box during inference

YOLOv5 uses BCE Loss (use_sigmoid=True) during the training of the classification branch. Assuming there are 4 object categories, the number of categories output by the classification branch is 4 instead of 5. Moreover, due to the use of sigmoid instead of softmax prediction, it is possible to predict multiple detection boxes that meet the filtering threshold at a certain position, which means that there may be a situation where one predicted bbox corresponds to multiple predicted labels. This is shown in the figure below:

Generally, when calculating mAP, the filtering threshold is set to 0.001. Due to the non-competitive prediction mode of sigmoid, one box may correspond to multiple labels. This calculation method can increase the recall rate when calculating mAP, but it may not be convenient for practical applications.

One common approach is to increase the filtering threshold. However, if you don't want to have many false negatives, it is recommended to set the `multi_label` parameter to False. It is located in the configuration file at `mode.test_cfg.multi_label` and its default value is True, which allows one detection box to correspond to multiple labels.

2 Simplify test pipeline

Note that the test pipeline for YOLOv5 is as follows:

```
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

It uses two different Resizes with different functions, with the aim of improving the mAP value during evaluation. In actual deployment, you can simplify this pipeline as shown below:

```
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='LetterResize',
```

(continues on next page)

(continued from previous page)

```
        scale=_base_.img_scale,
        allow_scale_up=True,
        use_mini_pad=True),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

In practical applications, YOLOv5 algorithm uses a simplified pipeline with multi_label set to False, score_thr increased to 0.25, and iou_threshold reduced to 0.45. In the YOLOv5 configuration, we provide a set of configuration parameters for detection on the ground, as detailed in yolov5_s-v61_syncbn-detect_8xb16-300e_coco.py.

3 Batch Shape speeds up the testing speed

Batch Shape is a testing technique proposed in YOLOv5 that can speed up inference. The idea is to no longer require that all images in the testing process be 640x640, but to test at variable scales, as long as the shapes within the current batch are the same. This approach can reduce additional image pixel padding and speed up the inference process. The specific implementation of Batch Shape can be found in the link. Almost all algorithms in MMYOLO default to enabling the Batch Shape strategy during testing. If users want to disable this feature, you can set `val_dataloader.dataset.batch_shapes_cfg=None`.

In practical applications, because dynamic shape is not as fast and efficient as fixed shape. Therefore, this strategy is generally not used in real-world scenarios.

8.2.2 TTA improves test accuracy

Data augmentation with TTA (Test Time Augmentation) is a versatile trick that can improve the performance of object detection models and is particularly useful in competition scenarios. MMYOLO has already supported TTA, and it can be enabled simply by adding `--tta` when testing. For more details, please refer to the TTA.

CHAPTER

NINE

MODEL DESIGN INSTRUCTIONS

9.1 YOLO series model basic class

The structural figure is provided by RangeKing@GitHub. Thank you RangeKing

Most YOLO series algorithms adopt a unified algorithm-building structure, typically as Darknet + PAFPN. In order to let users quickly understand the YOLO series algorithm architecture, we deliberately designed the `BaseBackbone` + `BaseYOLONeck` structure, as shown in the above figure.

The benefits of the abstract `BaseBackbone` include:

1. Subclasses do not need to be concerned about the forward process. Just build the model as a builder pattern.
2. It can be configured to achieve custom plug-in functions. Users can easily insert some similar attention modules.
3. All subclasses automatically support freezing certain stages and bn functions.

`BaseYOLONeck` has the same benefits as `BaseBackbone`.

9.1.1 BaseBackbone

- As shown in Figure 1, for P5, `BaseBackbone` includes 1 stem layer and 4 stage layers which are similar to the basic structure of ResNet.
- As shown in Figure 2, for P6, `BaseBackbone` includes 1 stem layer and 5 stage layers. Different backbone network algorithms inherit the `BaseBackbone`. Users can build each layer of the whole network by implementing customized basic modules through the internal `build_xx` method.

9.1.2 BaseYOLONeck

We reproduce the YOLO series Neck components in the similar way as the `BaseBackbone`, and we can mainly divide them into `Reduce layer`, `UpSample layer`, `TopDown layer`, `DownSample layer`, `BottomUP layer` and `output convolution layer`. Each layer can be customized its internal construction by the inheritance and rewrite from the `build_xx` method.

9.1.3 BaseDenseHead

MMYOLO uses the `BaseDenseHead` designed in MMDetection as the base class of the Head structure. Take YOLOv5 as an example, the forward function of its HeadModule replaces the original forward method.

9.2 HeadModule

As shown in the above graph, the solid line is the implementation in MMYOLO, whereas the original implementation in MMDetection is shown in the dotted line. MMYOLO has the following advantages over the original implementation:

1. In MMDetection, `bbox_head` is split into three large components: `assigner` + `box coder` + `sampler`. But because the transfer between these three components is universal, it is necessary to encapsulate additional objects. With the unification in MMYOLO, users do not need to separate them. The advantages of not deliberately forcing the division of the three components are: data encapsulation of internal data is no longer required, code logic is simplified, and the difficulty of community use and algorithm reproduction is reduced.
2. MMYOLO is Faster. When users customize the implementation algorithm, they can deeply optimize part of the code without relying on the original framework.

In general, with the partly decoupled model + `loss_by_feat` part in MMYOLO, users can construct any model with any `loss_by_feat` by modifying the configuration. For example, applying the `loss_by_feat` of YOLOX to the YOLOv5 model, etc.

Take the YOLOX configuration in MMDetection as an example, the Head module configuration is written as follows:

```
bbox_head=dict(
    type='YOLOXHead',
    num_classes=80,
    in_channels=128,
    feat_channels=128,
    stacked_convs=2,
    strides=(8, 16, 32),
    use_depthwise=False,
    norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
    act_cfg=dict(type='Swish'),
    ...
    loss_obj=dict(
        type='CrossEntropyLoss',
        use_sigmoid=True,
        reduction='sum',
        loss_weight=1.0),
    loss_l1=dict(type='L1Loss', reduction='sum', loss_weight=1.0)),
train_cfg=dict(assigner=dict(type='SimOTAAssigner', center_radius=2.5)),
```

For the head_module in MMYOLO, the new configuration is written as follows:

```
bbox_head=dict(
    type='YOLOXHead',
    head_module=dict(
        type='YOLOXHeadModule',
        num_classes=80,
        in_channels=256,
        feat_channels=256,
        widen_factor=widen_factor,
```

(continues on next page)

(continued from previous page)

```
        stacked_convs=2,
        featmap_strides=(8, 16, 32),
        use_depthwise=False,
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='SiLU', inplace=True),
    ),
    ...
    loss_obj=dict(
        type='mmdet.CrossEntropyLoss',
        use_sigmoid=True,
        reduction='sum',
        loss_weight=1.0),
    loss_bbox_aux=dict(type='mmdet.L1Loss', reduction='sum', loss_weight=1.0)),
train_cfg=dict(
    assigner=dict(
        type='mmdet.SimOTAAssigner',
        center_radius=2.5,
        iou_calculator=dict(type='mmdet.BboxOverlaps2D'))),
```

CHAPTER

TEN

ALGORITHM PRINCIPLES AND IMPLEMENTATION

10.1 Algorithm principles and implementation with YOLOv5

10.1.1 0 Introduction

RangeKing@github provides the graph above. Thanks, RangeKing!

YOLOv5 is an open-source object detection algorithm for real-time industrial applications which has received extensive attention. The reason for the explosion of YOLOv5 is not simply due to its excellent performance. It is more about the overall utility and robustness of its library. In short, the main features of YOLOv5 are:

1. **Friendly and perfect deployment supports**
2. **Fast training speed**: the training time in the case of 300 epochs is similar to most of the one-stage and two-stage algorithms under 12 epochs, such as RetinaNet, ATSS, and Faster R-CNN.
3. **Abundant optimization for corner cases**: YOLOv5 has implemented many optimizations. The functions and documentation are richer as well.

Figures 1 and 2 show that the main differences between the P5 and P6 versions of YOLOv5 are the network structure and the image input resolution. Other differences, such as the number of anchors and loss weights, can be found in the configuration file. This article will start with the principle of the YOLOv5 algorithm and then focus on analyzing the implementation in MMYOLO. The follow-up part includes the guide and speed benchmark of YOLOv5.

Hint: Unless specified, the P5 model is described by default in this documentation.

We hope this article becomes your core document to start and master YOLOv5. Since YOLOv5 is still constantly updated, we will also keep updating this document. So please always catch up with the latest version.

MMYOLO implementation configuration: https://github.com/open-mmlab/mmyolo/blob/main/configs/yolov5/

YOLOv5 official repository: https://github.com/ultralytics/yolov5

10.1.2 1 v6.1 algorithm principle and MMYOLO implementation analysis

YOLOv5 official release: https://github.com/ultralytics/yolov5/releases/tag/v6.1

The performance is shown in the table above. YOLOv5 has two models with different scales. P6 is larger with a 1280x1280 input size, whereas P5 is the model used more often. This article focuses on the structure of the P5 model.

Usually, we divide the object detection algorithm into different parts, such as data augmentation, model structure, loss calculation, etc. It is the same as YOLOv5:

Now we will briefly analyze the principle and our specific implementation in MMYOLO.

1.1 Data augmentation

Many data augmentation methods are used in YOLOv5, including:

- **Mosaic**
- **RandomAffine**
- **MixUp**
- **Image blur and other transformations using Albu**
- **HSV color space enhancement**
- **Random horizontal flips**

The mosaic probability is set to `1`, so it will always be triggered. MixUp is not used for the small and nano models, and the probability is `0.1` for other l/m/x series models. As small models have limited capabilities, we generally do not use strong data augmentations like MixUp.

The following picture demonstrates the `Mosaic + RandomAffine + MixUp` process.

1.1.1 Mosaic

Mosaic is a hybrid data augmentation method requiring four images to be stitched together, which is equivalent to increasing the training batch size.

We can summarize the process as:

1. Randomly generates coordinates of the intersection point of the four spliced images.
2. Randomly select the indexes of the other three images and read the corresponding annotations.
3. Resizes each image to the specified size by maintaining its aspect ratio.
4. Calculate the position of each image in the output image according to the top, bottom, left, and right rule. You also need to calculate the crop coordinates because the image may be out of bounds.
5. Uses the crop coordinates to crop the scaled image and paste it to the position calculated. The rest of the places will be pad with `114 pixels`.
6. Process the label of each image accordingly.

Note: since four images are stitched together, the output image area will be enlarged four times (from 640x640 to 1280x1280). Therefore, to revert to 640x640, you must add a **RandomAffine** transformation. Otherwise, the image area will always be four times larger.

1.1.2 RandomAffine

RandomAffine has two purposes:

1. Performs a stochastic geometric affine transformation to the image.
2. Reduces the size of the image generated by Mosaic back to 640x640.

RandomAffine includes geometric augmentations such as translation, rotation, scaling, misalignment, etc. Since Mosaic and RandomAffine are strong augmentations, they will introduce considerable noise. Therefore, the enhanced annotations need to be processed. The rules are

1. The width and height of the enhanced gt bbox should be larger than wh_thr;
2. The ratio of the area of gt bbox after and before the enhancement should be greater than ar_thr to prevent it from changing too much.
3. The maximum aspect ratio should be smaller than area_thr to prevent it from changing too much.

Object detection algorithms will rarely use this augmentation method as the annotation box becomes larger after the rotation, resulting in inaccuracy.

1.1.3 MixUp

MixUp, similar to Mosaic, is also a hybrid image augmentation. It randomly selects another image and mixes the two images together. There are various ways to do this, and the typical approach is to either stitch the label together directly or mix the label using `alpha` method. The original author's approach is straightforward: the label is directly stitched, and the images are mixed by distributional sampling.

Note: **In YOLOv5's implementation of MixUP, the other random image must be processed by Mosaic+RandomAffine before the mixing process.** This may not be the same as implementations in other open-source libraries.

1.1.4 Image blur and other augmentations

The rest of the augmentations are:

- **Image blur and other transformations using Albu**
- **HSV color space enhancement**
- **Random horizontal flips**

The Albu library has been packaged in MMDetection so users can directly use all Albu's methods through simple configurations. As a very ordinary and common processing method, HSV will not be further introduced now.

1.1.5 The implementations in MMYOLO

While conventional single-image augmentations such as random flip are relatively easy to implement, hybrid data augmentations like Mosaic are more complicated. Therefore, in MMDetection's reimplementation of YOLOX, a dataset wrapper called `MultiImageMixDataset` was introduced. The process is as follows:

For hybrid data augmentations such as Mosaic, you need to implement an additional `get_indexes` method to retrieve the index information of other images and then perform the enhancement. Take the YOLOX implementation in MMDetection as an example. The configuration file is like this:

```
train_pipeline = [
    dict(type='Mosaic', img_scale=img_scale, pad_val=114.0),
    dict(
        type='RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
    dict(
        type='MixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0),
    ...
]

train_dataset = dict(
    # use MultiImageMixDataset wrapper to support mosaic and mixup
    type='MultiImageMixDataset',
    dataset=dict(
        type='CocoDataset',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations', with_bbox=True)
        ]),
    pipeline=train_pipeline)
```

MultiImageMixDataset passes in a data augmentation method, including Mosaic and RandomAffine. CocoDataset also adds a pipeline to load the images and the annotations. This way, it is possible to quickly achieve a hybrid data augmentation method.

However, the above implementation has one drawback: **For users unfamiliar with MMDetection, they often forget that Mosaic must be used with MultiImageMixDataset. Otherwise, it will return an error. Plus, this approach increases the complexity and difficulty of understanding**.

To solve this problem, we have simplified it further in MMYOLO. By making the dataset object directly accessible to the pipeline, the implementation and the use of hybrid data augmentations can be the same as random flipping.

The configuration of YOLOX in MMYOLO is written as follows:

```
pre_transform = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True)
]

train_pipeline = [
    *pre_transform,
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='mmdet.RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
```

(continues on next page)

(continued from previous page)

```
    dict(
        type='YOLOXMixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0,
        pre_transform=pre_transform),
    ...
]
```

This eliminates the need for the MultiImageMixDataset and makes it much easier to use and understand.

Back to the YOLOv5 configuration, since the other randomly selected image in the MixUp also needs to be enhanced by Mosaic+RandomAffine before it can be used, the YOLOv5-m data enhancement configuration is as follows.

```
pre_transform = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True)
]

mosaic_transform= [
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='YOLOv5RandomAffine',
        max_rotate_degree=0.0,
        max_shear_degree=0.0,
        scaling_ratio_range=(0.1, 1.9),  # scale = 0.9
        border=(-img_scale[0] // 2, -img_scale[1] // 2),
        border_val=(114, 114, 114))
]

train_pipeline = [
    *pre_transform,
    *mosaic_transform,
    dict(
        type='YOLOv5MixUp',
        prob=0.1,
        pre_transform=[
            *pre_transform,
            *mosaic_transform
        ]),
    ...
]
```

1.2 Network structure

This section was written by RangeKing@github. Thanks a lot!

The YOLOv5 network structure is the standard `CSPDarknet` + `PAFPN` + `non-decoupled Head`.

The size of the YOLOv5 network structure is determined by the `deepen_factor` and `widen_factor` parameters. `deepen_factor` controls the depth of the network structure, that is, the number of stacks of `DarknetBottleneck` modules in `CSPLayer`. `widen_factor` controls the width of the network structure, that is, the number of channels of the module output feature map. Take YOLOv5-l as an example. Its `deepen_factor = widen_factor = 1.0`. the overall structure is shown in the graph above.

The upper part of the figure is an overview of the model; the lower part is the specific network structure, in which the modules are marked with numbers in serial, which is convenient for users to correspond to the configuration files of the YOLOv5 official repository. The middle part is the detailed composition of each sub-module.

If you want to use **netron** to visualize the details of the network structure, open the ONNX file format exported by MMDeploy in netron.

Hint: The shapes of the feature map in Section 1.2 are (B, C, H, W) by default.

1.2.1 Backbone

`CSPDarknet` in MMYOLO inherits from `BaseBackbone`. The overall structure is similar to `ResNet` with a total of 5 layers of design, including one `Stem Layer` and four `Stage Layer`:

- `Stem Layer` is a `ConvModule` whose kernel size is 6x6. It is more efficient than the `Focus` module used before v6.1.
- Except for the last `Stage Layer`, each `Stage Layer` consists of one `ConvModule` and one `CSPLayer`, as shown in the Details part in the graph above. `ConvModule` is a 3x3 `Conv2d` + `BatchNorm` + `SiLU activation function` module. `CSPLayer` is the C3 module in the official YOLOv5 repository, consisting of three `ConvModule` + n `DarknetBottleneck` with residual connections.
- The last `Stage Layer` adds an `SPPF` module at the end. The `SPPF` module is to serialize the input through multiple 5x5 `MaxPool2d` layers, which has the same effect as the `SPP` module but is faster.
- The P5 model passes the corresponding results from the second to the fourth `Stage Layer` to the `Neck` structure and extracts three output feature maps. Take a 640x640 input image as an example. The output features are (B, 256, 80, 80), (B,512,40,40), and (B,1024,20,20). The corresponding stride is 8/16/32.
- The P6 model passes the corresponding results from the second to the fifth `Stage Layer` to the `Neck` structure and extracts three output feature maps. Take a 1280x1280 input image as an example. The output features are (B, 256, 160, 160), (B,512,80,80), (B,768,40,40), and (B,1024,20,20). The corresponding stride is 8/16/32/64.

1.2.2 Neck

There is no **Neck** part in the official YOLOv5. However, to facilitate users to correspond to other object detection networks easier, we split the `Head` of the official repository into `PAFPN` and `Head`.

Based on the `BaseYOLONeck` structure, YOLOv5's `Neck` also follows the same build process. However, for non-existed modules, we use `nn.Identity` instead.

The feature maps output by the Neck module is the same as the Backbone. The P5 model is (B,256,80,80), (B,512,40,40) and (B,1024,20,20); the P6 model is (B,256,160,160), (B,512,80,80), (B,768,40,40) and (B,1024,20,20).

1.2.3 Head

The `Head` structure of YOLOv5 is the same as YOLOv3, which is a `non-decoupled Head`. The Head module includes three convolution modules that do not share weights. They are used only for input feature map transformation.

The `PAFPN` outputs three feature maps of different scales, whose shapes are (B,256,80,80), (B,512,40,40), and (B,1024,20,20) accordingly.

Since YOLOv5 has a non-decoupled output, that is, classification and bbox detection results are all in different channels of the same convolution module. Taking the COCO dataset as an example:

- When the input of P5 model is 640x640 resolution, the output shapes of the Head module are `(B, 3x(4+1+80), 80,80)`, `(B, 3x(4+1+80),40,40)` and `(B, 3x(4+1+80),20,20)`.
- When the input of P6 model is 1280x1280 resolution, the output shapes of the Head module are `(B, 3x(4+1+80),160,160)`, `(B, 3x(4+1+80),80,80)`, `(B, 3x(4+1+80),40,40)` and `(B, 3x(4+1+80),20, 20)`.

  `3` represents three anchors, `4` represents the bbox prediction branch, `1` represents the obj prediction branch, and `80` represents the class prediction branch of the COCO dataset.

1.3 Positive and negative sample assignment strategy

The core of the positive and negative sample assignment strategy is to determine which positions in all positions of the predicted feature map should be positive or negative and even which samples will be ignored.

This is one of the most significant components of the object detection algorithm because a good strategy can improve the algorithm's performance.

The assignment strategy of YOLOv5 can be briefly summarized as calculating the shape-matching rate between anchor and gt_bbox. Plus, the cross-neighborhood grid is also introduced to get more positive samples.

It consists of the following two main steps:

1. For any output layer, instead of the commonly used strategy based on Max IoU matching, YOLOv5 switched to comparing the shape matching ratio. First, the GT Bbox and the anchor of the current layer are used to calculate the aspect ratio. If the ratio is greater than the threshold, the GT Bbox and Anchor are considered not matched. Then the current GT Bbox is temporarily discarded, and the predicted position in the grid of this GT Bbox in the current layer is regarded as a negative sample.
2. For the remaining GT Bboxes (the matched GT Bboxes), YOLOv5 calculates which grid they fall in. Using the rounding rule to find the nearest two grids and considering all three grids as a group that is responsible for predicting the GT Bbox. The number of positive samples has increased by at least three times compared to the previous YOLO series algorithms.

Now we will explain each part of the assignment strategy in detail. Some descriptions and illustrations are directly or indirectly referenced from the official repo.

1.3.1 Anchor settings

YOLOv5 is an anchor-based object detection algorithm. Similar to YOLOv3, the anchor sizes are still obtained by clustering. However, the difference compared with YOLOv3 is that instead of clustering based on IoU, YOLOv5 switched to using the aspect ratio on the width and height (shape-match based method).

While training on customized data, user can use the tool in MMYOLO to analyze and get the appropriate anchor sizes of the dataset.

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} --algorithm v5-k-means
 --input-shape ${INPUT_SHAPE [WIDTH HEIGHT]} --output-dir ${OUTPUT_DIR}
```

Then modify the default anchor size setting in the config file:

```
anchors = [[(10, 13), (16, 30), (33, 23)], [(30, 61), (62, 45), (59, 119)],
           [(116, 90), (156, 198), (373, 326)]]
```

1.3.2 Bbox encoding and decoding process

The predicted bounding box will transform based on the pre-set anchors in anchor-based algorithms. Then, the transformation amount is predicted, known as the GT Bbox encoding process. Finally, the Pred Bbox decoding needs to be performed after the prediction to restore the bboxes to the original scale, known as the Pred Bbox decoding process.

In YOLOv3, the bbox regression formula is:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= a_w \cdot e^{t_w} \\
b_h &= a_h \cdot e^{t_h}
\end{aligned}
$$

In the above formula

$$
\begin{aligned}
a_w represents the width of the anchor \\
c_x represents the coordinate of the grid \\
\sigma represents the Sigmoid function.
\end{aligned}
$$

However, the regression formula in YOLOv5 is

$$
\begin{aligned}
b_x &= (2 \cdot \sigma(t_x) - 0.5) + c_x \\
b_y &= (2 \cdot \sigma(t_y) - 0.5) + c_y \\
b_w &= a_w \cdot (2 \cdot \sigma(t_w))^2 \\
b_h &= a_h \cdot (2 \cdot \sigma(t_h))^2
\end{aligned}
$$

Two main changes are:

- adjusted the range of the center point coordinate from (0, 1) to (-0.5, 1.5);
- adjusted the width and height from

$$(0 + \infty)$$

to

$$(04a_{wh})$$

The changes have the two benefits:

- It will be **better to predict zero and one** with the changed center point range, which makes the bbox coordinate regression more accurate.
- `exp(x)` in the width and height regression formula is unbounded, which may cause the **gradient out of control** and make the training stage unstable. The revised width-height regression in YOLOv5 optimizes this problem.

1.3.3 Assignment strategy

Note: in MMYOLO, **we call anchor as prior** for both anchor-based and anchor-free networks.

Positive sample assignment consists of the following two steps:

(1) Scale comparison

Compare the scale of the WH in the GT BBox and the WH in the Prior:

$$
\begin{aligned}
r_w &= w\_gt / w\_pt \\
r_h &= h\_gt / h\_pt \\
r_w^{max} &= max(r_w, 1/r_w) \\
r_h^{max} &= max(r_h, 1/r_h) \\
r^{max} &= max(r_w^{max}, r_h^{max}) \\
if\ \ r_{max} &< prior\_match\_thr : match!
\end{aligned}
$$

Taking the assignment process of the GT Bbox and the Prior of the P3 feature map as the example:

The reason why Prior 1 fails to match the GT Bbox is because:

$$
h\_gt\ /\ h\_prior\ =\ 4.8\ >\ prior\_match\_thr
$$

(2) Assign corresponded positive samples to the matched GT BBox in step 1

We still use the example in the previous step.

The value of (cx, cy, w, h) of the GT BBox is (26, 37, 36, 24), and the WH value of the Prior is [(15, 5), (24, 16), (16, 24)]. In the P3 feature map, the stride is eight. Prior 2 and prior 3 are matched.

The detailed process can be described as:

(2.1) Map the center point coordinates of the GT Bbox to the grid of P3.

$$
\begin{aligned}
GT_x^{center_grid} &= 26/8 = 3.25 \\
GT_y^{center_grid} &= 37/8 = 4.625
\end{aligned}
$$

(2.2) Divide the grid where the center point of GT Bbox locates into four quadrants. **Since the center point falls in the lower left quadrant, the left and lower grids of the object will also be considered positive samples**.

The following picture shows the distribution of positive samples when the center point falls to different positions:

So what improvements does the Assign method bring to YOLOv5?

- One GT Bbox can match multiple Priors.
- When a GT Bbox matches a Prior, at most three positive samples can be assigned.
- These strategies can **moderately alleviate the problem of unbalanced positive and negative samples, which is very common in object detection algorithms**.

The regression method in YOLOv5 corresponds to the Assign method:

1. Center point regression:
2. WH regression:

1.4 Loss design

YOLOv5 contains a total of three Loss, which are:

- Classes loss: BCE loss
- Objectness loss: BCE loss
- Location loss: CIoU loss

These three losses are aggregated according to a certain proportion:

$$Loss = \lambda_1 L_{cls} + \lambda_2 L_{obj} + \lambda_3 L_{loc}$$

The Objectness loss corresponding to the P3, P4, and P5 layers are added according to different weights. The default setting is

```
obj_level_weights=[4., 1., 0.4]
```

$$L_{obj} = 4.0 \cdot L_{obj}^{small} + 1.0 \cdot L_{obj}^{medium} + 0.4 \cdot L_{obj}^{large}$$

In the reimplementation, we found a certain gap between the CIoU used in YOLOv5 and the latest official CIoU, which is reflected in the calculation of the alpha parameter.

In the official version:

Reference: https://github.com/Zzh-tju/CIoU/blob/master/layers/modules/multibox_loss.py#L53-L55

```
alpha = (ious > 0.5).float() * v / (1 - ious + v)
```

In YOLOv5's version:

```
alpha = v / (v - ious + (1 + eps))
```

This is an interesting detail, and we need to test the accuracy gap caused by different alpha calculation methods in our follow-up development.

1.5 Optimization and training strategies

YOLOv5 has very fine-grained control over the parameter groups of each optimizer, which briefly includes the following sections.

1.5.1 Optimizer grouping

The optimization parameters are divided into three groups: Conv/Bias/BN. In the WarmUp stage, different groups use different lr and momentum update curves. At the same time, the iter-based update strategy is adopted in the WarmUp stage, and it becomes an epoch-based update strategy in the non-WarmUp stage, which is quite tricky.

In MMYOLO, the YOLOv5OptimizerConstructor optimizer constructor is used to implement optimizer parameter grouping. The role of an optimizer constructor is to control the initialization process of some special parameter groups finely so that it can meet the needs well.

Different parameter groups use different scheduling curve functions through YOLOv5ParamSchedulerHook.

1.5.2 weight decay parameter auto-adaptation

The author adopts different weight decay strategies for different batch sizes, specifically:

1. When the training batch size does not exceed 64, weight decay remains unchanged.
2. When the training batch size exceeds 64, weight decay will be linearly scaled according to the total batch size.

MMYOLO also implements through the YOLOv5OptimizerConstructor.

1.5.3 Gradient accumulation

To maximize the performance under different batch sizes, the author sets the gradient accumulation function automatically when the total batch size is less than 64.

The training process is similar to most YOLO, including the following strategies:

1. Not using pre-trained weights.
2. There is no multi-scale training strategy, and cudnn.benchmark can be turned on to accelerate training further.
3. The EMA strategy is used to smooth the model.
4. Automatic mixed-precision training with AMP by default.

What needs to be reminded is that the official YOLOv5 repository uses single-card v100 training for the small model with a bs is 128. However, m/l/x models are trained with different numbers of multi-cards. This training strategy is not relatively standard, **For this reason, eight cards are used in MMYOLO, and each card sets the bs to 16. At the same time, in order to avoid performance differences, SyncBN is turned on during training**.

1.6 Inference and post-processing

The YOLOv5 post-processing is very similar to YOLOv3. In fact, all post-processing stages of the YOLO series are similar.

1.6.1 Core parameters

1. **multi_label**

For multi-category prediction, you need to consider whether it is a multi-label case or not. Multi-label case predicts probabilities of more than one category at one location. As YOLOv5 uses sigmoid, it is possible that one object may have two different predictions. It is good to evaluate mAP, but not good to use. Therefore, multi_label is set to `True` during the evaluation and changed to `False` for inferencing and practical usage.

2. **score_thr and nms_thr**

The score_thr threshold is used for the score of each category, and the detection boxes with a score below the threshold are treated as background. nms_thr is used for nms process. During the evaluation, score_thr can be set very low, which improves the recall and the mAP. However, it is meaningless for practical usage and leads to a very slow inference performance. For this reason, different thresholds are set in the testing and inference phases.

3. **nms_pre and max_per_img**

nms_pre is the maximum number of frames to be preserved before NMS, which is used to prevent slowdown caused by too many input frames during the NMS process. max_per_img is the final maximum number of frames to be reserved, usually set to 300.

Take the COCO dataset as an example. It has 80 classes, and the input size is 640x640.

The inference and post-processing include:

(1) Dimensional transformation

YOLOv5 outputs three feature maps. Each feature map is scaled at 80x80, 40x40, and 20x20. As three anchors are at each position, the output feature map channel is 3x(5+80)=255. YOLOv5 uses a non-decoupled Head, while most other algorithms use decoupled Head. Therefore, to unify the post-processing logic, we decouple YOLOv5's Head into the category prediction branch, the bbox prediction branch, and the obj prediction branch.

The three scales of category prediction, bbox prediction, and obj prediction are stitched together and dimensionally transformed. For subsequent processing, the original channel dimensions are replaced at the end, and the shapes of the category prediction branch, bbox prediction branch, and obj prediction branch are (b, 3x80x80+3x40x40+3x20x20, 80)=(b,25200,80), (b,25200,4), and (b,25200,1), respectively.

(2) Decoding to the original graph scale

The classification branch and obj branch need to be computed with the sigmoid function, while the bbox prediction branch needs to be decoded and reduced to the original image in xyxy format.

(3) First filtering

Iterate through each graph in the batch, and then use score_thr to threshold filter the category prediction scores to remove the prediction results below score_thr.

(4) Second filtering

Multiply the obj prediction scores and the filtered category prediction scores, and then still use score_thr for threshold filtering. It is also necessary to consider **multi_label and nms_pre in this process to ensure that the number of detected boxes after filtering is no more than nms_pre**.

(5) Rescale to original size and NMS

Based on the pre-processing process, restore the remaining detection frames to the original graph scale before the network output and perform NMS. The final output detection frame cannot be more than **max_per_img**.

1.6.2 batch shape strategy

To speed up the inference process on the validation set, the authors propose the batch shape strategy, whose principle is to **ensure that the images within the same batch have the least number of pad pixels in the batch inference process and do not require all the images in the batch to have the same scale throughout the validation process**.

It first sorts images according to their aspect ratio of the entire test or validation set, and then forms a batch of the sorted images based on the settings. At the same time, the batch shape of the current batch is calculated to prevent too many pad pixels. We focus on padding with the original aspect ratio but not padding the image to a perfect square.

```
        image_shapes = []
        for data_info in data_list:
            image_shapes.append((data_info['width'], data_info['height']))

        image_shapes = np.array(image_shapes, dtype=np.float64)

        n = len(image_shapes)  # number of images
        batch_index = np.floor(np.arange(n) / self.batch_size).astype(
            np.int64)  # batch index
        number_of_batches = batch_index[-1] + 1  # number of batches

        aspect_ratio = image_shapes[:, 1] / image_shapes[:, 0]  # aspect ratio
        irect = aspect_ratio.argsort()
```

(continues on next page)

(continued from previous page)

```
        data_list = [data_list[i] for i in irect]

        aspect_ratio = aspect_ratio[irect]
        # Set training image shapes
        shapes = [[1, 1]] * number_of_batches
        for i in range(number_of_batches):
            aspect_ratio_index = aspect_ratio[batch_index == i]
            min_index, max_index = aspect_ratio_index.min(
            ), aspect_ratio_index.max()
            if max_index < 1:
                shapes[i] = [max_index, 1]
            elif min_index > 1:
                shapes[i] = [1, 1 / min_index]

        batch_shapes = np.ceil(
            np.array(shapes) * self.img_size / self.size_divisor +
            self.pad).astype(np.int64) * self.size_divisor

        for i, data_info in enumerate(data_list):
            data_info['batch_shape'] = batch_shapes[batch_index[i]]
```

10.1.3 2 Sum up

This article focuses on the principle of YOLOv5 and our implementation in MMYOLO in detail, hoping to help users understand the algorithm and the implementation process. At the same time, again, please note that since YOLOv5 itself is constantly being updated, this open-source library will also be continuously iterated. So please always check the latest version.

10.2 Algorithm principles and implementation with YOLOv8

10.2.1 0 Introduction

RangeKing@github provides the graph above. Thanks, RangeKing!

YOLOv8 is the next major update from YOLOv5, open sourced by Ultralytics on 2023.1.10, and now supports image classification, object detection and instance segmentation tasks.

However, instead of naming the open source library YOLOv8, ultralytics uses the word ultralytics directly because ultralytics positions the library as an algorithmic framework rather than a specific algorithm, with a major focus on scalability. It is expected that the library can be used not only for the YOLO model family, but also for non-YOLO models and various tasks such as classification segmentation pose estimation.

Overall, YOLOv8 is a powerful and flexible tool for object detection and image segmentation that offers the best of both worlds: **the SOTA technology and the ability to use and compare all previous YOLO versions.**

YOLOv8 official open source address: this

MMYOLO open source address for YOLOv8: this

The following table shows the official results of mAP, number of parameters and FLOPs tested on the COCO Val 2017 dataset. It is evident that YOLOv8 has significantly improved precision compared to YOLOv5. However, the number

of parameters and FLOPs of the N/S/M models have significantly increased. Additionally, it can be observed that the inference speed of YOLOv8 is slower in comparison to most of the YOLOv5 models.

It is worth mentioning that the recent YOLO series have shown significant performance improvements on the COCO dataset. However, their generalizability on custom datasets has not been extensively tested, which thereby will be a focus in the future development of MMYOLO.

Before reading this article, if you are not familiar with YOLOv5, YOLOv6 and RTMDet, you can read the detailed explanation of YOLOv5 and its implementation.

10.2.2 1 YOLOv8 Overview

The core features and modifications of YOLOv8 can be summarized as follows:

1. **A new state-of-the-art (SOTA) model is proposed, featuring an object detection model for P5 640 and P6 1280 resolutions, as well as a YOLACT-based instance segmentation model. The model also includes different size options with N/S/M/L/X scales, similar to YOLOv5, to cater to various scenarios.**
2. **The backbone network and neck module are based on the YOLOv7 ELAN design concept, replacing the C3 module of YOLOv5 with the C2f module. However, there are a lot of operations such as Split and Concat in this C2f module that are not as deployment-friendly as before.**
3. **The Head module has been updated to the current mainstream decoupled structure, separating the classification and detection heads, and switching from Anchor-Based to Anchor-Free.**
4. **The loss calculation adopts the TaskAlignedAssigner in TOOD and introduces the Distribution Focal Loss to the regression loss.**
5. **In the data augmentation part, Mosaic is closed in the last 10 training epoch, which is the same as YOLOX training part. As can be seen from the above summaries, YOLOv8 mainly refers to the design of recently proposed algorithms such as YOLOX, YOLOv6, YOLOv7 and PPYOLOE.**

Next, we will introduce various improvements in the YOLOv8 model in detail by 5 parts: model structure design, loss calculation, training strategy, model inference process and data augmentation.

10.2.3 2 Model structure design

The Figure 1 is the model structure diagram based on the official code of YOLOv8. **If you like this style of model structure diagram, welcome to check out the model structure diagram in algorithm README of MMYOLO, which currently covers YOLOv5, YOLOv6, YOLOX, RTMDet and YOLOv8.**

Comparing the YOLOv5 and YOLOv8 yaml configuration files without considering the head module, you can see that the changes are minor.

The structure on the left is YOLOv5-s and the other side is YOLOv8-s. The specific changes in the backbone network and neck module are:

- The kernel of the first convolutional layer has been changed from 6x6 to 3x3
- All C3 modules are replaced by C2f, and the structure is as follows, with more skip connections and additional split operations.
- Removed 2 convolutional connection layers from neck module
- The block number has been changed from 3-6-9-3 to 3-6-6-3.
- **If we look at the N/S/M/L/X models, we can see that of the N/S and L/X models only changed the scaling factors, but the number of channels in the S/ML backbone network is not the same and does not follow the same scaling factor principle. The main reason for this design is that the channel settings under the same**

set of scaling factors are not the most optimal, and the YOLOv7 network design does not follow one set of scaling factors for all models either.

The most significant changes in the model lay in the head module. The head module has been changed from the original coupling structure to the decoupling one, and its style has been changed from **YOLOv5's Anchor-Based to Anchor-Free**. The structure is shown below.

As demonstrated, the removal of the objectness branch and the retention of only the decoupled classification and regression branches stand as the major differences. Additionally, the regression branch now employs integral form representation as proposed in the Distribution Focal Loss.

10.2.4 3 Loss calculation

The loss calculation process consists of 2 parts: the sample assignment strategy and loss calculation.

The majority of contemporary detectors employ dynamic sample assignment strategies, such as YOLOX's simOTA, TOOD's TaskAlignedAssigner, and RTMDet's DynamicSoftLabelAssigner. Given the superiority of dynamic assignment strategies, the YOLOv8 algorithm directly incorporates the one employed in TOOD's TaskAlignedAssigner.

The matching strategy of TaskAlignedAssigner can be summarized as follows: positive samples are selected based on the weighted scores of classification and regression.

$$t = s^{\alpha} + u^{\beta}$$

`s` is the prediction score corresponding to the ground truth category, `u` is the IoU of the prediction bounding box and the gt bounding box.

1. For each ground truth, the task-aligned assigner calculates the `alignment metric` for each anchor by taking the weighted product of two values: the predicted classification score of the corresponding class, and the Intersection over Union (IoU) between the predicted bounding box and the Ground Truth bounding box.
2. For each Ground Truth, the larger top-k samples are selected as positive based on the `alignment_metrics` values directly.

The loss calculation consists of 2 parts: the classification and regression, without the objectness loss in the previous model.

- The classification branch still uses BCE Loss.
- The regression branch employs both Distribution Focal Loss and CIoU Loss.

The 3 Losses are weighted by a specific weight ratio.

10.2.5 4 Data augmentation

YOLOv8's data augmentation is similar to YOLOv5, whereas it stops the Mosaic augmentation in the final 10 epochs as proposed in YOLOX. The data process pipelines are illustrated in the diagram below.

The intensity of data augmentation required for different scale models varies, therefore the hyperparameters for the scaled models are adjusted depending on the situation. For larger models, techniques such as MixUp and CopyPaste are typically employed. The result of data augmentation can be seen in the example below:

The above visualization result can be obtained by running the browse_dataset script.

As the data augmentation process utilized in YOLOv8 is similar to YOLOv5, we will not delve into the specifics within this article. For a more in-depth understanding of each data transformation, we recommend reviewing the YOLOv5 algorithm analysis document in MMYOLO.

10.2.6 5 Training strategy

The distinctions between the training strategy of YOLOv8 and YOLOv5 are minimal. The most notable variation is that the overall number of training epochs for YOLOv8 has been raised from 300 to 500, resulting in a significant expansion in the duration of training. As an illustration, the training strategy for YOLOv8-S can be succinctly outlined as follows:

10.2.7 6 Inference process

The inference process of YOLOv8 is almost the same as YOLOv5. The only difference is that the integral representation bbox in Distribution Focal Loss needs to be decoded into a regular 4-dimensional bbox, and the subsequent calculation process is the same as YOLOv5.

Taking COCO 80 class as an example, assuming that the input image size is 640x640, the inference process implemented in MMYOLO is shown as follows.

(1) Decoding bounding box Integrate the probability of the distance between the center and the boundary of the box into the mathematical expectation of the distances.

(2) Dimensional transformation YOLOv8 outputs three feature maps with `80x80`, `40x40` and `20x20` scales. A total of 6 classification and regression different scales of feature map are output by the head module. The 3 different scales of category prediction branch and bbox prediction branch are combined and dimensionally transformed. For the convenience of subsequent processing, the original channel dimensions are transposed to the end, and the category prediction branch and bbox prediction branch shapes are (b, 80x80+40x40+20x20, 80)=(b,8400,80), (b,8400,4), respectively.

(3) Scale Restroation The classification prediction branch utilizes sigmoid calculations, whereas the bbox prediction branch requires decoding to xyxy format and conversion to the original scale of the input images.

(4) Thresholding Iterate through each graph in the batch and use `score_thr` to perform thresholding. In this process, we also need to consider multi_label and nms_pre to ensure that the number of detected bboxs after filtering is no more than nms_pre.

(5) Reduction to the original image scale and NMS Reusing the parameters for preprocessing, the remaining bboxs are first resized to the original image scale and then NMS is performed. The final number of bboxes cannot be more than `max_per_img`.

Special Note: **The Batch shape inference strategy, which is present in YOLOv5, is currently not activated in YOLOv8. By performing a quick test in MMYOLO, it can be observed that activating the Batch shape strategy can result in an approximate AP increase of around 0.1% to 0.2%.**

10.2.8 7 Feature map visualization

A comprehensive set of feature map visualization tools are provided in MMYOLO to help users visualize the feature maps.

Take the YOLOv8-s model as an example. The first step is to download the official weights, and then convert them to MMYOLO by using the yolov8_to_mmyolo script. Note that the script must be placed under the official repository in order to run correctly.

Assuming that you want to visualize the effect of the 3 feature maps output by backbone and the weights are named 'mmyolov8s.pth'. Run the following command:

```
cd mmyolo
python demo/featmap_vis_demo.py demo/demo.jpg configs/yolov8/yolov8_s_syncbn_fast_8xb16-
↪500e_coco.py mmyolov8s.pth --channel-reductio squeeze_mean
```

In particular, to ensure that the feature map and image are shown aligned, the original `test_pipeline` configuration needs to be replaced with the following:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # change
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

```
cd mmyolo
python demo/featmap_vis_demo.py demo/demo.jpg configs/yolov8/yolov8_s_syncbn_fast_8xb16-
↪500e_coco.py mmyolov8s.pth --channel-reductio squeeze_mean --target-layers neck
```

From the above figure, we can find the features at the object are more focused.

10.2.9 Summary

This article delves into the intricacies of the YOLOv8 algorithm, offering a comprehensive examination of its overall design, model structure, loss function, training data enhancement techniques, and inference process. To aid in comprehension, a plethora of diagrams are provided.

In summary, YOLOv8 is a highly efficient algorithm that incorporates image classification, Anchor-Free object detection, and instance segmentation. Its detection component incorporates numerous state-of-the-art YOLO algorithms to achieve new levels of performance.

MMYOLO open source address for YOLOV8 this

MMYOLO Algorithm Analysis Tutorial address is yolov5_description

10.3 Algorithm principles and implementation with RTMDet

10.3.1 0 Introduction

High performance, low latency one-stage object detection

RangeKing@github provides the graph above. Thanks, RangeKing!

Recently,the open-source community has spring up a large number of high-precision object detection projects, one of the most prominent projects is YOLO series. OpenMMLab has also launched MMYOLO in collaboration with the community. After investigating many improved models in current YOLO series, MMDetection core developers empirically summarized these designs and training methods, and optimized them to launch a single-stage object detector with high accuracy and low latency RTMDet, **R**eal-**t**ime **M**odels for Object **Det**ection (**R**elease **t**o **M**anufacture)

RTMDet consists of a series of tiny/s/m/l/x models of different sizes, which provide different choices for different application scenarios. Specifically, RTMDet-x achieves a 300+ FPS inference speed with an accuracy of 52.6 mAP.

Note: Note: Inference speed and accuracy test (excluding NMS) were performed on `TensorRT 8.4.3, cuDNN 8.2.0, FP16, batch size=1` on 1 NVIDIA 3090 GPU.

The lightest model, RTMDet-tiny, can achieve 40.9 mAP with only 4M parameters and inference speed < 1 ms.

The accuracy in this figure is a fair comparison to 300 training epochs, without distillation.

- Official repository: https://github.com/open-mmlab/mmdetection/blob/3.x/configs/rtmdet/README.md
- MMYOLO repository: https://github.com/open-mmlab/mmyolo/blob/main/configs/rtmdet/README.md

10.3.2 1 v1.0 algorithm principle and MMYOLO implementation analysis

1.1 Data augmentation

Many data augmentation methods are used in RTMDet, mainly include single image data augmentation:

- **RandomResize**
- **RandomCrop**
- **HSVRandomAug**
- **RandomFlip**

and mixed image data augmentation:

- **Mosaic**
- **MixUp**

The following picture demonstrates the data augmentation process:

The RandomResize hyperparameters are different on the large models M,L,X and the small models S, Tiny. Due to the number of parameters,the large models can use the `large jitter scale strategy` with parameters of (0.1,2.0). The small model adopts the `stand scale jitter` strategy with parameters of (0.5, 2.0).

The single image data augmentation has been packaged in `MMDetection` so users can directly use all methods through simple configurations. As a very ordinary and common processing method, this part will not be further introduced now. The implementation of mixed image data augmentation is described in the following.

Unlike YOLOv5, which considers the use of MixUp on S and Nano models is excessive. Small models don't need such strong data augmentation. However, RTMDet also uses MixUp on S and Tiny, because RTMDet will switch to normal aug at last 20 epochs, and this operation was proved to be effective by training. Moreover, RTMDet introduces a Cache scheme for mixed image data augmentation, which effectively reduces the image processing time and introduces adjustable hyperparameters.

`max_cached_images`, which is similar to `repeated augmentation` when using a smaller cache. The details are as follows:

1.1.1 Introducing Cache for mixins data augmentation

Mosaic&MixUp needs to blend multiple images, which takes k times longer than common data augmentation (k is the number of images mixed in). For example, in YOLOv5, every time Mosaic is done, the information of four images needs to be reloaded from the hard disk. RTMDet only needs to reload the current image, and the rest images participating in the mixed augmentation are obtained from the cache queue, which greatly improves the efficiency by sacrificing a certain memory space. Moreover, we can modify the cache size and pop mode to adjust the strength of augmentation.

As shown in the figure, N loaded images and labels are stored in the cache queue in advance. In each training step, only a new image and its label need to be loaded and updated to the cache queue (the images in the cache queue can be repeated, as shown in the figure for img3 twice). Meanwhile, if the cache queue length exceeds the preset length, it will pop a random image (in order to make the Tiny model more stable, the Tiny model doesn't use the random pop, but removes the first added image). When mixed data augmentation is needed, only the required images need to be randomly selected from the cache for splicing and other processing, instead of loading them all from the hard disk, which saves the time of image loading.

Note: The maximum length N of the cache queue is an adjustable parameter. According to the empirical principle, when ten caches are provided for each image to be blended, it can be considered to provide enough randomness, while the Mosaic enhancement is four image blends, so the number of caches defaults to N=40. Similarly, MixUp has a default cache size of 20, but tiny model requires more stable training conditions, so it has half cache size of other specs (10 for MixUp and 20 for Mosaic).

In the implementation, MMYOLO designed the `BaseMiximageTransform` class to support mixed data augmentation of multiple images:

```
if self.use_cached:
    # Be careful: deep copying can be very time-consuming
    # if results includes dataset.
    dataset = results.pop('dataset', None)
    self.results_cache.append(copy.deepcopy(results))  # Cache the currently loaded data
    if len(self.results_cache) > self.max_cached_images:
        if self.random_pop: # Except for the tiny model, self.random_pop=True
            index = random.randint(0, len(self.results_cache) - 1)
        else:
            index = 0
        self.results_cache.pop(index)

    if len(self.results_cache) <= 4:
        return results
else:
    assert 'dataset' in results
    # Be careful: deep copying can be very time-consuming
    # if results includes dataset.
    dataset = results.pop('dataset', None)
```

1.1.2 Mosaic

Mosaic concatenates four images into a large image, which is equivalent to increasing the batch size, as follows:

1. Randomly resample three images from customize datasets based on the index, possibly repeated.

```
def get_indexes(self, dataset: Union[BaseDataset, list]) -> list:
    """Call function to collect indexes.

    Args:
        dataset (:obj:`Dataset` or list): The dataset or cached list.

    Returns:
        list: indexes.
    """
    indexes = [random.randint(0, len(dataset)) for _ in range(3)]
    return indexes
```

2. Randomly select the midpoint of the intersection of four images.

```
# mosaic center x, y
center_x = int(
    random.uniform(*self.center_ratio_range) * self.img_scale[1])
center_y = int(
    random.uniform(*self.center_ratio_range) * self.img_scale[0])
center_position = (center_x, center_y)
```

3. Read and concatenate images based on the sampled index. Using the `keep-ratio` resize image (i.e. the maximum edge must be 640) before concatenating.

```
# keep_ratio resize
scale_ratio_i = min(self.img_scale[0] / h_i,
                    self.img_scale[1] / w_i)
img_i = mmcv.imresize(
    img_i, (int(w_i * scale_ratio_i), int(h_i * scale_ratio_i)))
```

4. After concatenating images, the bboxes and labels are all concatenated together, and then the bboxes are cropped but not filtered (some invalid bboxes may appear).

```
mosaic_bboxes.clip_([2 * self.img_scale[0], 2 * self.img_scale[1]])
```

Please reference the Mosaic theory of *YOLOv5* for more details.

1.1.3 MixUp

The MixUp implementation of RTMDet is the same as YOLOX, with the addition of cache function similar to above mentioned.

Please reference the MixUp theory of *YOLOv5* for more details.

1.1.4 Strong and weak two-stage training

Mosaic + MixUp has high distortion. Continuously using strong data augmentation isn't beneficial. YOLOX use strong and weak two-stage training mode firstly. However, the introduction of rotation and shear result in box annotation errors, which needs to introduce L1 loss to correct the performance of regression branch.

In order to make the data augmentation method more general, RTMDet uses Mosaic + MixUp without rotation during the first 280 epochs, and increases the intensity and positive samples by mixing eight images. During the last 20 epochs, a relatively small learning rate is used to fine-tune under weak agumentation, and slowly update parameters to model by EMA, which could obtain a large improvement.

1.2 Model structure

CHAPTER

ELEVEN

MMYOLO APPLICATION EXAMPLES

11.1 A benchmark for ionogram real-time object detection based on MMYOLO

11.1.1 Dataset

Digital ionogram is the most important way to obtain real-time ionospheric information. Ionospheric structure detection is of great research significance for accurate extraction of ionospheric key parameters.

This study utilize 4311 ionograms with different seasons obtained by the Chinese Academy of Sciences in Hainan, Wuhan, and Huailai to establish a dataset. The six structures, including Layer E, Es-l, Es-c, F1, F2, and Spread F are manually annotated using labelme. Dataset Download

Preview of annotated images

1. Dataset prepration

After downloading the data, put it in the root directory of the MMYOLO repository, and use `unzip test.zip` (for Linux) to unzip it to the current folder. The structure of the unzipped folder is as follows:

```
Iono4311/
├── images
|       ├── 20130401005200.png
|       └── ...
└── labels
        ├── 20130401005200.json
        └── ...
```

The `images` directory contains input imageswhile the `labels` directory contains annotation files generated by labelme.

2. Convert the dataset into COCO format

Use the script `tools/dataset_converters/labelme2coco.py` to convert labelme labels to COCO labels.

```
python tools/dataset_converters/labelme2coco.py --img-dir ./Iono4311/images \
                                                --labels-dir ./Iono4311/labels \
                                                --out ./Iono4311/annotations/annotations_
↪all.json
```

3. Check the converted COCO labels

To confirm that the conversion process went successfully, use the following command to display the COCO labels on the images.

```
python tools/analysis_tools/browse_coco_json.py --img-dir ./Iono4311/images \
                                                --ann-file ./Iono4311/annotations/
↪annotations_all.json
```

4. Divide dataset into training set, validation set and test set

Set 70% of the images in the dataset as the training set, 15% as the validation set, and 15% as the test set.

```
python tools/misc/coco_split.py --json ./Iono4311/annotations/annotations_all.json \
                                --out-dir ./Iono4311/annotations \
                                --ratios 0.7 0.15 0.15 \
                                --shuffle \
                                --seed 14
```

The file tree after division is as follows:

```
Iono4311/
├── annotations
│   ├── annotations_all.json
│   ├── class_with_id.txt
│   ├── test.json
│   ├── train.json
│   └── val.json
├── classes_with_id.txt
├── images
├── labels
├── test_images
├── train_images
└── val_images
```

11.1.2 Config files

The configuration files are stored in the directory `/projects/misc/ionogram_detection/`.

1. Dataset analysis

To perform a dataset analysis, a sample of 200 images from the dataset can be analyzed using the `tools/analysis_tools/dataset_analysis.py` script.

```
python tools/analysis_tools/dataset_analysis.py projects/misc/ionogram_detection/yolov5/
↪yolov5_s-v61_fast_1xb96-100e_ionogram.py \
                                                --out-dir output
```

Part of the output is as follows:

```
The information obtained is as follows:
+------------------------------+
| Information of dataset class |
+---------------+--------------+
| Class name    | Bbox num     |
+---------------+--------------+
| E             | 98           |
| Es-l          | 27           |
| Es-c          | 46           |
```

(continues on next page)

(continued from previous page)

```
| F1             | 100           |
| F2             | 194           |
| Spread-F       | 6             |
+----------------+---------------+
```

This indicates that the distribution of categories in the dataset is unbalanced.

Statistics of object sizes for each category

According to the statistics, small objects are predominant in the E, Es-l, Es-c, and F1 categories, while medium-sized objects are more common in the F2 and Spread F categories.

2. Visualization of the data processing part in the config

Taking YOLOv5-s as an example, according to the `train_pipeline` in the config file, the data augmentation strategies used during training include

- Mosaic augmentation
- Random affine
- Albumentations (include various digital image processing methods)
- HSV augmentation
- Random affine

Use the **'pipeline'** mode of the script `tools/analysis_tools/browse_dataset.py` to obtains all intermediate images in the data pipeline.

```
python tools/analysis_tools/browse_dataset.py projects/misc/ionogram_detection/yolov5/
↪yolov5_s-v61_fast_1xb96-100e_ionogram.py \
                                              -m pipeline \
                                              --out-dir output
```

Visualization for intermediate images in the data pipeline

3. Optimize anchor size

Use the script `tools/analysis_tools/optimize_anchors.py` to obtain prior anchor box sizes suitable for the dataset.

```
python tools/analysis_tools/optimize_anchors.py projects/misc/ionogram_detection/yolov5/
↪yolov5_s-v61_fast_1xb96-100e_ionogram.py \
                                                --algorithm v5-k-means \
                                                --input-shape 640 640 \
                                                --prior-match-thr 4.0 \
                                                --out-dir work_dirs/dataset_analysis_5_s
```

4. Model complexity analysis

With the config file, the parameters and FLOPs can be calculated by the script `tools/analysis_tools/get_flops.py`. Take yolov5-s as an example:

```
python tools/analysis_tools/get_flops.py projects/misc/ionogram_detection/yolov5/yolov5_
↪s-v61_fast_1xb96-100e_ionogram.py
```

The following output indicates that the model has 7.947G FLOPs with the input shape (640, 640), and a total of 7.036M learnable parameters.

```
==============================
Input shape: torch.Size([640, 640])
Model Flops: 7.947G
Model Parameters: 7.036M
==============================
```

11.1.3 Train and test

1. Train

Training visualization: By following the tutorial of Annotation-to-deployment workflow for custom dataset, this example uses wandb to visulize training.

Debug tricks: During the process of debugging code, sometimes it is necessary to train for several epochs, such as debugging the validation process or checking whether the checkpoint saving meets expectations. For datasets inherited from `BaseDataset` (such as `YOLOv5CocoDataset` in this example), setting `indices` in the `dataset` field can specify the number of samples per epoch to reduce the iteration time.

```
train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        _delete_=True,
        type='RepeatDataset',
        times=1,
        dataset=dict(
            type=_base_.dataset_type,
            indices=200,  # set indices=200represent every epoch only iterator 200␣
↪samples
            data_root=data_root,
            metainfo=metainfo,
            ann_file=train_ann_file,
            data_prefix=dict(img=train_data_prefix),
            filter_cfg=dict(filter_empty_gt=False, min_size=32),
            pipeline=_base_.train_pipeline)))
```

Start training

```
python tools/train.py projects/misc/ionogram_detection/yolov5/yolov5_s-v61_fast_1xb96-
↪100e_ionogram.py
```

2. Test

Specify the path of the config file and the model to start the test:

```
python tools/test.py projects/misc/ionogram_detection/yolov5/yolov5_s-v61_fast_1xb96-
↪100e_ionogram.py \
                    work_dirs/yolov5_s-v61_fast_1xb96-100e_ionogram/xxx
```

11.1.4 Experiments and results

Choose a suitable batch size

- Often, the batch size governs the training speed, and the ideal batch size will be the largest batch size supported by the available hardware.
- If the video memory is not yet fully utilized, doubling the batch size should result in a corresponding doubling (or close to doubling) of the training throughput. This is equivalent to maintaining a constant (or nearly constant) time per step as the batch size increases.
- Automatic Mixed Precision (AMP) is a technique to accelerate the training with minimal loss in accuracy. To enable AMP training, add `--amp` to the end of the training command.

Hardware information:

- GPUV100 with 32GB memory
- CPU10-core CPU with 40GB memory

Results

The proportion of data loading time to the total time of each step.

Based on the results above, we can conclude that

- AMP has little impact on the accuracy of the model, but can significantly reduce memory usage while training.
- Increasing batch size by three times does not reduce the training time by a corresponding factor of three. According to the `data_time` recorded during training, the larger the batch size, the larger the `data_time`, indicating that data loading has become the bottleneck limiting the training speed. Increasing `num_workers`, the number of processes used to load data, can accelerate the training speed.

Ablation studies

In order to obtain a training pipeline applicable to the dataset, the following ablation studies with the YOLOv5-s model as an example are performed.

Data augmentation

The results indicate that mosaic augmentation and random affine transformation can significantly improve the performance on the validation set.

Using pre-trained models

If you prefer not to use pre-trained weights, you can simply set `load_from = None` in the config file. For experiments that do not use pre-trained weights, it is recommended to increase the base learning rate by a factor of four and extend the number of training epochs to 200 to ensure adequate model training.

Comparison of loss reduction during training

The loss reduction curve shows that when using pre-trained weights, the loss decreases faster. It can be seen that even using models pre-trained on natural image datasets can accelerate model convergence when fine-tuned on radar image datasets.

Benchmark for ionogram object detection

CHAPTER

TWELVE

REPLACE THE BACKBONE NETWORK

Note:

1. When using other backbone networks, you need to ensure that the output channels of the backbone network match the input channels of the neck network.
2. The configuration files given below only ensure that the training will work correctly, and their training performance may not be optimal. Because some backbones require specific learning rates, optimizers, and other hyperparameters. Related contents will be added in the "Training Tips" section later.

12.1 Use backbone network implemented in MMYOLO

Suppose you want to use `YOLOv6EfficientRep` as the backbone network of `YOLOv5`, the example config is as the following:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    backbone=dict(
        type='YOLOv6EfficientRep',
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='ReLU', inplace=True))
)
```

12.2 Use backbone network implemented in other OpenMMLab repositories

The model registry in MMYOLO, MMDetection, MMClassification, and MMSegmentation all inherit from the root registry in MMEngine in the OpenMMLab 2.0 system, allowing these repositories to directly use modules already implemented by each other. Therefore, in MMYOLO, users can use backbone networks from MMDetection and MMClassification without reimplementation.

12.2.1 Use backbone network implemented in MMDetection

1. Suppose you want to use `ResNet-50` as the backbone network of `YOLOv5`, the example config is as the following:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [512, 1024, 2048]

model = dict(
    backbone=dict(
        _delete_=True, # Delete the backbone field in _base_
        type='mmdet.ResNet', # Using ResNet from mmdet
        depth=50,
        num_stages=4,
        out_indices=(1, 2, 3),
        frozen_stages=1,
        norm_cfg=dict(type='BN', requires_grad=True),
        norm_eval=True,
        style='pytorch',
        init_cfg=dict(type='Pretrained', checkpoint='torchvision://resnet50')),
    neck=dict(
        type='YOLOv5PAFPN',
        widen_factor=widen_factor,
        in_channels=channels, # Note: The 3 channels of ResNet-50 output are [512, 1024,␣
↪2048], which do not match the original yolov5-s neck and need to be changed.
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # input channels of head need to be changed accordingly
            widen_factor=widen_factor))
)
```

2. Suppose you want to use `SwinTransformer-Tiny` as the backbone network of `YOLOv5`, the example config is as the following:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [192, 384, 768]
checkpoint_file = 'https://github.com/SwinTransformer/storage/releases/download/v1.0.0/
↪swin_tiny_patch4_window7_224.pth'  # noqa

model = dict(
    backbone=dict(
        _delete_=True, # Delete the backbone field in _base_
        type='mmdet.SwinTransformer', # Using SwinTransformer from mmdet
        embed_dims=96,
        depths=[2, 2, 6, 2],
        num_heads=[3, 6, 12, 24],
```

(continues on next page)

(continued from previous page)

```
        window_size=7,
        mlp_ratio=4,
        qkv_bias=True,
        qk_scale=None,
        drop_rate=0.,
        attn_drop_rate=0.,
        drop_path_rate=0.2,
        patch_norm=True,
        out_indices=(1, 2, 3),
        with_cp=False,
        convert_weights=True,
        init_cfg=dict(type='Pretrained', checkpoint=checkpoint_file)),
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # Note: The 3 channels of SwinTransformer-Tiny output are␣
↪[192, 384, 768], which do not match the original yolov5-s neck and need to be changed.
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # input channels of head need to be changed accordingly
            widen_factor=widen_factor))
)
```

12.2.2 Use backbone network implemented in MMClassification

1. Suppose you want to use `ConvNeXt-Tiny` as the backbone network of `YOLOv5`, the example config is as the following:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# please run the command, mim install "mmcls>=1.0.0rc2", to install mmcls
# import mmcls.models to trigger register_module in mmcls
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)
checkpoint_file = 'https://download.openmmlab.com/mmclassification/v0/convnext/
↪downstream/convnext-tiny_3rdparty_32xb128-noema_in1k_20220301-795e9634.pth'  # noqa
deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [192, 384, 768]

model = dict(
    backbone=dict(
        _delete_=True, # Delete the backbone field in _base_
        type='mmcls.ConvNeXt', # Using ConvNeXt from mmcls
        arch='tiny',
        out_indices=(1, 2, 3),
        drop_path_rate=0.4,
        layer_scale_init_value=1.0,
```

(continues on next page)

(continued from previous page)

```
        gap_before_final_norm=False,
        init_cfg=dict(
            type='Pretrained', checkpoint=checkpoint_file,
            prefix='backbone.')), # The pre-trained weights of backbone network in MMCls␣
↪have prefix='backbone.'. The prefix in the keys will be removed so that these weights␣
↪can be normally loaded.
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # Note: The 3 channels of ConvNeXt-Tiny output are [192,␣
↪384, 768], which do not match the original yolov5-s neck and need to be changed.
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # input channels of head need to be changed accordingly
            widen_factor=widen_factor))
)
```

2. Suppose you want to use `MobileNetV3-small` as the backbone network of `YOLOv5`, the example config is as the following:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# please run the command, mim install "mmcls>=1.0.0rc2", to install mmcls
# import mmcls.models to trigger register_module in mmcls
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)
checkpoint_file = 'https://download.openmmlab.com/mmclassification/v0/mobilenet_v3/
↪convert/mobilenet_v3_small-8427ecf0.pth'  # noqa
deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [24, 48, 96]

model = dict(
    backbone=dict(
        _delete_=True, # Delete the backbone field in _base_
        type='mmcls.MobileNetV3', # Using MobileNetV3 from mmcls
        arch='small',
        out_indices=(3, 8, 11), # Modify out_indices
        init_cfg=dict(
            type='Pretrained',
            checkpoint=checkpoint_file,
            prefix='backbone.')), # The pre-trained weights of backbone network in MMCls␣
↪have prefix='backbone.'. The prefix in the keys will be removed so that these weights␣
↪can be normally loaded.
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # Note: The 3 channels of MobileNetV3 output are [24, 48,␣
↪96], which do not match the original yolov5-s neck and need to be changed.
```

(continues on next page)

(continued from previous page)

```
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # input channels of head need to be changed accordingly
            widen_factor=widen_factor))
)
```

12.2.3 Use backbone network in `timm` through MMClassification

MMClassification also provides a wrapper for the PyTorch **Im**age **M**odels (`timm`) backbone network, users can directly use the backbone network in `timm` through MMClassification. Suppose you want to use `EfficientNet-B1` as the backbone network of `YOLOv5`, the example config is as the following:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# please run the command, mim install "mmcls>=1.0.0rc2", to install mmcls
# and the command, pip install timm, to install timm
# import mmcls.models to trigger register_module in mmcls
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [40, 112, 320]

model = dict(
    backbone=dict(
        _delete_=True, # Delete the backbone field in _base_
        type='mmcls.TIMMBackbone', # Using timm from mmcls
        model_name='efficientnet_b1', # Using efficientnet_b1 in timm
        features_only=True,
        pretrained=True,
        out_indices=(2, 3, 4)),
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # Note: The 3 channels of EfficientNet-B1 output are [40,␣
↪112, 320], which do not match the original yolov5-s neck and need to be changed.
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # input channels of head need to be changed accordingly
            widen_factor=widen_factor))
)
```

12.2.4 Use backbone network implemented in MMSelfSup

Suppose you want to use `ResNet-50` which is self-supervised trained by `MoCo v3` in MMSelfSup as the backbone network of `YOLOv5`, the example config is as the following:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# please run the command, mim install "mmselfsup>=1.0.0rc3", to install mmselfsup
# import mmselfsup.models to trigger register_module in mmselfsup
custom_imports = dict(imports=['mmselfsup.models'], allow_failed_imports=False)
checkpoint_file = 'https://download.openmmlab.com/mmselfsup/1.x/mocov3/mocov3_resnet50_
↪8xb512-amp-coslr-800e_in1k/mocov3_resnet50_8xb512-amp-coslr-800e_in1k_20220927-
↪e043f51a.pth'  # noqa
deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [512, 1024, 2048]

model = dict(
    backbone=dict(
        _delete_=True, # Delete the backbone field in _base_
        type='mmselfsup.ResNet',
        depth=50,
        num_stages=4,
        out_indices=(2, 3, 4), # Note: out_indices of ResNet in MMSelfSup are 1 larger␣
↪than those in MMdet and MMCls
        frozen_stages=1,
        norm_cfg=dict(type='BN', requires_grad=True),
        norm_eval=True,
        style='pytorch',
        init_cfg=dict(type='Pretrained', checkpoint=checkpoint_file)),
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # Note: The 3 channels of ResNet-50 output are [512, 1024,␣
↪2048], which do not match the original yolov5-s neck and need to be changed.
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # input channels of head need to be changed accordingly
            widen_factor=widen_factor))
)
```

12.2.5 Don't used pre-training weights

When we replace the backbone network, the model initialization is trained by default loading the pre-training weight of the backbone network. Instead of using the pre-training weights of the backbone network, if you want to train the time model from scratch, You can set `init_cfg` in 'backbone' to 'None'. In this case, the backbone network will be initialized with the default initialization method, instead of using the trained pre-training weight.

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [512, 1024, 2048]

model = dict(
   backbone=dict(
       _delete_=True, # Delete the backbone field in _base_
       type='mmdet.ResNet', # Using ResNet from mmdet
       depth=50,
       num_stages=4,
       out_indices=(1, 2, 3),
       frozen_stages=1,
       norm_cfg=dict(type='BN', requires_grad=True),
       norm_eval=True,
       style='pytorch',
       init_cfg=None # If init_cfg is set to None, backbone will not be initialized with␣
↪pre-trained weights
   ),
   neck=dict(
       type='YOLOv5PAFPN',
       widen_factor=widen_factor,
       in_channels=channels, # Note: The 3 channels of ResNet-50 output are [512, 1024,␣
↪2048], which do not match the original yolov5-s neck and need to be changed.
       out_channels=channels),
   bbox_head=dict(
       type='YOLOv5Head',
       head_module=dict(
           type='YOLOv5HeadModule',
           in_channels=channels, # input channels of head need to be changed accordingly
           widen_factor=widen_factor))
)
```

CHAPTER

THIRTEEN

MODEL COMPLEXITY ANALYSIS

We provide a `tools/analysis_tools/get_flops.py` script to help with the complexity analysis for models of MMYOLO. Currently, it provides the interfaces to compute parameter, activation and flops of the given model, and supports printing the related information layer-by-layer in terms of network structure or table.

The commands as follows:

```
python tools/analysis_tools/get_flops.py
    ${CONFIG_FILE} \                            # config file path
    [--shape ${IMAGE_SIZE}] \                   # input image size (int), default 640*640
    [--show-arch ${ARCH_DISPLAY}] \             # print related information by network␣
↪layers
    [--not-show-table ${TABLE_DISPLAY}] \       # print related information by table
    [--cfg-options ${CFG_OPTIONS}]              # config file option
# [] stands for optional parameter, do not type [] when actually entering the command␣
↪line
```

Let's take the `rtmdet_s_syncbn_fast_8xb32-300e_coco.py` config file in RTMDet as an example to show how this script can be used:

13.1 Usage Example 1: Print Flops, Parameters and related information by table

```
python tools/analysis_tools/get_flops.py  configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_
↪coco.py
```

Output:

```
==============================
Input shape: torch.Size([640, 640])
Model Flops: 14.835G
Model Parameters: 8.887M
==============================
```

13.2 Usage Example 2: Print related information by network layers

```
python tools/analysis_tools/get_flops.py  configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_
↪coco.py --show-arch
```

Due to the complex structure of RTMDet, the output is long. The following shows only the output from bbox_head.head_module.rtm_reg section:

```
(rtm_reg): ModuleList(
        #params: 1.55K, #flops: 4.3M, #acts: 33.6K
        (0): Conv2d(
          128, 4, kernel_size=(1, 1), stride=(1, 1)
          #params: 0.52K, #flops: 3.28M, #acts: 25.6K
        )
        (1): Conv2d(
          128, 4, kernel_size=(1, 1), stride=(1, 1)
          #params: 0.52K, #flops: 0.82M, #acts: 6.4K
        )
        (2): Conv2d(
          128, 4, kernel_size=(1, 1), stride=(1, 1)
          #params: 0.52K, #flops: 0.2M, #acts: 1.6K
        )
```

CHAPTER

FOURTEEN

ANNOTATION-TO-DEPLOYMENT WORKFLOW FOR CUSTOM DATASET

In our daily work and study, we often encounter some tasks that need to train custom dataset. There are few scenarios in which open-source datasets can be used as online models, so we need to carry out a series of operations on our custom datasets to ensure that the models can be put into production and serve users.

See also:

The video of this document has been posted on Bilibili: A nanny level tutorials for custom datasets from annotationt to deployment

---

Note: All instructions in this document are done on Linux and are fully available on Windows, only slightly different in commands and operations.

Default that you have completed the installation of MMYOLO, if not installed, please refer to the document GET STARTED for installation.

In this tutorial, we will introduce the whole process from annotating custom dataset to final training, testing and deployment. The overview steps are as below:

1. Prepare dataset: `tools/misc/download_dataset.py`
2. Use the software of labelme to annotate: `demo/image_demo.py` + labelme
3. Convert the dataset into COCO format: `tools/dataset_converters/labelme2coco.py`
4. Split dataset:`tools/misc/coco_split.py`
5. Creat a config file based on dataset
6. Dataset visualization analysis: `tools/analysis_tools/dataset_analysis.py`
7. Optimize Anchor size: `tools/analysis_tools/optimize_anchors.py`
8. Visualization the data processing part of config: `tools/analysis_tools/browse_dataset.py`
9. Train: `tools/train.py`
10. Inference: `demo/image_demo.py`
11. Deployment

Note: After obtaining the model weight and the mAP of validation set, users need to deep analyse the bad cases of incorrect predictions in order to optimize model. MMYOLO will add this function in the future. Expect.

Each step is described in detail below.

14.1 1. Prepare custom dataset

- If you don't have your own dataset, or want to use a small dataset to run the whole process, you can use the 144 images `cat` dataset provided with this tutorial (the raw picture of this dataset is supplied by @RangeKing, cleaned by @PeterH0323). This `cat` dataset will be used as an example for the rest tutorial.

The download is also very simple, requiring only one command (dataset compression package size `217 MB`):

```
python tools/misc/download_dataset.py --dataset-name cat --save-dir ./data/cat --unzip --
↪delete
```

This dataset is automatically downloaded to the `./data/cat` dir with the following directory structure:

```
.
└── ./data/cat
    ├── images # image files
    │   ├── image1.jpg
    │   ├── image2.png
    │   └── ...
    ├── labels # labelme files
    │   ├── image1.json
    │   ├── image2.json
    │   └── ...
    ├── annotations # annotated files of COCO
    │   ├── annotations_all.json # all labels of COCO
    │   ├── trainval.json # 80% labels of the dataset
    │   └── test.json # 20% labels of the dataset
    └── class_with_id.txt # id + class_name file
```

This dataset can be trained directly. You can remove everything **outside** the `images` dir if you want to go through the whole process.

- If you already have a dataset, you can compose it into the following structure:

```
.
└── $DATA_ROOT
    └── images
        ├── image1.jpg
        ├── image2.png
        └── ...
```

14.2 2. Use the software of labelme to annotate

In general, there are two annotation methods:

- Software or algorithmic assistance + manual correction (Recommend, reduce costs and speed up)
- Only manual annotation

Note: At present, we also consider to access third-party libraries to support the integration of algorithm-assisted annotation and manual optimized annotation by calling MMYOLO inference API through GUI interface. If you have any interest or ideas, please leave a comment in the issue or contact us directly!

14.2.1 2.1 Software or algorithmic assistance + manual correction

The principle is using the existing model to inference, and save the result as label file. Manually operating the software and loading the generated label files, you only need to check whether each image is correctly labeled and whether there are missing objects.assistance + manual correctionyou can save a lot of time in order to **reduce costs and speed up** by this way.

Note: If the existing model doesn't have the categories defined in your dataset, such as COCO pre-trained model, you can manually annotate 100 images to train an initial model, and then software assistance.

The process is described below:

2.1.1 Software or algorithmic assistance

MMYOLO provide model inference script `demo/image_demo.py`. Setting `--to-labelme` to generate labelme format label file:

```
python demo/image_demo.py img \
                          config \
                          checkpoint
                          [--out-dir OUT_DIR] \
                          [--device DEVICE] \
                          [--show] \
                          [--deploy] \
                          [--score-thr SCORE_THR] \
                          [--class-name CLASS_NAME]
                          [--to-labelme]
```

These include:

- `img` image path, supported by dir, file, URL;
- `config`config file path of model;
- `checkpoint`weight file path of model;
- `--out-dir`inference results saved in this dir, default as `./output`, if set this `--show` parameter, the detection results are not saved;
- `--device`cumputing resources, including `CUDA`, `CPU` etc., default as `cuda:0`;
- `--show`display the detection results, default as `False`
- `--deploy`whether to switch to deploy mode;
- `--score-thr`confidence threshold, default as `0.3`;
- `--to-labelme`whether to export label files in `labelme` format, shouldn't exist with the `--show` at the same time.

For example:

Here, we'll use YOLOv5-s as an example to help us label the 'cat' dataset we just downloaded. First, download the weights for YOLOv5-s:

```
mkdir work_dirs
wget https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth -P ./work_
↪dirs
```

(continues on next page)

(continued from previous page)

Since the COCO 80 dataset already includes the `cat` class, we can directly load the COCO pre-trained model for assistant annotation.

```
python demo/image_demo.py ./data/cat/images \
                          ./configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
                          ./work_dirs/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                          --out-dir ./data/cat/labels \
                          --class-name cat \
                          --to-labelme
```

Tip:

- If your dataset needs to label with multiclass, you can use this `--class-name class1 class2` format;
- Removing the `--class-name` flag to output all classes.

the generated label files saved in `--out-dir`:

```
.
└── $OUT_DIR
    ├── image1.json
    ├── image1.json
    └── ...
```

Here is an example of the original image and it's generating json file:

2.1.2 Manual annotation

In this tutorial, we use labelme to annotate

- Install labelme

```
conda create -n labelme python=3.8
conda activate labelme
pip install labelme==5.1.1
```

- Start labelme

```
labelme ${image dir path (same as the previous step)} \
        --output ${the dir path of label filesame as --out-dir} \
        --autosave \
        --nodata
```

These include:

- `--output`saved path of labelme file. If there already exists label file of some images, it will be loaded;
- `--autosave`auto-save label file, and some tedioys steps will be omitted.
- `--nodata`doesn't store the base64 encoding of each image, so setting this flag will greatly reduce the size of the label file.

For example

```
cd /path/to/mmyolo
labelme ./data/cat/images --output ./data/cat/labels --autosave --nodata
```

Type in command and labelme will start, and then check label. If labelme fails to start, type `export QT_DEBUG_PLUGINS=1` in command to see which libraries are missing and install it.

Warning: Make sure to use `rectangle` with the shortcut `Ctrl + R` (see below).

14.2.2 2.2 Only manual annotation

The procedure is the same as 2.1.2 Manual annotation, except that this is a direct labeling, there is no pre-generated label.

14.3 3. Convert the dataset into COCO format

14.3.1 3.1 Using scripts to convert

MMYOLO provides scripts to convert labelme labels to COCO labels

```
python tools/dataset_converters/labelme2coco.py --img-dir ${image dir path} \
                                                --labels-dir ${label dir location} \
                                                --out ${output COCO label json path} \
                                                [--class-id-txt ${class_with_id.txt path}
↪]
```

These include: `--class-id-txt`: is the `.txt` file of `id class_name` dataset:

- If not specified, the script will be generated automatically in the same directory as `--out`, and save it as `class_with_id.txt`;
- If specified, the script will read but not add or overwrite. It will also check if there are any other classes in the `.txt` file and will give you an error if there are any. Please check the `.txt` file and add the new class and its `id`.

An example `.txt` file looks like this (`id` start at 1, just like COCO):

```
1 cat
2 dog
3 bicycle
4 motorcycle
```

For example:

Coonsider the `cat` dataset for this tutorial:

```
python tools/dataset_converters/labelme2coco.py --img-dir ./data/cat/images \
                                                --labels-dir ./data/cat/labels \
                                                --out ./data/cat/annotations/annotations_
↪all.json
```

For the `cat` dataset in this demo (note that we don't need to include the background class), we can see that the generated `class_with_id.txt` has only the 1 class:

```
1 cat
```

14.3.2 3.2 Check the converted COCO label

Using the following command, we can display the COCO label on the image, which will verify that there are no problems with the conversion:

```
python tools/analysis_tools/browse_coco_json.py --img-dir ${image dir path} \
                                                --ann-file ${COCO label json path}
```

For example:

```
python tools/analysis_tools/browse_coco_json.py --img-dir ./data/cat/images \
                                                --ann-file ./data/cat/annotations/
↪annotations_all.json
```

See also:

See Visualizing COCO label for more information on `tools/analysis_tools/browse_coco_json.py`.

14.4 4. Divide dataset into training set, validation set and test set

Usually, custom dataset is a large folder with full of images. We need to divide the dataset into training set, validation set and test set by ourselves. If the amount of data is small, we can not divide the validation set. Here's how the split script works:

```
python tools/misc/coco_split.py --json ${COCO label json path} \
                                --out-dir ${divide label json saved path} \
                                --ratios ${ratio of division} \
                                [--shuffle] \
                                [--seed ${random seed for division}]
```

These include:

- `--ratios`: ratio of division. If only 2 are set, the split is `trainval + test`, and if 3 are set, the split is `train + val + test`. Two formats are supported - integer and decimal:
  - Integer: divide the dataset in proportion after normalization. Example: `--ratio 2 1 1` (the code will convert to `0.5 0.25 0.25`) or `--ratio 3 1`the code will convert to `0.75 0.25`
  - Decimal: divide the dataset in proportion. **If the sum does not add up to 1, the script performs an automatic normalization correction.** Example: `--ratio 0.8 0.1 0.1` or `--ratio 0.8 0.2`
- `--shuffle`: whether to shuffle the dataset before splitting.
- `--seed`: the random seed of dataset division. If not set, this will be generated automatically.

For example:

```
python tools/misc/coco_split.py --json ./data/cat/annotations/annotations_all.json \
                                --out-dir ./data/cat/annotations \
                                --ratios 0.8 0.2 \
                                --shuffle \
                                --seed 10
```

14.5 5. Create a new config file based on the dataset

Make sure the dataset directory looks like this:

```
.
└── $DATA_ROOT
    ├── annotations
    │   ├── trainval.json # only divide into trainval + test according to the above␣
↪commands; If you use 3 groups to divide the ratio, here is train.jsonval.jsontest.json
    │   └── test.json
    ├── images
    │   ├── image1.jpg
    │   ├── image1.png
    │   └── ...
    └── ...
```

Since this is custom dataset, we need to create a new config and add some information we want to change.

About naming the new config:

- This config inherits from `yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py`;
- We will train the class `cat` from the dataset provided with this tutorial (if you are using your own dataset, you can define the class name of your own dataset);
- The GPU tested in this tutorial is 1 x 3080Ti with 12G video memory, and the computer memory is 32G. The maximum batch size for YOLOv5-s training is `batch size = 32` (see the Appendix for detailed machine information);
- Training epoch is `100 epoch`.

To sum up: you can name it `yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` and place it into the dir of `configs/custom_dataset`.

Create a new directory named `custom_dataset` inside configs dir, and add config file with the following content:

```
_base_ = '../yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py'

max_epochs = 100  # maximum epochs for training
data_root = './data/cat/'  # absolute path to the dataset directory
# data_root = '/root/workspace/mmyolo/data/cat/'  # absolute path to the dataset dir␣
↪inside the Docker container

# the path of result save, can be omitted, omitted save file name is located under work_
↪dirs with the same name of config file.
# If a config variable changes only part of its parameters, changing this variable will␣
↪save the new training file elsewhere
work_dir = './work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat'

# load_from can specify a local path or URL, setting the URL will automatically download,
↪ because the above has been downloaded, we set the local path here
# since this tutorial is fine-tuning on the cat dataset, we need to use `load_from` to␣
↪load the pre-trained model from MMYOLO. This allows for faster convergence and accuracy
load_from = './work_dirs/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth'  # noqa
```

(continues on next page)

(continued from previous page)

```
# according to your GPU situation, modify the batch size, and YOLOv5-s defaults to 8␣
↪cards x 16bs
train_batch_size_per_gpu = 32
train_num_workers = 4  # recommend to use train_num_workers = nGPU x 4

save_epoch_intervals = 2  # save weights every interval round

# according to your GPU situation, modify the base_lr, modification ratio is base_lr_
↪default * (your_bs / default_bs)
base_lr = _base_.base_lr / 4

anchors = [  # the anchor has been updated according to the characteristics of dataset.␣
↪The generation of anchor will be explained in the following section.
    [(68, 69), (154, 91), (143, 162)],  # P3/8
    [(242, 160), (189, 287), (391, 207)],  # P4/16
    [(353, 337), (539, 341), (443, 432)]  # P5/32
]

class_name = ('cat', )  # according to the label information of class_with_id.txt, set␣
↪the class_name
num_classes = len(class_name)
metainfo = dict(
    classes=class_name,
    palette=[(220, 20, 60)]  # the color of drawing, free to set
)

train_cfg = dict(
    max_epochs=max_epochs,
    val_begin=20,  # number of epochs to start validation.  Here 20 is set because the␣
↪accuracy of the first 20 epochs is not high and the test is not meaningful, so it is␣
↪skipped
    val_interval=save_epoch_intervals  # the test evaluation is performed  iteratively␣
↪every val_interval round
)

model = dict(
    bbox_head=dict(
        head_module=dict(num_classes=num_classes),
        prior_generator=dict(base_sizes=anchors),

        # loss_cls is dynamically adjusted based on num_classes, but when num_classes =␣
↪1, loss_cls is always 0
        loss_cls=dict(loss_weight=0.5 *
                      (num_classes / 80 * 3 / _base_.num_det_layers))))

train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        _delete_=True,
        type='RepeatDataset',
        # if the dataset is too small, you can use RepeatDataset, which repeats the␣
↪current dataset n times per epoch, where 5 is set.
```

(continues on next page)

(continued from previous page)

```
        times=5,
        dataset=dict(
            type=_base_.dataset_type,
            data_root=data_root,
            metainfo=metainfo,
            ann_file='annotations/trainval.json',
            data_prefix=dict(img='images/'),
            filter_cfg=dict(filter_empty_gt=False, min_size=32),
            pipeline=_base_.train_pipeline)))

val_dataloader = dict(
    dataset=dict(
        metainfo=metainfo,
        data_root=data_root,
        ann_file='annotations/trainval.json',
        data_prefix=dict(img='images/')))

test_dataloader = val_dataloader

val_evaluator = dict(ann_file=data_root + 'annotations/trainval.json')
test_evaluator = val_evaluator

optim_wrapper = dict(optimizer=dict(lr=base_lr))

default_hooks = dict(
    # set how many epochs to save the model, and the maximum number of models to save,
↪`save_best` is also the best model (recommended).
    checkpoint=dict(
        type='CheckpointHook',
        interval=save_epoch_intervals,
        max_keep_ckpts=5,
        save_best='auto'),
    param_scheduler=dict(max_epochs=max_epochs),
    # logger output interval
    logger=dict(type='LoggerHook', interval=10))
```

Note: We put an identical config file in `projects/misc/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py`. You can choose to copy to `configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` to start training directly.

14.6 6. Visual analysis of datasets

The script `tools/analysis_tools/dataset_analysis.py` will helo you get a plot of your dataset. The script can generate four types of analysis graphs:

- A distribution plot showing categories and the number of bbox instances: `show_bbox_num`
- A distribution plot showing categories and the width and height of bbox instances: `show_bbox_wh`
- A distribution plot showing categories and the width/height ratio of bbox instances: `show_bbox_wh_ratio`
- A distribution plot showing categories and the area of bbox instances based on the area rule: `show_bbox_area`

Here's how the script works:

```
python tools/analysis_tools/dataset_analysis.py ${CONFIG} \
                                                [--val-dataset ${TYPE}] \
                                                [--class-name ${CLASS_NAME}] \
                                                [--area-rule ${AREA_RULE}] \
                                                [--func ${FUNC}] \
                                                [--out-dir ${OUT_DIR}]
```

For example:

Consider the config of `cat` dataset in this tutorial:

Check the distribution of the training data:

```
python tools/analysis_tools/dataset_analysis.py configs/custom_dataset/yolov5_s-v61_
↪syncbn_fast_1xb32-100e_cat.py \
                                                --out-dir work_dirs/dataset_analysis_cat/
↪train_dataset
```

Check the distribution of the validation data:

```
python tools/analysis_tools/dataset_analysis.py configs/custom_dataset/yolov5_s-v61_
↪syncbn_fast_1xb32-100e_cat.py \
                                                --out-dir work_dirs/dataset_analysis_cat/
↪val_dataset \
                                                --val-dataset
```

Effect (click on the image to view a larger image):

Note: Due to the cat dataset used in this tutorial is relatively small, we use RepeatDataset in config. The numbers shown are actually repeated five times. If you want a repeat-free analysis, you can change the `times` argument in RepeatDataset from `5` to `1` for now.

From the analysis output, we can conclude that the training set of the `cat` dataset used in this tutorial has the following characteristics:

- The images are all `large object`;
- The number of categories cat is `655`;
- The width and height ratio of bbox is mostly concentrated in `1.0 ~ 1.11`, the minimum ratio is `0.36` and the maximum ratio is `2.9`;
- The width of bbox is about `500 ~ 600` , and the height is about `500 ~ 600`.

See also:

See Visualizing Dataset Analysis for more information on `tools/analysis_tools/dataset_analysis.py`

14.7 7. Optimize Anchor size

> **Warning:** This step only works for anchor-base models such as YOLOv5;
>
> This step can be skipped for Anchor-free models, such as YOLOv6, YOLOX.

The `tools/analysis_tools/optimize_anchors.py` script supports three anchor generation methods from YOLO series: `k-means`, `Differential Evolution` and `v5-k-means`.

In this tutorial, we will use YOLOv5 for training, with an input size of `640 x 640`, and `v5-k-means` to optimize anchor:

```
python tools/analysis_tools/optimize_anchors.py configs/custom_dataset/yolov5_s-v61_
↪syncbn_fast_1xb32-100e_cat.py \
                                                --algorithm v5-k-means \
                                                --input-shape 640 640 \
                                                --prior-match-thr 4.0 \
                                                --out-dir work_dirs/dataset_analysis_cat
```

Note: Because this command uses the k-means clustering algorithm, there is some randomness, which is related to the initialization. Therefore, the Anchor obtained by each execution will be somewhat different, but it is generated based on the dataset passed in, so it will not have any adverse effects.

The calculated anchors are as follows:

Modify the `anchors` variable in config file:

```
anchors = [
    [(68, 69), (154, 91), (143, 162)],  # P3/8
    [(242, 160), (189, 287), (391, 207)],  # P4/16
    [(353, 337), (539, 341), (443, 432)]  # P5/32
]
```

See also:

See Optimize Anchor Sizes for more information on `tools/analysis_tools/optimize_anchors.py`

14.8 8. Visualization the data processing part of config

The script `tools/analysis_tools/browse_dataset.py` allows you to visualize the data processing part of config directly in the window, with the option to save the visualization to a specific directory.

Let's use the config file we just created `configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` to visualize the images. Each image lasts for `3` seconds, and the images are not saved:

```
python tools/analysis_tools/browse_dataset.py configs/custom_dataset/yolov5_s-v61_syncbn_
↪fast_1xb32-100e_cat.py \
                                                --show-interval 3
```

See also:

See Visualizing Datasets for more information on `tools/analysis_tools/browse_dataset.py`

14.9 9. Train

Here are three points to explain:

1. Training visualization
2. YOLOv5 model training
3. Switching YOLO model training

14.9.1 9.1 Training visualization

If you need to use a browser to visualize the training process, MMYOLO currently offers two ways wandb and TensorBoard. Pick one according to your own situation (we'll expand support for more visualization backends in the future).

9.1.1 wandb

Wandb visualization need registered in website, and in the https://wandb.ai/settings for wandb API Keys.

Then install it from the command line:

```
pip install wandb
# After running wandb login, enter the API Keys obtained above, and the login is␣
↪successful.
wandb login
```

Add the `wandb` configuration at the end of config file we just created, `configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py`.

```
visualizer = dict(vis_backends=[dict(type='LocalVisBackend'), dict(type='WandbVisBackend
↪')])
```

9.1.2 TensorBoard

Install Tensorboard environment

```
pip install tensorboard
```

Add the `tensorboard` configuration at the end of config file we just created, `configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py`.

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

After running the training command, Tensorboard files will be generated in the visualization folder `work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat/${TIMESTAMP}/vis_data`. We can use Tensorboard to view the loss, learning rate, and coco/bbox_mAP visualizations from a web link by running the following command:

```
tensorboard --logdir=work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat
```

14.9.2 9.2 Perform training

Let's start the training with the following command (training takes about 2.5 hours) :

```
python tools/train.py configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py
```

If you have enabled wandb, you can log in to your account to view the details of this training in wandb:

The following is `1 x 3080Ti`, `batch size = 32`, training `100 epoch` optimal precision weight `work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_epoch_98.pth` obtained accuracy (see Appendix for detailed machine information):

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.968
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.968
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.886
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.977
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.977
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.977

bbox_mAP_copypaste: 0.968 1.000 1.000 -1.000 -1.000 0.968
Epoch(val) [98][116/116]  coco/bbox_mAP: 0.9680  coco/bbox_mAP_50: 1.0000  coco/bbox_mAP_
↪75: 1.0000  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_mAP_l: 0.9680
```

Tip: In general finetune best practice, it is recommended that backbone be left out of training and that the learning rate lr be scaled accordingly. However, in this tutorial, we found this approach can fall short to some extent. The possible reason is that the cat category is already in the COCO dataset, and the cat dataset used in this tutorial is relatively small

The following table shows the test accuracy of the MMYOLO YOLOv5 pre-trained model `yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth` without finetune on the cat dataset. It can be seen that the mAP of the `cat` category is only `0.866`, which improve to `0.968` after finetune, improved by '10.2%', which proves that the training was very successful:

```
+---------------+-------+--------------+-------+----------------+------+
| category      | AP    | category     | AP    | category       | AP   |
+---------------+-------+--------------+-------+----------------+------+
| person        | nan   | bicycle      | nan   | car            | nan  |
| motorcycle    | nan   | airplane     | nan   | bus            | nan  |
| train         | nan   | truck        | nan   | boat           | nan  |
| traffic light | nan   | fire hydrant | nan   | stop sign      | nan  |
| parking meter | nan   | bench        | nan   | bird           | nan  |
```

(continues on next page)

(continued from previous page)

```
| cat           | 0.866 | dog           | nan | horse          | nan  |
| sheep         | nan   | cow           | nan | elephant       | nan  |
| bear          | nan   | zebra         | nan | giraffe        | nan  |
| backpack      | nan   | umbrella      | nan | handbag        | nan  |
| tie           | nan   | suitcase      | nan | frisbee        | nan  |
| skis          | nan   | snowboard     | nan | sports ball    | nan  |
| kite          | nan   | baseball bat  | nan | baseball glove | nan  |
| skateboard    | nan   | surfboard     | nan | tennis racket  | nan  |
| bottle        | nan   | wine glass    | nan | cup            | nan  |
| fork          | nan   | knife         | nan | spoon          | nan  |
| bowl          | nan   | banana        | nan | apple          | nan  |
| sandwich      | nan   | orange        | nan | broccoli       | nan  |
| carrot        | nan   | hot dog       | nan | pizza          | nan  |
| donut         | nan   | cake          | nan | chair          | nan  |
| couch         | nan   | potted plant  | nan | bed            | nan  |
| dining table  | nan   | toilet        | nan | tv             | nan  |
| laptop        | nan   | mouse         | nan | remote         | nan  |
| keyboard      | nan   | cell phone    | nan | microwave      | nan  |
| oven          | nan   | toaster       | nan | sink           | nan  |
| refrigerator  | nan   | book          | nan | clock          | nan  |
| vase          | nan   | scissors      | nan | teddy bear     | nan  |
| hair drier    | nan   | toothbrush    | nan | None           | None |
+---------------+-------+---------------+-----+----------------+------+
```

See also:

For details on how to get the accuracy of the pre-trained weights, see the appendix2. How to test the accuracy of dataset on pre-trained weights

14.9.3 9.3 Switch other models in MMYOLO

MMYOLO integrates multiple YOLO algorithms, which makes switching between YOLO models very easy. There is no need to reacquaint with a new repo. You can easily switch between YOLO models by simply modifying the config file:

1. Create a new config file
2. Download the pre-trained weights
3. Starting training

Let's take YOLOv6-s as an example.

1. Create a new config file

```
_base_ = '../yolov6/yolov6_s_syncbn_fast_8xb32-400e_coco.py'

max_epochs = 100  # maximum of training epoch
data_root = './data/cat/'  # absolute path to the dataset directory

# the path of result save, can be omitted, omitted save file name is located under work_
↪dirs with the same name of config file.
# If a config variable changes only part of its parameters, changing this variable will␣
↪save the new training file elsewhere
```

(continues on next page)

(continued from previous page)

```
work_dir = './work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat'

# load_from can specify a local path or URL, setting the URL will automatically download,
↪ because the above has been downloaded, we set the local path here
# since this tutorial is fine-tuning on the cat dataset, we need to use `load_from` to␣
↪load the pre-trained model from MMYOLO. This allows for faster convergence and accuracy
load_from = './work_dirs/yolov6_s_syncbn_fast_8xb32-400e_coco_20221102_203035-932e1d91.
↪pth'  # noqa

# according to your GPU situation, modify the batch size, and YOLOv6-s defaults to 8␣
↪cards x 32bs
train_batch_size_per_gpu = 32
train_num_workers = 4  # recommend to use  train_num_workers = nGPU x 4

save_epoch_intervals = 2  # save weights every interval round

# according to your GPU situation, modify the base_lr, modification ratio is base_lr_
↪default * (your_bs / default_bs)
base_lr = _base_.base_lr / 8

class_name = ('cat', )  # according to the label information of class_with_id.txt, set␣
↪the class_name
num_classes = len(class_name)
metainfo = dict(
    classes=class_name,
    palette=[(220, 20, 60)]  # the color of drawing, free to set
)

train_cfg = dict(
    max_epochs=max_epochs,
    val_begin=20,  # number of epochs to start validation.  Here 20 is set because the␣
↪accuracy of the first 20 epochs is not high and the test is not meaningful, so it is␣
↪skipped
    val_interval=save_epoch_intervals,  # the test evaluation is performed  iteratively␣
↪every val_interval round
    dynamic_intervals=[(max_epochs - _base_.num_last_epochs, 1)]
)

model = dict(
    bbox_head=dict(
        head_module=dict(num_classes=num_classes)),
    train_cfg=dict(
        initial_assigner=dict(num_classes=num_classes),
        assigner=dict(num_classes=num_classes))
)

train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        _delete_=True,
        type='RepeatDataset',
```

(continues on next page)

(continued from previous page)

```
        # if the dataset is too small, you can use RepeatDataset, which repeats the␣
↪current dataset n times per epoch, where 5 is set.
        times=5,
        dataset=dict(
            type=_base_.dataset_type,
            data_root=data_root,
            metainfo=metainfo,
            ann_file='annotations/trainval.json',
            data_prefix=dict(img='images/'),
            filter_cfg=dict(filter_empty_gt=False, min_size=32),
            pipeline=_base_.train_pipeline)))

val_dataloader = dict(
    dataset=dict(
        metainfo=metainfo,
        data_root=data_root,
        ann_file='annotations/trainval.json',
        data_prefix=dict(img='images/')))

test_dataloader = val_dataloader

val_evaluator = dict(ann_file=data_root + 'annotations/trainval.json')
test_evaluator = val_evaluator

optim_wrapper = dict(optimizer=dict(lr=base_lr))

default_hooks = dict(
    # set how many epochs to save the model, and the maximum number of models to save,
↪`save_best` is also the best model (recommended).
    checkpoint=dict(
        type='CheckpointHook',
        interval=save_epoch_intervals,
        max_keep_ckpts=5,
        save_best='auto'),
    param_scheduler=dict(max_epochs=max_epochs),
    # logger output interval
    logger=dict(type='LoggerHook', interval=10))

custom_hooks = [
    dict(
        type='EMAHook',
        ema_type='ExpMomentumEMA',
        momentum=0.0001,
        update_buffers=True,
        strict_load=False,
        priority=49),
    dict(
        type='mmdet.PipelineSwitchHook',
        switch_epoch=max_epochs - _base_.num_last_epochs,
        switch_pipeline=_base_.train_pipeline_stage2)
]
```

Note: Similarly, We put an identical config file in `projects/misc/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py`. You can choose to copy to `configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py` to start training directly.

Even though the new config looks like a lot of stuff, it's actually a lot of duplication. You can use a comparison software to see that most of the configuration is identical to 'yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py'. Because the two config files need to inherit from different config files, you still need to add the necessary configuration.

2. Download the pre-trained weights

```
wget https://download.openmmlab.com/mmyolo/v0/yolov6/yolov6_s_syncbn_fast_8xb32-400e_
↪coco/yolov6_s_syncbn_fast_8xb32-400e_coco_20221102_203035-932e1d91.pth -P work_dirs/
```

3. Starting training

```
python tools/train.py configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py
```

In my experiments, the best model is `work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_epoch_96.pth`,which accuracy is as follows:

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.987
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.987
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.895
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.989
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.989
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.989

bbox_mAP_copypaste: 0.987 1.000 1.000 -1.000 -1.000 0.987
Epoch(val) [96][116/116]  coco/bbox_mAP: 0.9870  coco/bbox_mAP_50: 1.0000  coco/bbox_mAP_
↪75: 1.0000  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_mAP_l: 0.9870
```

The above demonstrates how to switch models in MMYOLO, you can quickly compare the accuracy of different models, and the model with high accuracy can be put into production. In my experiment, the best accuracy of YOLOv6 `0.9870` is `1.9 %` higher than the best accuracy of YOLOv5 `0.9680` , so we will use YOLOv6 for explanation.

14.10 10. Inference

Using the best model for inference, the best model path in the following command is `./work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_epoch_96.pth`, please modify the best model path you trained.

```
python demo/image_demo.py ./data/cat/images \
                          ./configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.
↪py \
                          ./work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/bbox_
↪mAP_epoch_96.pth \
```

(continues on next page)

(continued from previous page)

```
                                --out-dir ./data/cat/pred_images
```

Tip: If the inference result is not ideal, here are two cases:

1. Model underfitting:

   First, we need to determine if there is not enough training epochs resulting in underfitting. If there is not enough training, we need to change the `max_epochs` and `work_dir` parameters in the config file, or create a new config file named as above and start the training again.

2. The dataset needs to be optimized: If adding epochs still doesn't work, we can increase the number of datasets and re-examine and refine the annotations of the dataset before retraining.

14.11 11. Deployment

MMYOLO provides two deployment options:

1. MMDeploy framework for deployment
2. Using `projects/easydeploy` to deployment

14.11.1 11.1 MMDeploy framework for deployment

Considering that the wide variety of machine deployments, there are many times when a local machine will work, but not in production. Here, we recommended to use Docker, so that the environment can be deployed once and used for life, saving the time of operation and maintenance to build the environment and deploy production.

In this part, we will introduce the following steps:

1. Building a Docker image
2. Creating a Docker container
3. Transforming TensorRT models
4. Deploying model and performing inference

See also:

If you are not familiar with Docker, you can refer to the MMDeploy [source manual installation].(https://mmdeploy.readthedocs.io/en/latest/01-how-to-build/build_from_source.html) file to compile directly locally. Once installed, you can skip to11.1.3 Transforming TensorRT models

11.1.1 Building a Docker image

```
git clone -b dev-1.x https://github.com/open-mmlab/mmdeploy.git
cd mmdeploy
docker build docker/GPU/ -t mmdeploy:gpu --build-arg USE_SRC_INSIDE=true
```

Where `USE_SRC_INSIDE=true` is to pull the basis after switching the domestic source, the build speed will be faster.

After executing the script, the build will start, which will take a while:

11.1.2 Creating a Docker container

```
export MMYOLO_PATH=/path/to/local/mmyolo # write the path to MMYOLO on your machine to␣
↪an environment variable
docker run --gpus all --name mmyolo-deploy -v ${MMYOLO_PATH}:/root/workspace/mmyolo -it␣
↪mmdeploy:gpu /bin/bash
```

You can see your local MMYOLO environment mounted inside the container

See also:

You can read more about this in the MMDeploy official documentation Using Docker Images

11.1.3 Transforming TensorRT models

The first step is to install MMYOLO and pycuda in a Docker container

```
export MMYOLO_PATH=/root/workspace/mmyolo # path in the image, which doesn't need to␣
↪modify
cd ${MMYOLO_PATH}
export MMYOLO_VERSION=$(python -c "import mmyolo.version as v; print(v.__version__)")  #␣
↪Check the version number of MMYOLO used for training
echo "Using MMYOLO ${MMYOLO_VERSION}"
mim install --no-cache-dir mmyolo==${MMYOLO_VERSION}
pip install --no-cache-dir pycuda==2022.2
```

Performing model transformations

```
cd /root/workspace/mmdeploy
python ./tools/deploy.py \
    ${MMYOLO_PATH}/configs/deploy/detection_tensorrt-fp16_dynamic-192x192-960x960.py \
    ${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py \
    ${MMYOLO_PATH}/work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_
↪epoch_96.pth \
    ${MMYOLO_PATH}/data/cat/images/mmexport1633684751291.jpg \
    --test-img ${MMYOLO_PATH}/data/cat/images/mmexport1633684751291.jpg \
    --work-dir ./work_dir/yolov6_s_syncbn_fast_1xb32-100e_cat_deploy_dynamic_fp16 \
    --device cuda:0 \
    --log-level INFO \
    --show \
    --dump-info
```

Wait for a few minutes, `All process success.` appearance indicates success:

Looking at the exported path, you can see the file structure as shown in the following screenshot:

```
$WORK_DIR
  ├── deploy.json
  ├── detail.json
  ├── end2end.engine
  ├── end2end.onnx
  └── pipeline.json
```

See also:

For a detailed description of transforming models, see How to Transform Models

11.1.4 Deploying model and performing inference

We need to change the `data_root` in `${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py` to the path in the Docker container:

```
data_root = '/root/workspace/mmyolo/data/cat/'  # absolute path of the dataset dir in␣
↪the Docker container.
```

Execute speed and accuracy tests:

```
python tools/test.py \
    ${MMYOLO_PATH}/configs/deploy/detection_tensorrt-fp16_dynamic-192x192-960x960.py \
    ${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py \
    --model ./work_dir/yolov6_s_syncbn_fast_1xb32-100e_cat_deploy_dynamic_fp16/end2end.
↪engine \
    --speed-test \
    --device cuda
```

The speed test is as follows, we can see that the average inference speed is `24.10ms`, which is a speed improvement compared to PyTorch inference, but also reduce lots of video memory usage:

```
Epoch(test) [ 10/116]    eta: 0:00:20  time: 0.1919  data_time: 0.1330  memory: 12
Epoch(test) [ 20/116]    eta: 0:00:15  time: 0.1220  data_time: 0.0939  memory: 12
Epoch(test) [ 30/116]    eta: 0:00:12  time: 0.1168  data_time: 0.0850  memory: 12
Epoch(test) [ 40/116]    eta: 0:00:10  time: 0.1241  data_time: 0.0940  memory: 12
Epoch(test) [ 50/116]    eta: 0:00:08  time: 0.0974  data_time: 0.0696  memory: 12
Epoch(test) [ 60/116]    eta: 0:00:06  time: 0.0865  data_time: 0.0547  memory: 16
Epoch(test) [ 70/116]    eta: 0:00:05  time: 0.1521  data_time: 0.1226  memory: 16
Epoch(test) [ 80/116]    eta: 0:00:04  time: 0.1364  data_time: 0.1056  memory: 12
Epoch(test) [ 90/116]    eta: 0:00:03  time: 0.0923  data_time: 0.0627  memory: 12
Epoch(test) [100/116]    eta: 0:00:01  time: 0.0844  data_time: 0.0583  memory: 12
[tensorrt]-110 times per count: 24.10 ms, 41.50 FPS
Epoch(test) [110/116]    eta: 0:00:00  time: 0.1085  data_time: 0.0832  memory: 12
```

Accuracy test is as follows. This configuration uses FP16 format inference, which has some drop points, but it is faster and uses less video memory:

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.954
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.860
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.965
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.965
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.965

INFO - bbox_mAP_copypaste: 0.954 1.000 0.975 -1.000 -1.000 0.954
INFO - Epoch(test) [116/116]  coco/bbox_mAP: 0.9540  coco/bbox_mAP_50: 1.0000  coco/bbox_
↪mAP_75: 0.9750  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_mAP_l: 0.
↪9540
```

Deployment model and inference demonstration:

Note: You can use the MMDeploy SDK for deployment and use C++ to further improve inference speed.

```
cd ${MMYOLO_PATH}/demo
python deploy_demo.py \
    ${MMYOLO_PATH}/data/cat/images/mmexport1633684900217.jpg \
    ${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py \
    /root/workspace/mmdeploy/work_dir/yolov6_s_syncbn_fast_1xb32-100e_cat_deploy_dynamic_
↪fp16/end2end.engine \
    --deploy-cfg ${MMYOLO_PATH}/configs/deploy/detection_tensorrt-fp16_dynamic-192x192-
↪960x960.py \
    --out-dir ${MMYOLO_PATH}/work_dirs/deploy_predict_out \
    --device cuda:0 \
    --score-thr 0.5
```

Warning: The script `deploy_demo.py` doesn't achieve batch inference, and the pre-processing code needs to be improved. It cannot fully show the inference speed at the moment, only demonstrate the inference results. we will optimize in the future. Expect!

After executing, you can see the inference image results in `--out-dir` :

Note: You can also use other optimizations like increasing batch size, int8 quantization, etc.

11.1.5 Save and load the Docker container

It would be a waste of time to build a docker image every time. At this point you can consider using docker's packaging api for packaging and loading.

```
# save, the result tar package can be placed on mobile hard disk
docker save mmyolo-deploy > mmyolo-deploy.tar

# load image to system
docker load < /path/to/mmyolo-deploy.tar
```

14.11.2 11.2 Using `projects/easydeploy` to deploy

See also:

See deployment documentation for details.

TODO: This part will be improved in the next version...

14.12 Appendix

14.12.1 1. The detailed environment for training the machine in this tutorial is as follows:

```
sys.platform: linux
Python: 3.9.13 | packaged by conda-forge | (main, May 27 2022, 16:58:50) [GCC 10.3.0]
CUDA available: True
numpy_random_seed: 2147483648
GPU 0: NVIDIA GeForce RTX 3080 Ti
CUDA_HOME: /usr/local/cuda
NVCC: Cuda compilation tools, release 11.5, V11.5.119
GCC: gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
PyTorch: 1.10.0
PyTorch compiling details: PyTorch built with:
  - GCC 7.3
  - C++ Version: 201402
  - Intel(R) oneAPI Math Kernel Library Version 2021.4-Product Build 20210904 for␣
↪Intel(R) 64 architecture applications
  - Intel(R) MKL-DNN v2.2.3 (Git Hash 7336ca9f055cf1bfa13efb658fe15dc9b41f0740)
  - OpenMP 201511 (a.k.a. OpenMP 4.5)
  - LAPACK is enabled (usually provided by MKL)
  - NNPACK is enabled
  - CPU capability usage: AVX2
  - CUDA Runtime 11.3
  - NVCC architecture flags: -gencode;arch=compute_37,code=sm_37;-gencode;arch=compute_
↪50,code=sm_50;-gencode;
                             arch=compute_60,code=sm_60;-gencode;arch=compute_61,code=sm_
↪61;-gencode;arch=compute_70,code=sm_70;
                             -gencode;arch=compute_75,code=sm_75;-gencode;arch=compute_
↪80,code=sm_80;-gencode;
                             arch=compute_86,code=sm_86;-gencode;arch=compute_37,
↪code=compute_37
  - CuDNN 8.2
  - Magma 2.5.2
  - Build settings: BLAS_INFO=mkl, BUILD_TYPE=Release, CUDA_VERSION=11.3, CUDNN_
↪VERSION=8.2.0,
                    CXX_COMPILER=/opt/rh/devtoolset-7/root/usr/bin/c++, CXX_FLAGS= -Wno-
↪deprecated -fvisibility-inlines-hidden
                    -DUSE_PTHREADPOOL -fopenmp -DNDEBUG -DUSE_KINETO -DUSE_FBGEMM -DUSE_
↪QNNPACK -DUSE_PYTORCH_QNNPACK -DUSE_XNNPACK
                    -DSYMBOLICATE_MOBILE_DEBUG_HANDLE -DEDGE_PROFILER_USE_KINETO -O2 -
↪fPIC -Wno-narrowing -Wall -Wextra
                    -Werror=return-type -Wno-missing-field-initializers -Wno-type-limits␣
↪-Wno-array-bounds -Wno-unknown-pragmas
                    -Wno-sign-compare -Wno-error=deprecated-declarations -Wno-stringop-
↪overflow -Wno-psabi -Wno-error=pedantic
                    -Wno-error=redundant-decls -Wno-error=old-style-cast -fdiagnostics-
↪color=always -faligned-new
                    -Wno-unused-but-set-variable -Wno-maybe-uninitialized -fno-math-
↪errno -fno-trapping-math -Werror=format
                    -Wno-stringop-overflow, LAPACK_INFO=mkl, PERF_WITH_AVX=1, PERF_WITH_
↪AVX2=1, PERF_WITH_AVX512=1,
```

(continues on next page)

(continued from previous page)

```
                    TORCH_VERSION=1.10.0, USE_CUDA=ON, USE_CUDNN=ON, USE_EXCEPTION_PTR=1,
↪ USE_GFLAGS=OFF, USE_GLOG=OFF, USE_MKL=ON,
                    USE_MKLDNN=ON, USE_MPI=OFF, USE_NCCL=ON, USE_NNPACK=ON, USE_
↪OPENMP=ON,

TorchVision: 0.11.0
OpenCV: 4.6.0
MMEngine: 0.3.1
MMCV: 2.0.0rc3
MMDetection: 3.0.0rc3
MMYOLO: 0.2.0+cf279a5
```

14.12.2 2. How to test the accuracy of our dataset on the pre-trained weights:

Warning: Premise: The class is in the COCO 80 class!

In this part, we will use the `cat` dataset as an example, using:

- config file: `configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py`
- weight `yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth`

1. modify the path in config file

Because `configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py` is inherited from `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py`. Therefore, you can mainly modify the `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py` file.

2. modify label

Note: It is recommended to make a copy of the label directly to prevent damage to original label

Change the `categories` in `trainval.json` to COCO's original:

```
  "categories": [{"supercategory": "person","id": 1,"name": "person"},{"supercategory":
↪"vehicle","id": 2,"name": "bicycle"},{"supercategory": "vehicle","id": 3,"name": "car"}
↪,{"supercategory": "vehicle","id": 4,"name": "motorcycle"},{"supercategory": "vehicle",
↪"id": 5,"name": "airplane"},{"supercategory": "vehicle","id": 6,"name": "bus"},{
↪"supercategory": "vehicle","id": 7,"name": "train"},{"supercategory": "vehicle","id":␣
↪8,"name": "truck"},{"supercategory": "vehicle","id": 9,"name": "boat"},{"supercategory
↪": "outdoor","id": 10,"name": "traffic light"},{"supercategory": "outdoor","id": 11,
↪"name": "fire hydrant"},{"supercategory": "outdoor","id": 13,"name": "stop sign"},{
↪"supercategory": "outdoor","id": 14,"name": "parking meter"},{"supercategory": "outdoor
↪","id": 15,"name": "bench"},{"supercategory": "animal","id": 16,"name": "bird"},{
↪"supercategory": "animal","id": 17,"name": "cat"},{"supercategory": "animal","id": 18,
↪"name": "dog"},{"supercategory": "animal","id": 19,"name": "horse"},{"supercategory":
↪"animal","id": 20,"name": "sheep"},{"supercategory": "animal","id": 21,"name": "cow"},{
↪"supercategory": "animal","id": 22,"name": "elephant"},{"supercategory": "animal","id
↪": 23,"name": "bear"},{"supercategory": "animal","id": 24,"name": "zebra"},{
↪"supercategory": "animal","id": 25,"name": "giraffe"},{"supercategory": "accessory","id
↪": 27,"name": "backpack"},{"supercategory": "accessory","id": 28,"name": "umbrella"},{
↪"supercategory": "accessory","id": 31,"name": "handbag"},{"supercategory": "accessory",
↪"id": 32,"name": "tie"},{"supercategory": "accessory","id": 33,"name": "suitcase"},{
↪"supercategory": "sports","id": 34,"name": "frisbee"},{"supercategory": "sports","id":
↪35,"name": "skis"},{"supercategory": "sports","id": 36,"name": "snowboard"},{
↪"supercategory": "sports","id": 37,"name": "sports ball"},{"supercategory": "sports",
↪"id": 38,"name": "kite"},{"supercategory": "sports","id": 39,"name": "baseball bat"},{
↪"supercategory": "sports","id": 40,"name": "baseball glove"},{"supercategory": "sports
```

(continues on next page)

(continued from previous page)

Also, change the `category_id` in the `annotations` to the `id` corresponding to COCO, for example, `cat` is 17 in this example. Here are some of the results:

```
  "annotations": [
    {
      "iscrowd": 0,
      "category_id": 17, # This "category_id" is changed to the id corresponding to COCO,
↪ for example, cat is 17
      "id": 32,
      "image_id": 32,
      "bbox": [
        822.49072265625,
        958.3897094726562,
        1513.693115234375,
        988.3231811523438
      ],
      "area": 1496017.9949368387,
      "segmentation": [
        [
          822.49072265625,
          958.3897094726562,
          822.49072265625,
          1946.712890625,
          2336.183837890625,
          1946.712890625,
          2336.183837890625,
          958.3897094726562
        ]
      ]
    }
  ]
```

3. executive command

```
python tools\test.py configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
                     work_dirs/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth \
                     --cfg-options test_evaluator.classwise=True
```

After executing it, we can see the test metrics:

```
+---------------+-------+--------------+-----+----------------+------+
| category      | AP    | category     | AP  | category       | AP   |
+---------------+-------+--------------+-----+----------------+------+
| person        | nan   | bicycle      | nan | car            | nan  |
| motorcycle    | nan   | airplane     | nan | bus            | nan  |
| train         | nan   | truck        | nan | boat           | nan  |
| traffic light | nan   | fire hydrant | nan | stop sign      | nan  |
| parking meter | nan   | bench        | nan | bird           | nan  |
| cat           | 0.866 | dog          | nan | horse          | nan  |
| sheep         | nan   | cow          | nan | elephant       | nan  |
```

(continues on next page)

(continued from previous page)

bear	nan	zebra	nan	giraffe	nan
backpack	nan	umbrella	nan	handbag	nan
tie	nan	suitcase	nan	frisbee	nan
skis	nan	snowboard	nan	sports ball	nan
kite	nan	baseball bat	nan	baseball glove	nan
skateboard	nan	surfboard	nan	tennis racket	nan
bottle	nan	wine glass	nan	cup	nan
fork	nan	knife	nan	spoon	nan
bowl	nan	banana	nan	apple	nan
sandwich	nan	orange	nan	broccoli	nan
carrot	nan	hot dog	nan	pizza	nan
donut	nan	cake	nan	chair	nan
couch	nan	potted plant	nan	bed	nan
dining table	nan	toilet	nan	tv	nan
laptop	nan	mouse	nan	remote	nan
keyboard	nan	cell phone	nan	microwave	nan
oven	nan	toaster	nan	sink	nan
refrigerator	nan	book	nan	clock	nan
vase	nan	scissors	nan	teddy bear	nan
hair drier	nan	toothbrush	nan	None	None

CHAPTER

FIFTEEN

VISUALIZATION

This article includes feature map visualization and Grad-Based and Grad-Free CAM visualization

15.1 Feature map visualization

Visualization provides an intuitive explanation of the training and testing process of the deep learning model.

In MMYOLO, you can use the `Visualizer` provided in MMEngine for feature map visualization, which has the following features:

- Support basic drawing interfaces and feature map visualization.
- Support selecting different layers in the model to get the feature map. The display methods include `squeeze_mean`, `select_max`, and `topk`. Users can also customize the layout of the feature map display with `arrangement`.

15.1.1 Feature map generation

You can use `demo/featmap_vis_demo.py` to get a quick view of the visualization results. To better understand all functions, we list all primary parameters and their features here as follows:

- `img`: the image to visualize. Can be either a single image file or a list of image file paths.
- `config`: the configuration file for the algorithm.
- `checkpoint`: the weight file of the corresponding algorithm.
- `--out-file`: the file path to save the obtained feature map on your device.
- `--device`: the hardware used for image inference. For example, `--device cuda:0` means use the first GPU, whereas `--device cpu` means use CPU.
- `--score-thr`: the confidence score threshold. Only bboxes whose confidence scores are higher than this threshold will be displayed.
- `--preview-model`: if there is a need to preview the model. This could make users understand the structure of the feature layer more straightforwardly.
- `--target-layers`: the specific layer to get the visualized feature map result.
  - If there is only one parameter, the feature map of that specific layer will be visualized. For example, `--target-layers backbone` , `--target-layers neck` , `--target-layers backbone.stage4`, etc.

  - If the parameter is a list, all feature maps of the corresponding layers will be visualized. For example, `--target-layers backbone.stage4 neck` means that the stage4 layer of the backbone and the three layers of the neck are output simultaneously, a total of four layers of feature maps.

- `--channel-reduction`: if needs to compress multiple channels into a single channel and then display it overlaid with the picture as the input tensor usually has multiple channels. Three parameters can be used here:
  - `squeeze_mean`: The input channel C will be compressed into one channel using the mean function, and the output dimension becomes (1, H, W).
  - `select_max`: Sum the input channel C in the spatial space, and the dimension becomes (C,). Then select the channel with the largest value.
  - `None`: Indicates that no compression is required. In this case, the `topk` feature maps with the highest activation degree can be selected to display through the `topk` parameter.
- `--topk`: only valid when the `channel_reduction` parameter is `None`. It selects the `topk` channels according to the activation degree and then displays it overlaid with the image. The display layout can be specified using the `--arrangement` parameter, which is an array of two numbers separated by space. For example, `--topk 5 --arrangement 2 3` means the five feature maps with the highest activation degree are displayed in `2 rows and 3 columns`. Similarly, `--topk 7 --arrangement 3 3` means the seven feature maps with the highest activation degree are displayed in `3 rows and 3 columns`.
  - If `topk` is not -1, topk channels will be selected to display in order of the activation degree.
  - If `topk` is -1, channel number C must be either 1 or 3 to indicate that the input data is a picture. Otherwise, an error will prompt the user to compress the channel with `channel_reduction`.
- Considering that the input feature map is usually very small, the function will upsample the feature map by default for easy visualization.

Note: When the image and feature map scales are different, the `draw_featmap` function will automatically perform an upsampling alignment. If your image has an operation such as `Pad` in the preprocessing during the inference, the feature map obtained is processed with `Pad`, which may cause misalignment problems if you directly upsample the image.

15.1.2 Usage examples

Take the pre-trained YOLOv5-s model as an example. Please download the model weight file to the root directory.

```
cd mmyolo
wget https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth
```

(1) Compress the multi-channel feature map into a single channel with `select_max` and display it. By extracting the output of the `backbone` layer for visualization, the feature maps of the three output layers in the `backbone` will be generated:

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.
↪py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth \
                                --target-layers backbone \
                                --channel-reduction select_max
```

The above code has the problem that the image and the feature map need to be aligned. There are two solutions for this:

1. Change the post-process to simple `Resize` in the YOLOv5 configuration, which does not affect visualization.
2. Use the images after the pre-process stage instead of before the pre-process when visualizing.

For simplicity purposes, we take the first solution in this demo. However, the second solution will be made in the future so that everyone can use it without extra modification on the configuration file. More specifically, change the original `test_pipeline` with the version with Resize process only.

The original `test_pipeline` is:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

Change to the following version:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # change the␣
↪LetterResize to mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

The correct result is shown as follows:

(2) Compress the multi-channel feature map into a single channel using the `squeeze_mean` parameter and display it. By extracting the output of the `neck` layer for visualization, the feature maps of the three output layers of `neck` will be generated:

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.
↪py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth \
                                --target-layers neck \
                                --channel-reduction squeeze_mean
```

(3) Compress the multi-channel feature map into a single channel using the `squeeze_mean` parameter and display it. Then, visualize the feature map by extracting the outputs of the `backbone.stage4` and `backbone.stage3` layers, and the feature maps of the two output layers will be generated:

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.
↪py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth \
                                --target-layers backbone.stage4 backbone.stage3 \
                                --channel-reduction squeeze_mean
```

(4) Use the `--topk 3 --arrangement 2 2` parameter to select the top 3 channels with the highest activation degree in the multi-channel feature map and display them in a `2x2` layout. Users can change the layout to what they want through the `arrangement` parameter, and the feature map will be automatically formatted. First, the `top3` feature map in each layer is formatted in a `2x2` shape, and then each layer is formatted in `2x2` as well:

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.
↪py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth \
                                --target-layers backbone.stage3 backbone.stage4 \
                                --channel-reduction None \
                                --topk 3 \
                                --arrangement 2 2
```

(5) When the visualization process finishes, you can choose to display the result or store it locally. You only need to add the parameter `--out-file xxx.jpg`:

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.
↪py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth \
                                --target-layers backbone \
                                --channel-reduction select_max \
                                --out-file featmap_backbone.jpg
```

15.2 Grad-Based and Grad-Free CAM Visualization

Object detection CAM visualization is much more complex and different than classification CAM. This article only briefly explains the usage, and a separate document will be opened to describe the implementation principles and precautions in detail later.

You can call `demo/boxmap_vis_demo.py` to get the AM visualization results at the Box level easily and quickly. Currently, `YOLOv5/YOLOv6/YOLOX/RTMDet` is supported.

Taking YOLOv5 as an example, as with the feature map visualization, you need to modify the `test_pipeline` first, otherwise there will be a problem of misalignment between the feature map and the original image.

The original `test_pipeline` is:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

Change to the following version:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # change the␣
↪LetterResize to mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

(1) Use the `GradCAM` method to visualize the AM of the last output layer of the neck module

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth
```

The corresponding feature AM is as follows:

It can be seen that the `GradCAM` effect can highlight the AM information at the box level.

You can choose to visualize only the top prediction boxes with the highest prediction scores via the `--topk` parameter

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth \
        --topk 2
```

(2) Use the AblationCAM method to visualize the AM of the last output layer of the neck module

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
```

(continues on next page)

(continued from previous page)

```
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth \
        --method ablationcam
```

Since `AblationCAM` is weighted by the contribution of each channel to the score, it is impossible to visualize only the AM information at the box level like `GradCAN`. But you can use `--norm-in-bbox` to only show bbox inside AM

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth \
        --method ablationcam \
        --norm-in-bbox
```

15.3 Perform inference on large images

First install `sahi` with:

```
pip install -U sahi>=0.11.4
```

Perform MMYOLO inference on large images (as satellite imagery) as:

```
wget -P checkpoint https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_m-v61_syncbn_
↪fast_8xb16-300e_coco/yolov5_m-v61_syncbn_fast_8xb16-300e_coco_20220917_204944-516a710f.
↪pth

python demo/large_image_demo.py \
    demo/large_image.jpg \
    configs/yolov5/yolov5_m-v61_syncbn_fast_8xb16-300e_coco.py \
    checkpoint/yolov5_m-v61_syncbn_fast_8xb16-300e_coco_20220917_204944-516a710f.pth \
```

Arrange slicing parameters as:

```
python demo/large_image_demo.py \
    demo/large_image.jpg \
    configs/yolov5/yolov5_m-v61_syncbn_fast_8xb16-300e_coco.py \
    checkpoint/yolov5_m-v61_syncbn_fast_8xb16-300e_coco_20220917_204944-516a710f.pth \
    --patch-size 512
    --patch-overlap-ratio 0.25
```

Export debug visuals while performing inference on large images as:

```
python demo/large_image_demo.py \
    demo/large_image.jpg \
    configs/yolov5/yolov5_m-v61_syncbn_fast_8xb16-300e_coco.py \
    checkpoint/yolov5_m-v61_syncbn_fast_8xb16-300e_coco_20220917_204944-516a710f.pth \
    --debug
```

`sahi` citation:

```
@article{akyon2022sahi,
  title={Slicing Aided Hyper Inference and Fine-tuning for Small Object Detection},
  author={Akyon, Fatih Cagatay and Altinuc, Sinan Onur and Temizel, Alptekin},
  journal={2022 IEEE International Conference on Image Processing (ICIP)},
  doi={10.1109/ICIP46576.2022.9897990},
  pages={966-970},
  year={2022}
}
```

CHAPTER

SIXTEEN

MMDEPLOY DEPLOYMENT TUTORIAL

16.1 Basic Deployment Guide

16.1.1 Introduction of MMDeploy

MMDeploy is an open-source deep learning model deployment toolset. It is a part of the OpenMMLab project, and provides **a unified experience of exporting different models** to various platforms and devices of the OpenMMLab series libraries. Using MMDeploy, developers can easily export the specific compiled SDK they need from the training result, which saves a lot of effort.

More detailed introduction and guides can be found here

16.1.2 Supported Algorithms

Currently our deployment kit supports on the following models and backends:

Note: ncnn and other inference backends support are coming soon.

16.1.3 Installation

Please install mmdeploy by following this guide.

Note: If you install mmdeploy prebuilt package, please also clone its repository by 'git clone https://github.com/open-mmlab/mmdeploy.git –depth=1' to get the 'tools' file for deployment.

16.1.4 How to Write Config for MMYOLO

All config files related to the deployment are located at `configs/deploy`.

You only need to change the relative data processing part in the model config file to support either static or dynamic input for your model. Besides, MMDeploy integrates the post-processing parts as customized ops, you can modify the strategy in `post_processing` parameter in `codebase_config`.

Here is the detail description:

```
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
```

(continues on next page)

(continued from previous page)

```
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
```

- `score_threshold`: set the score threshold to filter candidate bboxes before `nms`
- `confidence_threshold`: set the confidence threshold to filter candidate bboxes before `nms`
- `iou_threshold`: set the `iou` threshold for removing duplicates in `nms`
- `max_output_boxes_per_class`: set the maximum number of bboxes for each class
- `pre_top_k`: set the number of fixedcandidate bboxes before `nms`, sorted by scores
- `keep_top_k`: set the number of output candidate bboxs after `nms`
- `background_label_id`: set to `-1` as MMYOLO has no background class information

Configuration for Static Inputs

1. Model Config

Taking `YOLOv5` of MMYOLO as an example, here are the details:

```
_base_ = '../../yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'

test_pipeline = [
    dict(type='LoadImageFromFile', backend_args=_base_.backend_args),
    dict(
        type='LetterResize',
        scale=_base_.img_scale,
        allow_scale_up=False,
        use_mini_pad=False,
    ),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

test_dataloader = dict(
    dataset=dict(pipeline=test_pipeline, batch_shapes_cfg=None))
```

`_base_ = '../../yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'` inherits the model config in the training stage.

`test_pipeline` adds the data processing piple for the deployment, `LetterResize` controls the size of the input images and the input for the converted model

`test_dataloader` adds the dataloader config for the deployment, `batch_shapes_cfg` decides whether to use the `batch_shapes` strategy. More details can be found at yolov5 configs

2. Deployment Config

Here we still use the `YOLOv5` in MMYOLO as the example. We can use `detection_onnxruntime_static.py` as the config to deploy `YOLOv5` to `ONNXRuntime` with static inputs.

```
_base_ = ['./base_static.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
backend_config = dict(type='onnxruntime')
```

`backend_config` indicates the deployment backend with `type='onnxruntime'`, other information can be referred from the third section.

To deploy the `YOLOv5` to `TensorRT`, please refer to the `detection_tensorrt_static-640x640.py` as follows.

```
_base_ = ['./base_static.py']
onnx_config = dict(input_shape=(640, 640))
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 640, 640],
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 640, 640])))
    ])
use_efficientnms = False
```

`backend_config` indices the backend with `type='tensorrt'`.

Different from `ONNXRuntime` deployment configuration, `TensorRT` needs to specify the input image size and the parameters required to build the engine file, including:

- `onnx_config` specifies the input shape as `input_shape=(640, 640)`
- `fp16_mode=False` and `max_workspace_size=1 << 30` in `backend_config['common_config']` indicates whether to build the engine in the parameter format of `fp16`, and the maximum video memory for the cur-

rent gpu device, respectively. The unit is in GB. For detailed configuration of fp16, please refer to the detection_tensorrt-fp16_static-640x640.py

- The min_shape/opt_shape/max_shape in backend_config['model_inputs']['input_shapes']['input'] should remain the same under static input, the default is [1, 3, 640, 640].

use_efficientnms is a new configuration introduced by the MMYOLO series, indicating whether to enable Efficient NMS Plugin to replace TRTBatchedNMS plugin in MMDeploy when exporting onnx.

You can refer to the official efficient NMS plugins by TensorRT for more details.

Note: this out-of-box feature is **only available in TensorRT>=8.0**, no need to compile it by yourself.

Configuration for Dynamic Inputs

1. Model Config

When you deploy a dynamic input model, you don't need to modify any model configuration files but the deployment configuration files.

2. Deployment Config

To deploy the YOLOv5 in MMYOLO to ONNXRuntime, please refer to the detection_onnxruntime_dynamic.py.

```
_base_ = ['./base_dynamic.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
backend_config = dict(type='onnxruntime')
```

backend_config indicates the backend with type='onnxruntime'. Other parameters stay the same as the static input section.

To deploy the YOLOv5 to TensorRT, please refer to the detection_tensorrt_dynamic-192x192-960x960.py.

```
_base_ = ['./base_dynamic.py']
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 192, 192],
```

(continues on next page)

(continued from previous page)

```
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 960, 960])))
    ])
use_efficientnms = False
```

`backend_config` indicates the backend with `type='tensorrt'`. Since the dynamic and static inputs are different in `TensorRT`, please check the details at TensorRT dynamic input official introduction.

`TensorRT` deployment requires you to specify `min_shape`, `opt_shape` , and `max_shape`. `TensorRT` limits the size of the input image between `min_shape` and `max_shape`.

`min_shape` is the minimum size of the input image. `opt_shape` is the common size of the input image, inference performance is best under this size. `max_shape` is the maximum size of the input image.

`use_efficientnms` configuration is the same as the `TensorRT` static input configuration in the previous section.

INT8 Quantization Support

Note: Int8 quantization support will soon be released.

16.1.5 How to Convert Model

Usage

Deploy with MMDeploy Tools

Set the root directory of `MMDeploy` as an env parameter `MMDEPLOY_DIR` using `export MMDEPLOY_DIR=/the/root/path/of/MMDeploy` command.

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    ${DEPLOY_CFG_PATH} \
    ${MODEL_CFG_PATH} \
    ${MODEL_CHECKPOINT_PATH} \
    ${INPUT_IMG} \
    --test-img ${TEST_IMG} \
    --work-dir ${WORK_DIR} \
    --calib-dataset-cfg ${CALIB_DATA_CFG} \
    --device ${DEVICE} \
    --log-level INFO \
    --show \
    --dump-info
```

Parameter Description

- `deploy_cfg`: set the deployment config path of MMDeploy for the model, including the type of inference framework, whether quantize, whether the input shape is dynamic, etc. There may be a reference relationship between configuration files, e.g. `configs/deploy/detection_onnxruntime_static.py`
- `model_cfg`: set the MMYOLO model config path, e.g. `configs/deploy/model/yolov5_s-deploy.py`, regardless of the path to MMDeploy
- `checkpoint`: set the torch model path. It can start with `http/https`, more details are available in `mmengine.fileio` apis
- `img`: set the path to the image or point cloud file used for testing during model conversion
- `--test-img`: set the image file that used to test model. If not specified, it will be set to `None`
- `--work-dir`: set the work directory that used to save logs and models
- `--calib-dataset-cfg`: use for calibration only for INT8 mode. If not specified, it will be set to None and use "val" dataset in model config for calibration
- `--device`: set the device used for model conversion. The default is `cpu`, for TensorRT used `cuda:0`
- `--log-level`: set log level which in `'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'`. If not specified, it will be set to `INFO`
- `--show`: show the result on screen or not
- `--dump-info`: output SDK information or not

Deploy with MMDeploy API

Suppose the working directory is the root path of mmyolo. Take YoloV5 model as an example. You can download its checkpoint from here, and then convert it to onnx model as follows:

```
from mmdeploy.apis import torch2onnx
from mmdeploy.backend.sdk.export_info import export2SDK

img = 'demo/demo.jpg'
work_dir = 'mmdeploy_models/mmyolo/onnx'
save_file = 'end2end.onnx'
deploy_cfg = 'configs/deploy/detection_onnxruntime_dynamic.py'
model_cfg = 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model_checkpoint = 'checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth'
device = 'cpu'

# 1. convert model to onnx
torch2onnx(img, work_dir, save_file, deploy_cfg, model_cfg,
           model_checkpoint, device)

# 2. extract pipeline info for inference by MMDeploy SDK
export2SDK(deploy_cfg, model_cfg, work_dir, pth=model_checkpoint,
           device=device)
```

16.1.6 Model specification

Before moving on to model inference chapter, let's know more about the converted result structure which is very important for model inference. It is saved in the directory specified with `--wodk_dir`.

The converted results are saved in the working directory `mmdeploy_models/mmyolo/onnx` in the previous example. It includes:

```
mmdeploy_models/mmyolo/onnx
├── deploy.json
├── detail.json
├── end2end.onnx
└── pipeline.json
```

in which,

- **end2end.onnx**: backend model which can be inferred by ONNX Runtime
- ***xxx*.json**: the necessary information for mmdeploy SDK

The whole package **mmdeploy_models/mmyolo/onnx** is defined as **mmdeploy SDK model**, i.e., **mmdeploy SDK model** includes both backend model and inference meta information.

16.1.7 Model inference

Backend model inference

Take the previous converted `end2end.onnx` model as an example, you can use the following code to inference the model and visualize the results.

```python
from mmdeploy.apis.utils import build_task_processor
from mmdeploy.utils import get_input_shape, load_config
import torch

deploy_cfg = 'configs/deploy/detection_onnxruntime_dynamic.py'
model_cfg = 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
device = 'cpu'
backend_model = ['mmdeploy_models/mmyolo/onnx/end2end.onnx']
image = 'demo/demo.jpg'

# read deploy_cfg and model_cfg
deploy_cfg, model_cfg = load_config(deploy_cfg, model_cfg)

# build task and backend model
task_processor = build_task_processor(model_cfg, deploy_cfg, device)
model = task_processor.build_backend_model(backend_model)

# process input image
input_shape = get_input_shape(deploy_cfg)
model_inputs, _ = task_processor.create_input(image, input_shape)

# do model inference
with torch.no_grad():
    result = model.test_step(model_inputs)
```

(continues on next page)

(continued from previous page)

```
# visualize results
task_processor.visualize(
    image=image,
    model=model,
    result=result[0],
    window_name='visualize',
    output_file='work_dir/output_detection.png')
```

With the above code, you can find the inference result `output_detection.png` in `work_dir`.

SDK model inference

You can also perform SDK model inference like following,

```
from mmdeploy_runtime import Detector
import cv2

img = cv2.imread('demo/demo.jpg')

# create a detector
detector = Detector(model_path='mmdeploy_models/mmyolo/onnx',
                    device_name='cpu', device_id=0)
# perform inference
bboxes, labels, masks = detector(img)

# visualize inference result
indices = [i for i in range(len(bboxes))]
for index, bbox, label_id in zip(indices, bboxes, labels):
    [left, top, right, bottom], score = bbox[0:4].astype(int), bbox[4]
    if score < 0.3:
        continue

    cv2.rectangle(img, (left, top), (right, bottom), (0, 255, 0))

cv2.imwrite('work_dir/output_detection.png', img)
```

Besides python API, mmdeploy SDK also provides other FFI (Foreign Function Interface), such as C, C++, C#, Java and so on. You can learn their usage from demos.

16.1.8 How to Evaluate Model

Usage

After the model is converted to your backend, you can use `${MMDEPLOY_DIR}/tools/test.py` to evaluate the performance.

```
python3 ${MMDEPLOY_DIR}/tools/test.py \
    ${DEPLOY_CFG} \
    ${MODEL_CFG} \
    --model ${BACKEND_MODEL_FILES} \
```

(continues on next page)

(continued from previous page)

```
    --device ${DEVICE} \
    --work-dir ${WORK_DIR} \
    [--cfg-options ${CFG_OPTIONS}] \
    [--show] \
    [--show-dir ${OUTPUT_IMAGE_DIR}] \
    [--interval ${INTERVAL}] \
    [--wait-time ${WAIT_TIME}] \
    [--log2file work_dirs/output.txt]
    [--speed-test] \
    [--warmup ${WARM_UP}] \
    [--log-interval ${LOG_INTERVERL}] \
    [--batch-size ${BATCH_SIZE}] \
    [--uri ${URI}]
```

Parameter Description

- `deploy_cfg`: set the deployment config file path.
- `model_cfg`: set the MMYOLO model config file path.
- `--model`: set the converted model. For example, if we exported a TensorRT model, we need to pass in the file path with the suffix ".engine".
- `--device`: indicate the device to run the model. Note that some backends limit the running devices. For example, TensorRT must run on CUDA.
- `--work-dir`: the directory to save the file containing evaluation metrics.
- `--cfg-options`: pass in additional configs, which will override the current deployment configs.
- `--show`: show the evaluation result on screen or not.
- `--show-dir`: save the evaluation result to this directory, valid only when specified.
- `--interval`: set the display interval between each two evaluation results.
- `--wait-time`: set the display time of each window.
- `--log2file`: log evaluation results and speed to file.
- `--speed-test`: test the inference speed or not.
- `--warmup`: warm up before speed test or not, works only when `speed-test` is specified.
- `--log-interval`: the interval between each log, works only when `speed-test` is specified.
- `--batch-size`: set the batch size for inference, which will override the `samples_per_gpu` in data config. The default value is 1, however, not every model supports `batch_size > 1`.
- `--uri`: Remote ipv4:port or ipv6:port for inference on edge device.

Note: other parameters in `${MMDEPLOY_DIR}/tools/test.py` are used for speed test, they will not affect the evaluation results.

16.2 YOLOv5 Deployment

Please check the *basic_deployment_guide* to get familiar with the configurations.

16.2.1 Model Training and Validation

TODO

16.2.2 MMDeploy Environment Setup

Please check the installation document of `MMDeploy` at build_from_source. Please build both `MMDeploy` and the customized Ops to your specific platform.

Note: please check at `MMDeploy` FAQ or create new issues in `MMDeploy` when you come across any problems.

16.2.3 How to Prepare Configuration File

This deployment guide uses the `YOLOv5` model trained on `COCO` dataset in MMYOLO to illustrate the whole process, including both static and dynamic inputs and different procedures for `TensorRT` and `ONNXRuntime`.

For Static Input

1. Model Config

To deploy the model with static inputs, you need to ensure that the model inputs are in fixed size, e.g. the input size is set to `640x640` while uploading data in the test pipeline and test dataloader.

Here is a example in `yolov5_s-static.py`

```
_base_ = '../../yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'

test_pipeline = [
    dict(type='LoadImageFromFile', backend_args=_base_.backend_args),
    dict(
        type='LetterResize',
        scale=_base_.img_scale,
        allow_scale_up=False,
        use_mini_pad=False,
    ),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

test_dataloader = dict(
    dataset=dict(pipeline=test_pipeline, batch_shapes_cfg=None))
```

As the YOLOv5 will turn on allow_scale_up and use_mini_pad during the test to change the size of the input image in order to achieve higher accuracy. However, it will cause the input size mismatch problem when deploying in the static input model.

Compared with the original configuration file, this configuration has been modified as follows:

- turn off the settings related to reshaping the image in test_pipeline, e.g. setting allow_scale_up=False and use_mini_pad=False in LetterResize
- turn off the batch_shapes in test_dataloader as batch_shapes_cfg=None.

2. Deployment Cofnig

To deploy the model to ONNXRuntime, please refer to the detection_onnxruntime_static.py as follows:

```
_base_ = ['./base_static.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
backend_config = dict(type='onnxruntime')
```

The post_processing in the default configuration aligns the accuracy of the current model with the trained pytorch model. If you need to modify the relevant parameters, you can refer to the detailed introduction of *basic_deployment_guide*.

To deploy the model to TensorRT, please refer to the detection_tensorrt_static-640x640.py.

```
_base_ = ['./base_static.py']
onnx_config = dict(input_shape=(640, 640))
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 640, 640],
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 640, 640])))
    ])
use_efficientnms = False
```

In this guide, we use the default settings such as input_shape=(640, 640) and fp16_mode=False to build in network in fp32 mode. Moreover, we set max_workspace_size=1 << 30 for the gpu memory which allows TensorRT to build the engine with maximum 1GB memory.

For Dynamic Input

1. Model Config

As TensorRT limits the minimum and maximum input size, we can use any size for the inputs when deploy the model in dynamic mode. In this way, we can keep the default settings in `yolov5_s-v61_syncbn_8xb16-300e_coco.py`. The data processing and dataloader parts are as follows.

```
batch_shapes_cfg = dict(
    type='BatchShapePolicy',
    batch_size=val_batch_size_per_gpu,
    img_size=img_scale[0],
    size_divisor=32,
    extra_pad_ratio=0.5)

test_pipeline = [
    dict(type='LoadImageFromFile', backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

val_dataloader = dict(
    batch_size=val_batch_size_per_gpu,
    num_workers=val_num_workers,
    persistent_workers=persistent_workers,
    pin_memory=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        test_mode=True,
        data_prefix=dict(img='val2017/'),
        ann_file='annotations/instances_val2017.json',
        pipeline=test_pipeline,
        batch_shapes_cfg=batch_shapes_cfg))
```

We use `allow_scale_up=False` to control when the input small images will be upsampled or not in the initialization of `LetterResize`. At the same time, the default `use_mini_pad=False` turns off the minimum padding strategy of the image, and `val_dataloader['dataset']` is passed in `batch_shapes_cfg=batch_shapes_cfg` to ensure that the minimum padding is performed according to the input size in `batch`. These configs will change the dimensions of the input image, so the converted model can support dynamic inputs according to the above dataset loader when testing.

2. Deployment Cofnig

To deploy the model to `ONNXRuntime`, please refer to the `detection_onnxruntime_dynamic.py` for more details.

```
_base_ = ['./base_dynamic.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
backend_config = dict(type='onnxruntime')
```

Differs from the static input config we introduced in previous section, dynamic input config additionally inherits the `dynamic_axes`. The rest of the configuration stays the same as the static inputs.

To deploy the model to `TensorRT`, please refer to the `detection_tensorrt_dynamic-192x192-960x960.py` for more details.

```
_base_ = ['./base_dynamic.py']
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 192, 192],
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 960, 960])))
    ])
use_efficientnms = False
```

In our example, the network is built in `fp32` mode as `fp16_mode=False`, and the maximum graphic memory is `1GB` for building the `TensorRT` engine as `max_workspace_size=1 << 30`.

At the same time, `min_shape=[1, 3, 192, 192]`, `opt_shape=[1, 3, 640, 640]`, and `max_shape=[1, 3, 960, 960]` in the default setting set the model with minimum input size to `192x192`, the maximum size to `960x960`, and the most common size to `640x640`.

When you deploy the model, it can adopt to the input image dimensions automatically.

16.2.4 How to Convert Model

Note: The `MMDeploy` root directory used in this guide is `/home/openmmlab/dev/mmdeploy`, please modify it to your `MMDeploy` directory.

Use the following command to download the pretrained YOLOv5 weight and save it to your device:

```
wget https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth -O /home/
↪openmmlab/dev/mmdeploy/yolov5s.pth
```

Set the relevant env parameters using the following command as well:

```
export MMDEPLOY_DIR=/home/openmmlab/dev/mmdeploy
export PATH_TO_CHECKPOINTS=/home/openmmlab/dev/mmdeploy/yolov5s.pth
```

YOLOv5 Static Model Deployment

ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_onnxruntime_static.py \
    configs/deploy/model/yolov5_s-static.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cpu
```

TensorRT

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_tensorrt_static-640x640.py \
    configs/deploy/model/yolov5_s-static.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cuda:0
```

YOLOv5 Dynamic Model Deployment

ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_onnxruntime_dynamic.py \
    configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
```

(continues on next page)

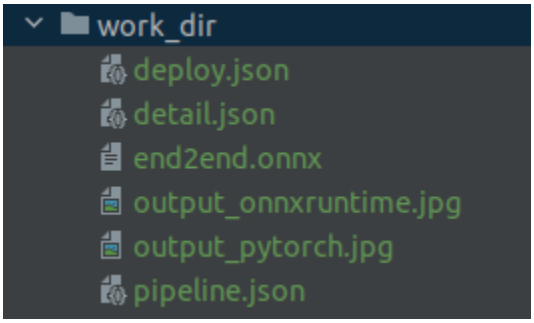
(continued from previous page)

```
    --work-dir work_dir \
    --show \
    --device cpu
    --dump-info
```

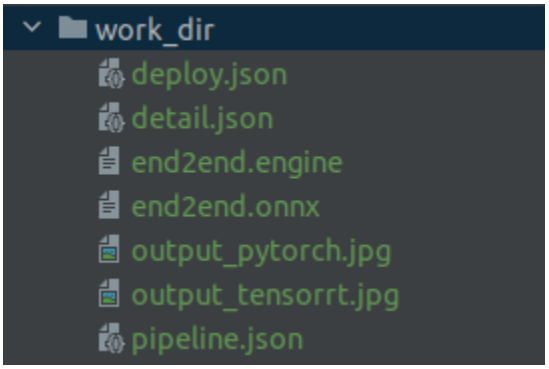
TensorRT

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_tensorrt_dynamic-192x192-960x960.py \
    configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cuda:0
    --dump-info
```

When convert the model using the above commands, you will find the following files under the `work_dir` folder:



or



After exporting to `onnxruntime`, you will get six files as shown in Figure 1, where `end2end.onnx` represents the exported `onnxruntime` model. The `xxx.json` are the meta info for `MMDeploy SDK` inference.

After exporting to `TensorRT`, you will get the seven files as shown in Figure 2, where `end2end.onnx` represents the exported intermediate model. `MMDeploy` uses this model to automatically continue to convert the `end2end.engine` model for `TensorRT Deployment`. The `xxx.json` are the meta info for `MMDeploy SDK` inference.

16.2.5 How to Evaluate Model

After successfully convert the model, you can use `${MMDEPLOY_DIR}/tools/test.py` to evaluate the converted model. The following part shows how to evaluate the static models of `ONNXRuntime` and `TensorRT`. For dynamic model evaluation, please modify the configuration of the inputs.

ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/test.py \
        configs/deploy/detection_onnxruntime_static.py \
        configs/deploy/model/yolov5_s-static.py \
        --model work_dir/end2end.onnx  \
        --device cpu \
        --work-dir work_dir
```

Once the process is done, you can get the output results as this:

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.373
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.566
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.405
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.206
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.419
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.489
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.311
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.511
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.565
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.377
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.620
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.718
11/02 10:22:22 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.206 0.419 0.489
```

TensorRT

Note: `TensorRT` must run on `CUDA` devices!

```
python3 ${MMDEPLOY_DIR}/tools/test.py \
        configs/deploy/detection_tensorrt_static-640x640.py \
        configs/deploy/model/yolov5_s-static.py \
        --model work_dir/end2end.engine  \
        --device cuda:0 \
        --work-dir work_dir
```

Once the process is done, you can get the output results as this:

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.373
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.566
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.405
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.205
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.418
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.489
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.310
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.507
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.556
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.351
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.610
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.714
11/02 10:21:31 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.205 0.418 0.489
```

More useful evaluation tools will be released in the future.

16.3 Deploy using Docker

MMYOLO provides a deployment Dockerfile for deployment purpose. Please make sure your local docker version is greater than 19.03.

Note: users in mainland China can comment out the Optional part in the dockerfile for better experience.

```
# (Optional)
RUN sed -i 's/http:\/\/archive.ubuntu.com\/ubuntu\//http:\/\/mirrors.aliyun.com\/ubuntu\/
↪/g' /etc/apt/sources.list && \
    pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

To build the docker image,

```
# build an image with PyTorch 1.12, CUDA 11.6, TensorRT 8.2.4 ONNXRuntime 1.8.1
docker build -f docker/Dockerfile_deployment -t mmyolo:v1 .
```

To run the docker image,

```
export DATA_DIR=/path/to/your/dataset
docker run --gpus all --shm-size=8g -it --name mmyolo -v ${DATA_DIR}:/openmmlab/mmyolo/
↪data/coco mmyolo:v1
```

DATA_DIR is the path of your COCO dataset.

We provide a script.sh file for you which runs the whole pipeline. Create the script under /openmmlab/mmyolo directory in your docker container using the following content.

```
#!/bin/bash
wget -q https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_8xb16-
↪300e_coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth \
  -O yolov5s.pth
export MMDEPLOY_DIR=/openmmlab/mmdeploy
export PATH_TO_CHECKPOINTS=/openmmlab/mmyolo/yolov5s.pth
```

(continues on next page)

(continued from previous page)

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
  configs/deploy/detection_tensorrt_static-640x640.py \
  configs/deploy/model/yolov5_s-static.py \
  ${PATH_TO_CHECKPOINTS} \
  demo/demo.jpg \
  --work-dir work_dir_trt \
  --device cuda:0

python3 ${MMDEPLOY_DIR}/tools/test.py \
  configs/deploy/detection_tensorrt_static-640x640.py \
  configs/deploy/model/yolov5_s-static.py \
  --model work_dir_trt/end2end.engine \
  --device cuda:0 \
  --work-dir work_dir_trt

python3 ${MMDEPLOY_DIR}/tools/deploy.py \
  configs/deploy/detection_onnxruntime_static.py \
  configs/deploy/model/yolov5_s-static.py \
  ${PATH_TO_CHECKPOINTS} \
  demo/demo.jpg \
  --work-dir work_dir_ort \
  --device cpu

python3 ${MMDEPLOY_DIR}/tools/test.py \
  configs/deploy/detection_onnxruntime_static.py \
  configs/deploy/model/yolov5_s-static.py \
  --model work_dir_ort/end2end.onnx \
  --device cpu \
  --work-dir work_dir_ort
```

Then run the script under `/openmmlab/mmyolo`.

```
sh script.sh
```

This script automatically downloads the `YOLOv5` pretrained weights in `MMYOLO` and convert the model using `MMDeploy`. You will get the output result as follows.

- TensorRT

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.37294
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.56565
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.40474
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.20472
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.41857
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.48867
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.31000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.50722
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.55619
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.35140
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.61063
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.71398
11/03 05:31:50 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.205 0.419 0.489
```

- ONNXRuntime

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.37329
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.56650
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.40494
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.20608
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.41881
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.48873
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.31057
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.51073
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.56509
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.37725
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.62035
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.71803
11/03 05:47:44 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.206 0.419 0.489
```

We can see from the above images that the accuracy of converted models shrink within 1% compared with the pytorch MMYOLO-YOLOv5 models.

If you need to test the inference speed of the converted model, you can use the following commands.

- TensorRT

```
python3 ${MMDEPLOY_DIR}/tools/profiler.py \
  configs/deploy/detection_tensorrt_static-640x640.py \
  configs/deploy/model/yolov5_s-static.py \
  data/coco/val2017 \
  --model work_dir_trt/end2end.engine \
  --device cuda:0
```

- ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/profiler.py \
  configs/deploy/detection_onnxruntime_static.py \
  configs/deploy/model/yolov5_s-static.py \
  data/coco/val2017 \
  --model work_dir_ort/end2end.onnx \
  --device cpu
```

16.3.1 Model Inference

Backend Model Inference

ONNXRuntime

For the converted model `end2end.onnx`you can do the inference with the following code

```
from mmdeploy.apis.utils import build_task_processor
from mmdeploy.utils import get_input_shape, load_config
import torch

deploy_cfg = './configs/deploy/detection_onnxruntime_dynamic.py'
```

(continues on next page)

(continued from previous page)

```
model_cfg = '../mmyolo/configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
device = 'cpu'
backend_model = ['./work_dir/end2end.onnx']
image = '../mmyolo/demo/demo.jpg'

# read deploy_cfg and model_cfg
deploy_cfg, model_cfg = load_config(deploy_cfg, model_cfg)

# build task and backend model
task_processor = build_task_processor(model_cfg, deploy_cfg, device)
model = task_processor.build_backend_model(backend_model)

# process input image
input_shape = get_input_shape(deploy_cfg)
model_inputs, _ = task_processor.create_input(image, input_shape)

# do model inference
with torch.no_grad():
    result = model.test_step(model_inputs)

# visualize results
task_processor.visualize(
    image=image,
    model=model,
    result=result[0],
    window_name='visualize',
    output_file='work_dir/output_detection.png')
```

TensorRT

For the converted model `end2end.engine`you can do the inference with the following code

```
from mmdeploy.apis.utils import build_task_processor
from mmdeploy.utils import get_input_shape, load_config
import torch

deploy_cfg = './configs/deploy/detection_tensorrt_dynamic-192x192-960x960.py'
model_cfg = '../mmyolo/configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
device = 'cuda:0'
backend_model = ['./work_dir/end2end.engine']
image = '../mmyolo/demo/demo.jpg'

# read deploy_cfg and model_cfg
deploy_cfg, model_cfg = load_config(deploy_cfg, model_cfg)

# build task and backend model
task_processor = build_task_processor(model_cfg, deploy_cfg, device)
model = task_processor.build_backend_model(backend_model)

# process input image
input_shape = get_input_shape(deploy_cfg)
```

(continues on next page)

(continued from previous page)

```
model_inputs, _ = task_processor.create_input(image, input_shape)

# do model inference
with torch.no_grad():
    result = model.test_step(model_inputs)

# visualize results
task_processor.visualize(
    image=image,
    model=model,
    result=result[0],
    window_name='visualize',
    output_file='work_dir/output_detection.png')
```

SDK Model Inference

ONNXRuntime

For the converted model `end2end.onnx`you can do the SDK inference with the following code

```
from mmdeploy_runtime import Detector
import cv2

img = cv2.imread('../mmyolo/demo/demo.jpg')

# create a detector
detector = Detector(model_path='work_dir',
                    device_name='cpu', device_id=0)
# perform inference
bboxes, labels, masks = detector(img)

# visualize inference result
indices = [i for i in range(len(bboxes))]
for index, bbox, label_id in zip(indices, bboxes, labels):
    [left, top, right, bottom], score = bbox[0:4].astype(int), bbox[4]
    if score < 0.3:
        continue

    cv2.rectangle(img, (left, top), (right, bottom), (0, 255, 0))

cv2.imwrite('work_dir/output_detection.png', img)
```

TensorRT

For the converted model `end2end.engine`you can do the SDK inference with the following code

```python
from mmdeploy_runtime import Detector
import cv2

img = cv2.imread('../mmyolo/demo/demo.jpg')

# create a detector
detector = Detector(model_path='work_dir',
                    device_name='cuda', device_id=0)
# perform inference
bboxes, labels, masks = detector(img)

# visualize inference result
indices = [i for i in range(len(bboxes))]
for index, bbox, label_id in zip(indices, bboxes, labels):
    [left, top, right, bottom], score = bbox[0:4].astype(int), bbox[4]
    if score < 0.3:
        continue

    cv2.rectangle(img, (left, top), (right, bottom), (0, 255, 0))

cv2.imwrite('work_dir/output_detection.png', img)
```

Besides python API, mmdeploy SDK also provides other FFI (Foreign Function Interface), such as C, C++, C#, Java and so on. You can learn their usage from demos.

CHAPTER

SEVENTEEN

EASYDEPLOY DEPLOYMENT TUTORIAL

17.1 EasyDeploy Deployment

CHAPTER

EIGHTEEN

TROUBLESHOOTING STEPS FOR COMMON ERRORS

CHAPTER

NINETEEN

MM SERIES REPO ESSENTIAL BASICS

CHAPTER

TWENTY

DATASET PREPARATION AND DESCRIPTION

20.1 DOTA Dataset

20.1.1 Download dataset

The DOTA dataset can be downloaded from DOTA or OpenDataLab.

We recommend using OpenDataLab to download the dataset, as the folder structure has already been arranged as needed and can be directly extracted without the need to adjust the folder structure.

Please unzip the file and place it in the following structure.

```
${DATA_ROOT}
├── train
│   ├── images
│   │   ├── P0000.png
│   │   ├── ...
│   ├── labelTxt-v1.0
│   │   ├── labelTxt
│   │   │   ├── P0000.txt
│   │   │   ├── ...
│   │   ├── trainset_reclabelTxt
│   │   │   ├── P0000.txt
│   │   │   ├── ...
├── val
│   ├── images
│   │   ├── P0003.png
│   │   ├── ...
│   ├── labelTxt-v1.0
│   │   ├── labelTxt
│   │   │   ├── P0003.txt
│   │   │   ├── ...
│   │   ├── valset_reclabelTxt
│   │   │   ├── P0003.txt
│   │   │   ├── ...
├── test
│   ├── images
│   │   ├── P0006.png
│   │   ├── ...
```

The folder ending with reclabelTxt stores the labels for the horizontal boxes and is not used when slicing.

20.1.2 Split DOTA dataset

Script `tools/dataset_converters/dota/dota_split.py` can split and prepare DOTA dataset.

```
python tools/dataset_converters/dota/dota_split.py \
    [--splt-config ${SPLIT_CONFIG}] \
    [--data-root ${DATA_ROOT}] \
    [--out-dir ${OUT_DIR}] \
    [--ann-subdir ${ANN_SUBDIR}] \
    [--phase ${DATASET_PHASE}] \
    [--nproc ${NPROC}] \
    [--save-ext ${SAVE_EXT}] \
    [--overwrite]
```

shapely is required, please install shapely first by `pip install shapely`.

Description of all parameters

- `--split-config` : The split config for image slicing.
- `--data-root`: Root dir of DOTA dataset.
- `--out-dir`: Output dir for split result.
- `--ann-subdir`: The subdir name for annotation. Defaults to `labelTxt-v1.0`.
- `--phase`: Phase of the data set to be prepared. Defaults to `trainval test`
- `--nproc`: Number of processes. Defaults to 8.
- `--save-ext`: Extension of the saved image. Defaults to `png`
- `--overwrite`: Whether to allow overwrite if annotation folder exist.

Based on the configuration in the DOTA paper, we provide two commonly used split config.

- `./split_config/single_scale.json` means single-scale split.
- `./split_config/multi_scale.json` means multi-scale split.

DOTA dataset usually uses the trainval set for training and the test set for online evaluation, since most papers provide the results of online evaluation. If you want to evaluate the model performance locally firstly, please split the train set and val set.

Examples:

Split DOTA trainval set and test set with single scale.

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/single_scale.json'
    --data-root ${DATA_ROOT} \
    --out-dir ${OUT_DIR}
```

If you want to split DOTA-v1.5 dataset, which have different annotation dir 'labelTxt-v1.5'.

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/single_scale.json'
    --data-root ${DATA_ROOT} \
    --out-dir ${OUT_DIR} \
    --ann-subdir 'labelTxt-v1.5'
```

If you want to split DOTA train and val set with single scale.

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/single_scale.json'
    --data-root ${DATA_ROOT} \
    --phase train val \
    --out-dir ${OUT_DIR}
```

For multi scale split:

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/multi_scale.json'
    --data-root ${DATA_ROOT} \
    --out-dir ${OUT_DIR}
```

The new data structure is as follows:

```
${OUT_DIR}
├── trainval
│   ├── images
│   │   ├── P0000__1024__0___0.png
│   │   ├── ...
│   ├── annfiles
│   │   ├── P0000__1024__0___0.txt
│   │   ├── ...
├── test
│   ├── images
│   │   ├── P0006__1024__0___0.png
│   │   ├── ...
│   ├── annfiles
│   │   ├── P0006__1024__0___0.txt
│   │   ├── ...
```

Then change `data_root` to ${OUT_DIR}.

CHAPTER

TWENTYONE

RESUME TRAINING

Resume training means to continue training from the state saved from one of the previous trainings, where the state includes the model weights, the state of the optimizer and the optimizer parameter adjustment strategy.

The user can add `--resume` at the end of the training command to resume training, and the program will automatically load the latest weight file from `work_dirs` to resume training. If there is an updated checkpoint in `work_dir` (e.g. the training was interrupted during the last training), the training will be resumed from that checkpoint, otherwise (e.g. the last training did not have time to save the checkpoint or a new training task was started) the training will be restarted. Here is an example of resuming training:

```
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py --resume
```

CHAPTER

TWENTYTWO

AUTOMATIC MIXED PRECISIONAMPTRAINING

To enable Automatic Mixing Precision (AMP) training, add `--amp` to the end of the training command, which is as follows:

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

Specific examples are as follows:

```
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py --amp
```

CHAPTER

TWENTYTHREE

MULTI-SCALE TRAINING AND TESTING

23.1 Multi-scale training

The popular YOLOv5, YOLOv6, YOLOv7, YOLOv8 and RTMDet algorithms are supported in MMYOLO currently, and their default configuration is single-scale 640x640 training. There are two implementations of multi-scale training commonly used in the MM family of open source libraries

1. Each image output in `train_pipeline` is at variable scale, and pad different scales of input images to the same scale by stack_batch function in DataPreprocessor. Most of the algorithms in MMDet are implemented using this approach.
2. Each image output in `train_pipeline` is at a fixed scale, and `DataPreprocessor` performs up- and down-sampling of image batches for multi-scale training directly.

Both two multi-scale training approaches are supported in MMYOLO. Theoretically, the first implementation can generate richer scales, but its training efficiency is not as good as the second one due to its independent augmentation of a single image. Therefore, we recommend using the second approach.

Take `configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py` configuration as an example, its default configuration is 640x640 fixed scale training, suppose you want to implement training in multiples of 32 and multi-scale range (480, 800), you can refer to YOLOX practice by YOLOXBatchSyncRandomResize in the DataPreprocessor.

Create a new configuration under the `configs/yolov5` path named `configs/yolov5/yolov5_s-v61_fast_1xb12-ms-40e_cat.py` with the following contents.

```
_base_ = 'yolov5_s-v61_fast_1xb12-40e_cat.py'

model = dict(
    data_preprocessor=dict(
        type='YOLOv5DetDataPreprocessor',
        pad_size_divisor=32,
        batch_augments=[
            dict(
                type='YOLOXBatchSyncRandomResize',
                # multi-scale range (480, 800)
                random_size_range=(480, 800),
                # The output scale needs to be divisible by 32
                size_divisor=32,
                interval=1)
        ])
)
```

The above configuration will enable multi-scale training. We have already provided this configuration under `configs/yolov5/` for convenience. The rest of the YOLO family of algorithms are similar.

23.2 Multi-scale testing

MMYOLO multi-scale testing is equivalent to Test-Time Enhancement TTA and is currently supported, see *Test-Time Augmentation TTA*.

CHAPTER

TWENTYFOUR

TTA RELATED NOTES

24.1 Test Time Augmentation (TTA)

MMYOLO support for TTA in v0.5.0+, so that users can specify the `-tta` parameter to enable it during evaluation. Take `YOLOv5-s` as an example, its single GPU TTA test command is as follows

```
python tools/test.py configs/yolov5/yolov5_n-v61_syncbn_fast_8xb16-300e_coco.py https://
↪download.openmmlab.com/mmyolo/v0/yolov5/yolov5_n-v61_syncbn_fast_8xb16-300e_coco/
↪yolov5_n-v61_syncbn_fast_8xb16-300e_coco_20220919_090739-b804c1ad.pth  --tta
```

For TTA to work properly, you must ensure that the variables `tta_model` and `tta_pipeline` are present in the configuration, see det_p5_tta.py for details.

The default TTA in MMYOLO performs 3 multi-scale enhancements, followed by 2 horizontal flip enhancements, for a total of 6 parallel pipelines. take `YOLOv5-s` as an example, its TTA configuration is as follows

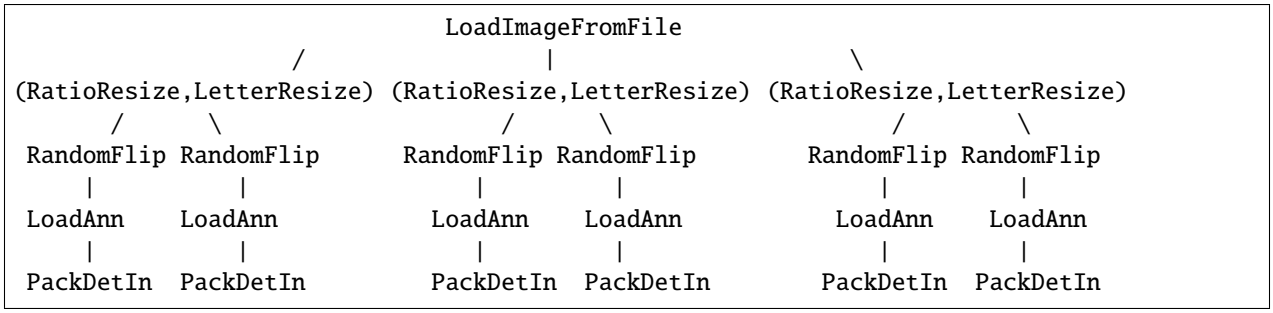
```
img_scales = [(640, 640), (320, 320), (960, 960)]

_multiscale_resize_transforms = [
    dict(
        type='Compose',
        transforms=[
            dict(type='YOLOv5KeepRatioResize', scale=s),
            dict(
                type='LetterResize',
                scale=s,
                allow_scale_up=False,
                pad_val=dict(img=114))
        ]) for s in img_scales
]

tta_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='TestTimeAug',
        transforms=[
            _multiscale_resize_transforms,
            [
                dict(type='mmdet.RandomFlip', prob=1.),
                dict(type='mmdet.RandomFlip', prob=0.)
            ], [dict(type='mmdet.LoadAnnotations', with_bbox=True)],
```

(continues on next page)

(continued from previous page)

```
            [
                dict(
                    type='mmdet.PackDetInputs',
                    meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                               'scale_factor', 'pad_param', 'flip',
                               'flip_direction'))
            ]
        ])
]
```

The schematic diagram is shown below.



You can modify `img_scales` to support different multi-scale enhancements, or you can insert a new pipeline to implement custom TTA requirements. Assuming you only want to do horizontal flip enhancements, the configuration should be modified as follows.

```
tta_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='TestTimeAug',
        transforms=[
            [
                dict(type='mmdet.RandomFlip', prob=1.),
                dict(type='mmdet.RandomFlip', prob=0.)
            ], [dict(type='mmdet.LoadAnnotations', with_bbox=True)],
            [
                dict(
                    type='mmdet.PackDetInputs',
                    meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                               'scale_factor', 'pad_param', 'flip',
                               'flip_direction'))
            ]
        ])
]
```

CHAPTER

TWENTYFIVE

PLUGINS

MMYOLO supports adding plugins such as `none_local` and `dropblock` after different stages of Backbone. Users can directly manage plugins by modifying the plugins parameter of the backbone in the config. For example, add `GeneralizedAttention` plugins for `YOLOv5`. The configuration files are as follows:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    backbone=dict(
        plugins=[
            dict(
                cfg=dict(
                    type='GeneralizedAttention',
                    spatial_range=-1,
                    num_heads=8,
                    attention_type='0011',
                    kv_stride=2),
                stages=(False, False, True, True))
        ]))
```

`cfg` parameter indicates the specific configuration of the plugin. The `stages` parameter indicates whether to add plugins after the corresponding stage of the backbone. The length of the list `stages` must be the same as the number of backbone stages.

MMYOLO currently supports the following plugins:

1. CBAM
2. GeneralizedAttention
3. NonLocal2d
4. ContextBlock

CHAPTER

TWENTYSIX

FREEZE LAYERS

26.1 Freeze the weight of backbone

In MMYOLO, we can freeze some `stages` of the backbone network by setting `frozen_stages` parameters, so that these `stage` parameters do not participate in model updating. It should be noted that `frozen_stages = i` means that all parameters from the initial `stage` to the `i`th `stage` will be frozen. The following is an example of `YOLOv5`. Other algorithms are the same logic.

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    backbone=dict(
        frozen_stages=1 # Indicates that the parameters in the first stage and all␣
↪stages before it are frozen
    ))
```

26.2 Freeze the weight of neck

In addition, it's able to freeze the whole `neck` with the parameter `freeze_all` in MMYOLO. The following is an example of `YOLOv5`. Other algorithms are the same logic.

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    neck=dict(
        freeze_all=True # If freeze_all=True, all parameters of the neck will be frozen
    ))
```

CHAPTER

TWENTYSEVEN

OUTPUT PREDICTION RESULTS

If you want to save the prediction results as a specific file for offline evaluation, MMYOLO currently supports both json and pkl formats.

Note: The json file only save `image_id`, `bbox`, `score` and `category_id`. The json file can be read using the json library. The pkl file holds more content than the json file, and also holds information such as the file name and size of the predicted image; the pkl file can be read using the pickle library. The pkl file can be read using the pickle library.

27.1 Output into json file

If you want to output the prediction results as a json file, the command is as follows.

```
python tools/test.py {path_to_config} {path_to_checkpoint} --json-prefix {json_prefix}
```

The argument after `--json-prefix` should be a filename prefix (no need to enter the `.json` suffix) and can also contain a path. For a concrete example:

```
python tools/test.py configs\yolov5\yolov5_s-v61_syncbn_8xb16-300e_coco.py yolov5_s-v61_
↪syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth --json-prefix work_dirs/demo/
↪json_demo
```

Running the above command will output the `json_demo.bbox.json` file in the `work_dirs/demo` folder.

27.2 Output into pkl file

If you want to output the prediction results as a pkl file, the command is as follows.

```
python tools/test.py {path_to_config} {path_to_checkpoint} --out {path_to_output_file}
```

The argument after `--out` should be a full filename (**must be** with a `.pkl` or `.pickle` suffix) and can also contain a path. For a concrete example:

```
python tools/test.py configs\yolov5\yolov5_s-v61_syncbn_8xb16-300e_coco.py yolov5_s-v61_
↪syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth --out work_dirs/demo/pkl_demo.
↪pkl
```

Running the above command will output the `pkl_demo.pkl` file in the `work_dirs/demo` folder.

CHAPTER

TWENTYEIGHT

SET THE RANDOM SEED

If you want to set the random seed during training, you can use the following command.

```
python ./tools/train.py \
    ${CONFIG} \                                # path of the config file
    --cfg-options randomness.seed=2023 \       # set seed to 2023
    [randomness.diff_rank_seed=True] \         # set different seeds according to global␣
↪rank
    [randomness.deterministic=True]            # set the deterministic option for CUDNN␣
↪backend
# [] stands for optional parameters, when actually entering the command line, you do not␣
↪need to enter []
```

`randomness` has three parameters that can be set, with the following meanings.

- `randomness.seed=2023`, set the random seed to 2023.
- `randomness.diff_rank_seed=True`, set different seeds according to global rank. Defaults to False.
- `randomness.deterministic=True`, set the deterministic option for cuDNN backend, i.e., set `torch.backends.cudnn.deterministic` to True and `torch.backends.cudnn.benchmark` to False. Defaults to False. See https://pytorch.org/docs/stable/notes/randomness.html for more details.

CHAPTER

TWENTYNINE

MODULE COMBINATION

CHAPTER

THIRTY

USE MIM TO RUN SCRIPTS FROM OTHER OPENMMLAB REPOSITORIES

Note:

1. All script calls across libraries are currently not supported and are being fixed. More examples will be added to this document when the fix is complete. 2.
2. mAP plotting and average training speed calculation are fixed in the MMDetection dev-3.x branch, which currently needs to be installed via the source code to be run successfully.

30.1 Log Analysis

30.1.1 Curve plotting

`tools/analysis_tools/analyze_logs.py` plots loss/mAP curves given a training log file. Run `pip install seaborn` first to install the dependency.

```
mim run mmdet analyze_logs plot_curve \
    ${LOG} \                                    # path of train log in json format
    [--keys ${KEYS}] \                          # the metric that you want to plot,␣
↪default to 'bbox_mAP'
    [--start-epoch ${START_EPOCH}]              # the epoch that you want to start,␣
↪default to 1
    [--eval-interval ${EVALUATION_INTERVAL}] \  # the evaluation interval when training,
↪ default to 1
    [--title ${TITLE}] \                        # title of figure
    [--legend ${LEGEND}] \                      # legend of each plot, default to None
    [--backend ${BACKEND}] \                    # backend of plt, default to None
    [--style ${STYLE}] \                        # style of plt, default to 'dark'
    [--out ${OUT_FILE}]                         # the path of output file
# [] stands for optional parameters, when actually entering the command line, you do not␣
↪need to enter []
```

Examples:

- Plot the classification loss of some run.

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls \
    --legend loss_cls
```

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

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls loss_bbox \
    --legend loss_cls loss_bbox \
    --out losses_yolov5_s.pdf
```

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

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    yolov5_n-v61_syncbn_fast_8xb16-300e_coco_20220919_090739.log.json \
    --keys bbox_mAP \
    --legend yolov5_s yolov5_n \
    --eval-interval 10 # Note that the evaluation interval must be the same as␣
↪during training. Otherwise, it will raise an error.
```

30.1.2 Compute the average training speed

```
mim run mmdet analyze_logs cal_train_time \
    ${LOG} \                                # path of train log in json format
    [--include-outliers]                    # include the first value of every epoch␣
↪when computing the average time
```

Examples:

```
mim run mmdet analyze_logs cal_train_time \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json
```

The output is expected to be like the following.

```
-----Analyze train time of yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.
↪json-----
slowest epoch 278, average time is 0.1705 s/iter
fastest epoch 300, average time is 0.1510 s/iter
time std over epochs is 0.0026
average iter time: 0.1556 s/iter
```

CHAPTER

THIRTYONE

APPLY MULTIPLE NECKS

If you want to stack multiple Necks, you can directly set the Neck parameters in the config. MMYOLO supports concatenating multiple Necks in the form of `List`. You need to ensure that the output channel of the previous Neck matches the input channel of the next Neck. If you need to adjust the number of channels, you can insert the `mmdet.ChannelMapper` module to align the number of channels between multiple Necks. The specific configuration is as follows:

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = _base_.widen_factor
model = dict(
    type='YOLODetector',
    neck=[
        dict(
            type='YOLOv5PAFPN',
            deepen_factor=deepen_factor,
            widen_factor=widen_factor,
            in_channels=[256, 512, 1024],
            out_channels=[256, 512, 1024], # The out_channels is controlled by widen_
↪factorso the YOLOv5PAFPN's out_channels equls to out_channels * widen_factor
            num_csp_blocks=3,
            norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
            act_cfg=dict(type='SiLU', inplace=True)),
        dict(
            type='mmdet.ChannelMapper',
            in_channels=[128, 256, 512],
            out_channels=128,
        ),
        dict(
            type='mmdet.DyHead',
            in_channels=128,
            out_channels=256,
            num_blocks=2,
            # disable zero_init_offset to follow official implementation
            zero_init_offset=False)
    ]
    bbox_head=dict(head_module=dict(in_channels=[512,512,512])) # The out_channels is␣
↪controlled by widen_factorso the YOLOv5HeadModuled in_channels * widen_factor equals␣
↪to  the last neck's out_channels
)
```

CHAPTER

THIRTYTWO

SPECIFY SPECIFIC GPUS DURING TRAINING OR INFERENCE

If you have multiple GPUs, such as 8 GPUs, numbered `0, 1, 2, 3, 4, 5, 6, 7`, GPU 0 will be used by default for training or inference. If you want to specify other GPUs for training or inference, you can use the following commands:

```
CUDA_VISIBLE_DEVICES=5 python ./tools/train.py ${CONFIG} #train
CUDA_VISIBLE_DEVICES=5 python ./tools/test.py ${CONFIG} ${CHECKPOINT_FILE} #test
```

If you set `CUDA_VISIBLE_DEVICES` to -1 or a number greater than the maximum GPU number, such as 8, the CPU will be used for training or inference.

If you want to use several of these GPUs to train in parallel, you can use the following command:

```
CUDA_VISIBLE_DEVICES=0,1,2,3 ./tools/dist_train.sh ${CONFIG} ${GPU_NUM}
```

Here the `GPU_NUM` is 4. In addition, if multiple tasks are trained in parallel on one machine and each task requires multiple GPUs, the PORT of each task need to be set differently to avoid communication conflict, like the following commands:

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

CHAPTER

THIRTYTHREE

SINGLE AND MULTI-CHANNEL APPLICATION EXAMPLES

33.1 Training example on a single-channel image dataset

The default training images in MMYOLO are all color three-channel data. If you want to use a single-channel dataset for training and testing, it is expected that the following modifications are needed.

1. All image processing pipelines have to support single channel operations
2. The input channel of the first convolutional layer of the backbone network of the model needs to be changed from 3 to 1
3. If you wish to load COCO pre-training weights, you need to handle the first convolutional layer weight size mismatch

The following uses the `cat` dataset as an example to describe the entire modification process, if you are using a custom grayscale image dataset, you can skip the dataset preprocessing step.

33.1.1 1 Dataset pre-processing

The processing training of the custom dataset can be found in *Annotation-to-deployment workflow for custom dataset*

`cat` is a three-channel color image dataset. For demonstration purpose, you can run the following code and commands to replace the dataset images with single-channel images for subsequent validation.

1. Download the `cat` dataset for decompression

```
python tools/misc/download_dataset.py --dataset-name cat --save-dir ./data/cat --unzip --
↪delete
```

2. Convert datasets to grayscale maps

```
import argparse
import imghdr
import os
from typing import List
import cv2

def parse_args():
    parser = argparse.ArgumentParser(description='data_path')
    parser.add_argument('path', type=str, help='Original dataset path')
    return parser.parse_args()

def main():
```

(continues on next page)

(continued from previous page)

```
    args = parse_args()

    path = args.path + '/images/'
    save_path = path
    file_list: List[str] = os.listdir(path)
    # Grayscale conversion of each imager
    for file in file_list:
        if imghdr.what(path + '/' + file) != 'jpeg':
            continue
        img = cv2.imread(path + '/' + file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        cv2.imwrite(save_path + '/' + file, img)

if __name__ == '__main__':
    main()
```

Name the above script as `cvt_single_channel.py`, and run the command as:

```
python cvt_single_channel.py data/cat
```

33.1.2 2 Modify the base configuration file

At present, some image processing functions of MMYOLO, such as color space transformation, are not compatible with single-channel images, so if we use single-channel data for training directly, we need to modify part of the pipeline, which is a large amount of work. In order to solve the incompatibility problem, the recommended approach is to load the single-channel image as a three-channel image as a three-channel data, but convert it to single-channel format before input to the network. This approach will slightly increase the arithmetic burden, but the user basically does not need to modify the code to use.

Take `projects/misc/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` as the `base` configuration, copy it to the `configs/yolov5` directory, and add `yolov5_s-v61_syncbn_fast_1xb32-100e_cat_single_channel.py` file. We can inherit `YOLOv5DetDataPreprocessor` from the `mmyolo/models/data_preprocessors/data_preprocessor.py` file and name the new class `YOLOv5SCDetDataPreprocessor`, in which convert the image to a single channel, add the dependency library and register the new class in `mmyolo/models/data_preprocessors/__init__.py`. The `YOLOv5SCDetDataPreprocessor` sample code is

```
@MODELS.register_module()
class YOLOv5SCDetDataPreprocessor(YOLOv5DetDataPreprocessor):
    """Rewrite collate_fn to get faster training speed.

    Note: It must be used together with `mmyolo.datasets.utils.yolov5_collate`
    """

    def forward(self, data: dict, training: bool = False) -> dict:
        """Perform normalization, padding, bgr2rgb conversion and convert to single␣
↪channel image based on ``DetDataPreprocessor``.

        Args:
            data (dict): Data sampled from dataloader.
            training (bool): Whether to enable training time augmentation.
```

(continues on next page)

(continued from previous page)

```
        Returns:
            dict: Data in the same format as the model input.
        """
        if not training:
            return super().forward(data, training)

        data = self.cast_data(data)
        inputs, data_samples = data['inputs'], data['data_samples']
        assert isinstance(data['data_samples'], dict)

        # TODO: Supports multi-scale training
        if self._channel_conversion and inputs.shape[1] == 3:
            inputs = inputs[:, [2, 1, 0], ...]

        if self._enable_normalize:
            inputs = (inputs - self.mean) / self.std

        if self.batch_augments is not None:
            for batch_aug in self.batch_augments:
                inputs, data_samples = batch_aug(inputs, data_samples)

        img_metas = [{'batch_input_shape': inputs.shape[2:]}] * len(inputs)
        data_samples = {
            'bboxes_labels': data_samples['bboxes_labels'],
            'img_metas': img_metas
        }

        # Convert to single channel image
        inputs = inputs.mean(dim=1, keepdim=True)

        return {'inputs': inputs, 'data_samples': data_samples}
```

At this point, the `yolov5_s-v61_syncbn_fast_1xb32-100e_cat_single_channel.py` configuration file reads as follows.

```
_base_ = 'yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py'

_base_.model.data_preprocessor.type = 'YOLOv5SCDetDataPreprocessor'
```

33.1.3 3 Pre-training model loading problem

When using a pre-trained 3-channel model directly, it's theoretically possible to experience a decrease in accuracy, though this has not been experimentally verified. To mitigate this potential issue, there are several solutions, including adjusting the weight of each channel in the input layer. One approach is to set the weight of each channel in the input layer to the average of the weights of the original 3 channels. Alternatively, the weight of each channel could be set to one of the weights of the original 3 channels, or the input layer could be trained directly without modifying the weights, depending on the specific circumstances. In this work, we chose to adjust the weights of the 3 channels in the input layer to the average of the weights of the pre-trained 3 channels.

```
import torch

def main():
    # Load weights file
    state_dict = torch.load(
        'checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.
↪pth'
    )

    # Modify input layer weights
    weights = state_dict['state_dict']['backbone.stem.conv.weight']
    avg_weight = weights.mean(dim=1, keepdim=True)
    state_dict['state_dict']['backbone.stem.conv.weight'] = avg_weight

    # Save the modified weights to a new file
    torch.save(
        state_dict,
        'checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187_
↪single_channel.pth'
    )

if __name__ == '__main__':
    main()
```

At this point, the `yolov5_s-v61_syncbn_fast_1xb32-100e_cat_single_channel.py` configuration file reads as follows

```
_base_ = 'yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py'

_base_.model.data_preprocessor.type = 'YOLOv5SCDetDataPreprocessor'

load_from = './checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187_single_channel.pth'
```

33.1.4 4 Model training effect

The left figure shows the actual label and the right figure shows the target detection result.

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.958
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.958
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.881
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.969
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.969
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.969
bbox_mAP_copypaste: 0.958 1.000 1.000 -1.000 -1.000 0.958
Epoch(val) [100][116/116]  coco/bbox_mAP: 0.9580  coco/bbox_mAP_50: 1.0000  coco/bbox_
↪mAP_75: 1.0000  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_mAP_l: 0.
↪9580
```

(continues on next page)

(continued from previous page)

33.2 Training example on a multi-channel image dataset

TODO

CHAPTER

THIRTYFOUR

VISUALIZE COCO LABELS

`tools/analysis_tools/browse_coco_json.py` is a script that can visualization to display the COCO label in the picture.

```
python tools/analysis_tools/browse_coco_json.py [--data-root ${DATA_ROOT}] \
                                                [--img-dir ${IMG_DIR}] \
                                                [--ann-file ${ANN_FILE}] \
                                                [--wait-time ${WAIT_TIME}] \
                                                [--disp-all] [--category-names CATEGORY_
↪NAMES [CATEGORY_NAMES ...]] \
                                                [--shuffle]
```

If images and labels are in the same folder, you can specify `--data-root` to the folder, and then `--img-dir` and `--ann-file` to specify the relative path of the folder. The code will be automatically spliced. If the image and label files are not in the same folder, you do not need to specify `--data-root`, but directly specify `--img-dir` and `--ann-file` of the absolute path.

E.g:

1. Visualize all categories of `COCO` and display all types of annotations such as `bbox` and `mask`:

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --disp-all
```

If images and labels are not in the same folder, you can use a absolutely path:

```
python tools/analysis_tools/browse_coco_json.py --img-dir '/dataset/image/coco/train2017
↪' \
                                                --ann-file '/label/instances_train2017.
↪json' \
                                                --disp-all
```

2. Visualize all categories of `COCO`, and display only the `bbox` type labels, and shuffle the image to show:

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --shuffle
```

3. Only visualize the `bicycle` and `person` categories of `COCO` and only the `bbox` type labels are displayed:

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --category-names 'bicycle' 'person'
```

4. Visualize all categories of COCO, and display all types of label such as bbox, mask, and shuffle the image to show:

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --disp-all \
                                                --shuffle
```

CHAPTER

THIRTYFIVE

VISUALIZE DATASETS

`tools/analysis_tools/browse_dataset.py` helps the user to browse a detection dataset (both images and bounding box annotations) visually, or save the image to a designated directory.

```
python tools/analysis_tools/browse_dataset.py ${CONFIG} \
                                              [--out-dir ${OUT_DIR}] \
                                              [--not-show] \
                                              [--show-interval ${SHOW_INTERVAL}]
```

E,g

1. Use `config` file `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py` to visualize the picture. The picture will pop up directly and be saved to the directory `work_dirs/browse_ dataset` at the same time:

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-
↪300e_coco.py' \
                                              --out-dir 'work_dirs/browse_dataset'
```

2. Use `config` file `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py` to visualize the picture. The picture will pop up and display directly. Each picture lasts for `10` seconds. At the same time, it will be saved to the directory `work_dirs/browse_ dataset`:

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-
↪300e_coco.py' \
                                              --out-dir 'work_dirs/browse_dataset' \
                                              --show-interval 10
```

3. Use `config` file `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py` to visualize the picture. The picture will pop up and display directly. Each picture lasts for `10` seconds and the picture will not be saved:

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-
↪300e_coco.py' \
                                              --show-interval 10
```

4. Use `config` file `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py` to visualize the picture. The picture will not pop up directly, but only saved to the directory `work_dirs/browse_ dataset`:

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-
↪300e_coco.py' \
                                              --out-dir 'work_dirs/browse_dataset' \
                                              --not-show
```

CHAPTER

THIRTYSIX

PRINT THE WHOLE CONFIG

`print_config.py` in MMDetection prints the whole config verbatim, expanding all its imports. The command is as following.

```
mim run mmdet print_config \
    ${CONFIG} \                                 # path of the config file
    [--save-path] \                             # save path of whole config, suffixed with .
↪py, .json or .yml
    [--cfg-options ${OPTIONS [OPTIONS...]}]  # override some settings in the used config
```

Examples:

```
mim run mmdet print_config \
    configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py \
    --save-path ./work_dirs/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py
```

Running the above command will save the `yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py` config file with the inheritance relationship expanded to ``yolov5_s-v61_syncbn_fast_1xb4-300e_balloon_whole.pyin the./work_dirs` folder.

CHAPTER

THIRTYSEVEN

VISUALIZE DATASET ANALYSIS

`tools/analysis_tools/dataset_analysis.py` help users get the renderings of the four functions, and save the pictures to the `dataset_analysis` folder under the current running directory.

Description of the script's functions:

The data required by each sub function is obtained through the data preparation of `main()`.

Function 1: Generated by the sub function `show_bbox_num` to display the distribution of categories and bbox instances.

Function 2: Generated by the sub function `show_bbox_wh` to display the width and height distribution of categories and bbox instances.

Function 3: Generated by the sub function `show_bbox_wh_ratio` to display the width to height ratio distribution of categories and bbox instances.

Function 3: Generated by the sub function `show_bbox_area` to display the distribution map of category and bbox instance area based on area rules.

Print List: Generated by the sub function `show_class_list` and `show_data_list`.

```
python tools/analysis_tools/dataset_analysis.py ${CONFIG} \
                                                [--type ${TYPE}] \
                                                [--class-name ${CLASS_NAME}] \
                                                [--area-rule ${AREA_RULE}] \
                                                [--func ${FUNC}] \
                                                [--out-dir ${OUT_DIR}]
```

E,g

1.Use `config` file `configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py` analyze the dataset, By default,the data loading type is `train_dataset`, the area rule is `[0,32,96,1e5]`, generate a result graph containing all functions and save the graph to the current running directory `./dataset_analysis` folder:

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py
```

2.Use `config` file `configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py` analyze the dataset, change the data loading type from the default `train_dataset` to `val_dataset` through the `--val-dataset` setting:

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --val-dataset
```

3.Use `config` file `configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py` analyze the dataset, change the display of all generated classes to specific classes. Take the display of `person` classes as an example:

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --class-name person
```

4.Use `config` file `configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py` analyze the dataset, redefine the area rule through `--area-rule` . Take `30 70 125` as an example, the area rule becomes `[0,30,70,125,1e5]`

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --area-rule 30 70 125
```

5.Use `config` file `configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py` analyze the dataset, change the display of four function renderings to only display `Function 1` as an example:

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --func show_bbox_num
```

6.Use `config` file `configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py` analyze the dataset, modify the picture saving address to `work_dirs/dataset_analysis`:

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --out-dir work_dirs/dataset_analysis
```

CHAPTER

THIRTYEIGHT

OPTIMIZE ANCHORS SIZE

Script `tools/analysis_tools/optimize_anchors.py` supports three methods to optimize YOLO anchors including `k-means` anchor cluster, `Differential Evolution` and `v5-k-means`.

38.1 k-means

In k-means method, the distance criteria is based IoU, python shell as follow:

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} \
                                                --algorithm k-means \
                                                --input-shape ${INPUT_SHAPE [WIDTH␣
↪HEIGHT]} \
                                                --out-dir ${OUT_DIR}
```

38.2 Differential Evolution

In differential_evolution method, based differential evolution algorithm, use `avg_iou_cost` as minimum target function, python shell as follow:

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} \
                                                --algorithm DE \
                                                --input-shape ${INPUT_SHAPE [WIDTH␣
↪HEIGHT]} \
                                                --out-dir ${OUT_DIR}
```

38.3 v5-k-means

In v5-k-means method, clustering standard as same with YOLOv5 which use shape-match, python shell as follow:

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} \
                                                --algorithm v5-k-means \
                                                --input-shape ${INPUT_SHAPE [WIDTH␣
↪HEIGHT]} \
                                                --prior_match_thr ${PRIOR_MATCH_THR} \
                                                --out-dir ${OUT_DIR}
```

CHAPTER

THIRTYNINE

EXTRACTS A SUBSET OF COCO

The training dataset of the COCO2017 dataset includes 118K images, and the validation set includes 5K images, which is a relatively large dataset. Loading JSON in debugging or quick verification scenarios will consume more resources and bring slower startup speed.

The `extract_subcoco.py` script provides the ability to extract a specified number/classes/area-size of images. The user can use the `--num-img`, `--classes`, `--area-size` parameter to get a COCO subset of the specified condition of images.

For example, extract images use scripts as follows:

```
python tools/misc/extract_subcoco.py \
    ${ROOT} \
    ${OUT_DIR} \
    --num-img 20 \
    --classes cat dog person \
    --area-size small
```

It gone be extract 20 images, and only includes annotations which belongs to cat(or dog/person) and bbox area size is small, after filter by class and area size, the empty annotation images won't be chosen, guarantee the images be extracted definitely has annotation info.

Currently, only support COCO2017. In the future will support user-defined datasets of standard coco JSON format.

The root path folder format is as follows:

```
├── root
│   ├── annotations
│   ├── train2017
│   ├── val2017
│   ├── test2017
```

1. Extract 10 training images and 10 validation images using only 5K validation sets.

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --num-img 10
```

2. Extract 20 training images using the training set and 20 validation images using the validation set.

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --num-img 20 --use-training-set
```

3. Set the global seed to 1. The default is no setting.

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --num-img 20 --use-training-set -
↪-seed 1
```

4. Extract images by specify classes

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --classes cat dog person
```

5. Extract images by specify anchor size

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --area-size small
```

CHAPTER

FORTY

HYPER-PARAMETER SCHEDULER VISUALIZATION

`tools/analysis_tools/vis_scheduler` aims to help the user to check the hyper-parameter scheduler of the optimizer(without training), which support the "learning rate", "momentum", and "weight_decay".

```
python tools/analysis_tools/vis_scheduler.py \
    ${CONFIG_FILE} \
    [-p, --parameter ${PARAMETER_NAME}] \
    [-d, --dataset-size ${DATASET_SIZE}] \
    [-n, --ngpus ${NUM_GPUs}] \
    [-o, --out-dir ${OUT_DIR}] \
    [--title ${TITLE}] \
    [--style ${STYLE}] \
    [--window-size ${WINDOW_SIZE}] \
    [--cfg-options]
```

Description of all arguments

- `config`: The path of a model config file.
- **`-p, --parameter`**: The param to visualize its change curve, choose from "lr", "momentum" or "wd". Default to use "lr".
- **`-d, --dataset-size`**: The size of the datasets. If set`DATASETS.build` will be skipped and `${DATASET_SIZE}` will be used as the size. Default to use the function `DATASETS.build`.
- **`-n, --ngpus`**: The number of GPUs used in training, default to be 1.
- **`-o, --out-dir`**: The output path of the curve plot, default not to output.
- `--title`: Title of figure. If not set, default to be config file name.
- `--style`: Style of plt. If not set, default to be `whitegrid`.
- `--window-size`: The shape of the display window. If not specified, it will be set to `12*7`. If used, it must be in the format `'W*H'`.
- `--cfg-options`: Modifications to the configuration file, refer to *Learn about Configs*.

Note: Loading annotations maybe consume much time, you can directly specify the size of the dataset with `-d, dataset-size` to save time.

You can use the following command to plot the step learning rate schedule used in the config `configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_coco.py`:

```
python tools/analysis_tools/vis_scheduler.py \
    configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_coco.py \
    --dataset-size 118287 \
    --ngpus 8 \
    --out-dir ./output
```

CHAPTER

FORTYONE

DATASET CONVERSION

The folder `tools/data_converters` currently contains `ballon2coco.py`, `yolo2coco.py`, and `labelme2coco.py` - three dataset conversion tools.

- `ballon2coco.py` converts the `balloon` dataset (this small dataset is for starters only) to COCO format.

```
python tools/dataset_converters/balloon2coco.py
```

- `yolo2coco.py` converts a dataset from `yolo-style` **.txt** format to COCO format, please use it as follows:

```
python tools/dataset_converters/yolo2coco.py /path/to/the/root/dir/of/your_dataset
```

Instructions:

1. `image_dir` is the root directory of the yolo-style dataset you need to pass to the script, which should contain `images`, `labels`, and `classes.txt`. `classes.txt` is the class declaration corresponding to the current dataset. One class a line. The structure of the root directory should be formatted as this example shows:

```
.
└── $ROOT_PATH
    ├── classes.txt
    ├── labels
    │   ├── a.txt
    │   ├── b.txt
    │   └── ...
    ├── images
    │   ├── a.jpg
    │   ├── b.png
    │   └── ...
    └── ...
```

2. The script will automatically check if `train.txt`, `val.txt`, and `test.txt` have already existed under `image_dir`. If these files are located, the script will organize the dataset accordingly. Otherwise, the script will convert the dataset into one file. The image paths in these files must be **ABSOLUTE** paths.

3. By default, the script will create a folder called `annotations` in the `image_dir` directory which stores the converted JSON file. If `train.txt`, `val.txt`, and `test.txt` are not found, the output file is `result.json`. Otherwise, the corresponding JSON file will be generated, named as `train.json`, `val.json`, and `test.json`. The `annotations` folder may look similar to this:

```
.
└── $ROOT_PATH
    ├── annotations
    │   ├── result.json
```

(continues on next page)

(continued from previous page)

```
    │   └── ...
    ├── classes.txt
    ├── labels
    │   ├── a.txt
    │   ├── b.txt
    │   └── ...
    ├── images
    │   ├── a.jpg
    │   ├── b.png
    │   └── ...
    └── ...
```

CHAPTER

FORTYTWO

DOWNLOAD DATASET

`tools/misc/download_dataset.py` supports downloading datasets such as `COCO`, `VOC`, `LVIS` and `Balloon`.

```
python tools/misc/download_dataset.py --dataset-name coco2017
python tools/misc/download_dataset.py --dataset-name voc2007
python tools/misc/download_dataset.py --dataset-name voc2012
python tools/misc/download_dataset.py --dataset-name lvis
python tools/misc/download_dataset.py --dataset-name balloon [--save-dir ${SAVE_DIR}] [--
↪unzip]
```

CHAPTER

FORTYTHREE

LOG ANALYSIS

43.1 Curve plotting

`tools/analysis_tools/analyze_logs.py` in MMDetection plots loss/mAP curves given a training log file. Run `pip install seaborn` first to install the dependency.

```
mim run mmdet analyze_logs plot_curve \
    ${LOG} \                                    # path of train log in json format
    [--keys ${KEYS}] \                          # the metric that you want to plot,␣
↪default to 'bbox_mAP'
    [--start-epoch ${START_EPOCH}]              # the epoch that you want to start,␣
↪default to 1
    [--eval-interval ${EVALUATION_INTERVAL}] \  # the evaluation interval when training,
↪ default to 1
    [--title ${TITLE}] \                        # title of figure
    [--legend ${LEGEND}] \                      # legend of each plot, default to None
    [--backend ${BACKEND}] \                    # backend of plt, default to None
    [--style ${STYLE}] \                        # style of plt, default to 'dark'
    [--out ${OUT_FILE}]                         # the path of output file
# [] stands for optional parameters, when actually entering the command line, you do not␣
↪need to enter []
```

Examples:

- Plot the classification loss of some run.

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls \
    --legend loss_cls
```

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

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls loss_bbox \
    --legend loss_cls loss_bbox \
    --out losses_yolov5_s.pdf
```

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

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    yolov5_n-v61_syncbn_fast_8xb16-300e_coco_20220919_090739.log.json \
    --keys bbox_mAP \
    --legend yolov5_s yolov5_n \
    --eval-interval 10 # Note that the evaluation interval must be the same as␣
↪during training. Otherwise, it will raise an error.
```

43.2 Compute the average training speed

```
mim run mmdet analyze_logs cal_train_time \
    ${LOG} \                                # path of train log in json format
    [--include-outliers]                    # include the first value of every epoch␣
↪when computing the average time
```

Examples:

```
mim run mmdet analyze_logs cal_train_time \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json
```

The output is expected to be like the following.

```
-----Analyze train time of yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.
↪json-----
slowest epoch 278, average time is 0.1705 s/iter
fastest epoch 300, average time is 0.1510 s/iter
time std over epochs is 0.0026
average iter time: 0.1556 s/iter
```

CHAPTER

FORTYFOUR

CONVERT MODEL

The six scripts under the `tools/model_converters` directory can help users convert the keys in the official pre-trained model of YOLO to the format of MMYOLO, and use MMYOLO to fine-tune the model.

44.1 YOLOv5

Take conversion `yolov5s.pt` as an example:

1. Clone the official YOLOv5 code to the local (currently the maximum supported version is `v6.1`):

```
git clone -b v6.1 https://github.com/ultralytics/yolov5.git
cd yolov5
```

2. Download official weight file:

```
wget https://github.com/ultralytics/yolov5/releases/download/v6.1/yolov5s.pt
```

3. Copy file `tools/model_converters/yolov5_to_mmyolo.py` to the path of YOLOv5 official code clone:

```
cp ${MMDET_YOLO_PATH}/tools/model_converters/yolov5_to_mmyolo.py yolov5_to_mmyolo.py
```

4. Conversion

```
python yolov5_to_mmyolo.py --src ${WEIGHT_FILE_PATH} --dst mmyolov5.pt
```

The converted `mmyolov5.pt` can be used by MMYOLO. The official weight conversion of YOLOv6 is also used in the same way.

44.2 YOLOX

The conversion of YOLOX model **does not need** to download the official YOLOX code, just download the weight.

Take conversion `yolox_s.pth` as an example:

1. Download official weight file:

```
wget https://github.com/Megvii-BaseDetection/YOLOX/releases/download/0.1.1rc0/yolox_s.pth
```

2. Conversion

```
python tools/model_converters/yolox_to_mmyolo.py --src yolox_s.pth --dst mmyolox.pt
```

The converted `mmyolox.pt` can be used by MMYOLO.

CHAPTER

FORTYFIVE

LEARN ABOUT CONFIGS WITH YOLOV5

MMYOLO and other OpenMMLab repositories use MMEngine's config system. It has a modular and inheritance design, which is convenient to conduct various experiments.

45.1 Config file content

MMYOLO uses a modular design, all modules with different functions can be configured through the config. Taking yolov5_s-v61_syncbn_8xb16-300e_coco.py as an example, we will introduce each field in the config according to different function modules:

45.1.1 Important parameters

When changing the training configuration, it is usually necessary to modify the following parameters. For example, the scaling factors `deepen_factor` and `widen_factor` are used by the network to control the size of the model in MMYOLO. So we recommend defining these parameters separately in the configuration file.

```
img_scale = (640, 640)              # height of image, width of image
deepen_factor = 0.33                # The scaling factor that controls the depth of the␣
↪network structure, 0.33 for YOLOv5-s
widen_factor = 0.5                  # The scaling factor that controls the width of the␣
↪network structure, 0.5 for YOLOv5-s
max_epochs = 300                    # Maximum training epochs: 300 epochs
save_epoch_intervals = 10           # Validation intervals. Run validation every 10 epochs.
train_batch_size_pre_gpu = 16       # Batch size of a single GPU during training
train_num_workers = 8               # Worker to pre-fetch data for each single GPU
val_batch_size_pre_gpu = 1          # Batch size of a single GPU during validation.
val_num_workers = 2                 # Worker to pre-fetch data for each single GPU during␣
↪validation
```

45.1.2 Model config

In MMYOLO's config, we use `model` to set up detection algorithm components. In addition to neural network components such as `backbone`, `neck`, etc, it also requires `data_preprocessor`, `train_cfg`, and `test_cfg`. `data_preprocessor` is responsible for processing a batch of data output by the dataloader. `train_cfg` and `test_cfg` in the model config are for training and testing hyperparameters of the components.

```
anchors = [[(10, 13), (16, 30), (33, 23)], # Basic size of multi-scale prior box
           [(30, 61), (62, 45), (59, 119)],
           [(116, 90), (156, 198), (373, 326)]]
strides = [8, 16, 32] # Strides of multi-scale prior box

model = dict(
    type='YOLODetector', # The name of detector
    data_preprocessor=dict(  # The config of data preprocessor, usually includes image␣
↪normalization and padding
        type='mmdet.DetDataPreprocessor',  # The type of the data preprocessor, refer to␣
↪https://mmdetection.readthedocs.io/en/dev-3.x/api.html#module-mmdet.models.data_
↪preprocessors. It is worth noticing that using `YOLOv5DetDataPreprocessor` achieves␣
↪faster training speed.
        mean=[0., 0., 0.],  # Mean values used to pre-training the pre-trained backbone␣
↪models, ordered in R, G, B
        std=[255., 255., 255.], # Standard variance used to pre-training the pre-trained␣
↪backbone models, ordered in R, G, B
        bgr_to_rgb=True),  # whether to convert image from BGR to RGB
    backbone=dict(  # The config of backbone
        type='YOLOv5CSPDarknet',  # The type of backbone, currently it is available␣
↪candidates are 'YOLOv5CSPDarknet', 'YOLOv6EfficientRep', 'YOLOXCSPDarknet'
        deepen_factor=deepen_factor, # The scaling factor that controls the depth of the␣
↪network structure
        widen_factor=widen_factor, # The scaling factor that controls the width of the␣
↪network structure
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001), # The config of␣
↪normalization layers.
        act_cfg=dict(type='SiLU', inplace=True)), # The config of activation function
    neck=dict(
        type='YOLOv5PAFPN',  # The neck of detector is YOLOv5FPN, We also support
↪'YOLOv6RepPAFPN', 'YOLOXPAFPN'.
        deepen_factor=deepen_factor, # The scaling factor that controls the depth of the␣
↪network structure
        widen_factor=widen_factor, # The scaling factor that controls the width of the␣
↪network structure
        in_channels=[256, 512, 1024], # The input channels, this is consistent with the␣
↪output channels of backbone
        out_channels=[256, 512, 1024], # The output channels of each level of the␣
↪pyramid feature map, this is consistent with the input channels of head
        num_csp_blocks=3, # The number of bottlenecks of CSPLayer
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001), # The config of␣
↪normalization layers.
        act_cfg=dict(type='SiLU', inplace=True)), # The config of activation function
    bbox_head=dict(
        type='YOLOv5Head', # The type of BBox head is 'YOLOv5Head', we also support
↪'YOLOv6Head', 'YOLOXHead'
```

(continues on next page)

(continued from previous page)

```
        head_module=dict(
            type='YOLOv5HeadModule', # The type of Head module is 'YOLOv5HeadModule', we␣
↪also support 'YOLOv6HeadModule', 'YOLOXHeadModule'
            num_classes=80, # Number of classes for classification
            in_channels=[256, 512, 1024], # The input channels, this is consistent with␣
↪the input channels of neck
            widen_factor=widen_factor, # The scaling factor that controls the width of␣
↪the network structure
            featmap_strides=[8, 16, 32], # The strides of the multi-scale feature maps
            num_base_priors=3), # The number of prior boxes on a certain point
        prior_generator=dict( # The config of prior generator
            type='mmdet.YOLOAnchorGenerator', # The prior generator uses
↪'YOLOAnchorGenerator. Refer to https://github.com/open-mmlab/mmdetection/blob/dev-3.x/
↪mmdet/models/task_modules/prior_generators/anchor_generator.py for more details
            base_sizes=anchors, # Basic scale of the anchor
            strides=strides), # The strides of the anchor generator. This is consistent␣
↪with the FPN feature strides. The strides will be taken as base_sizes if base_sizes is␣
↪not set.
    ),
    test_cfg=dict(
        multi_label=True, # The config of multi-label for multi-clas prediction. The␣
↪default setting is True.
        nms_pre=30000,  # The number of boxes before NMS
        score_thr=0.001, # Threshold to filter out boxes.
        nms=dict(type='nms', # Type of NMS
                 iou_threshold=0.65), # NMS threshold
        max_per_img=300)) # Max number of detections of each image
```

45.1.3 Dataset and evaluator config

Dataloaders are required for the training, validation, and testing of the runner. Dataset and data pipeline need to be set to build the dataloader. Due to the complexity of this part, we use intermediate variables to simplify the writing of dataloader configs. More complex data augmentation methods are adopted for the lightweight object detection algorithms in MMYOLO. Therefore, MMYOLO has a wider range of dataset configurations than other models in MMDetection.

The training and testing data flow of YOLOv5 have a certain difference. We will introduce them separately here.

```
dataset_type = 'CocoDataset'  # Dataset type, this will be used to define the dataset
data_root = 'data/coco/'  # Root path of data

pre_transform = [ # Training data loading pipeline
    dict(
        type='LoadImageFromFile'), # First pipeline to load images from file path
    dict(type='LoadAnnotations', # Second pipeline to load annotations for current image
         with_bbox=True) # Whether to use bounding box, True for detection
]

albu_train_transforms = [                    # Albumentation is introduced for image␣
↪data augmentation. We follow the code of YOLOv5-v6.1, please make sure its version is␣
↪1.0.+
```

(continues on next page)

(continued from previous page)

```
    dict(type='Blur', p=0.01),       # Blur augmentation, the probability is 0.01
    dict(type='MedianBlur', p=0.01), # Median blue augmentation, the probability is 0.01
    dict(type='ToGray', p=0.01),         # Randomly convert RGB to gray-scale image, the␣
↪probability is 0.01
    dict(type='CLAHE', p=0.01)                 # CLAHE(Limited Contrast Adaptive␣
↪Histogram Equalization) augmentation, the probability is 0.01
]
train_pipeline = [                                  # Training data processing pipeline
    *pre_transform,                                  # Introduce the pre-defined training␣
↪data loading processing
    dict(
        type='Mosaic',          # Mosaic augmentation
        img_scale=img_scale,    # The image scale after Mosaic augmentation
        pad_val=114.0,          # Pixel values filled with empty areas
        pre_transform=pre_transform), # Pre-defined training data loading pipeline
    dict(
        type='YOLOv5RandomAffine',            # Random Affine augmentation for YOLOv5
        max_rotate_degree=0.0,          # Maximum degrees of rotation transform
        max_shear_degree=0.0,           # Maximum degrees of shear transform
        scaling_ratio_range=(0.5, 1.5), # Minimum and maximum ratio of scaling transform
        border=(-img_scale[0] // 2, -img_scale[1] // 2), # Distance from height and␣
↪width sides of input image to adjust output shape. Only used in mosaic dataset.
        border_val=(114, 114, 114)), # Border padding values of 3 channels.
    dict(
        type='mmdet.Albu',                       # Albumentation of MMDetection
        transforms=albu_train_transforms, # Pre-defined albu_train_transforms
        bbox_params=dict(
            type='BboxParams',
            format='pascal_voc',
            label_fields=['gt_bboxes_labels', 'gt_ignore_flags']),
        keymap={
            'img': 'image',
            'gt_bboxes': 'bboxes'
        }),
    dict(type='YOLOv5HSVRandomAug'),            # Random augmentation on HSV channel
    dict(type='mmdet.RandomFlip', prob=0.5),        # Random flip, the probability is 0.5
    dict(
        type='mmdet.PackDetInputs',                                  # Pipeline that␣
↪formats the annotation data and decides which keys in the data should be packed into␣
↪data_samples
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'flip',
                   'flip_direction'))
]
train_dataloader = dict( # Train dataloader config
    batch_size=train_batch_size_pre_gpu, # Batch size of a single GPU during training
    num_workers=train_num_workers, # Worker to pre-fetch data for each single GPU during␣
↪training
    persistent_workers=True, # If ``True``, the dataloader will not shut down the worker␣
↪processes after an epoch end, which can accelerate training speed.
    pin_memory=True, # If ``True``, the dataloader will allow pinned memory, which can␣
↪reduce copy time between CPU and memory
    sampler=dict( # training data sampler
```

(continues on next page)

(continued from previous page)

```
        type='DefaultSampler', # DefaultSampler which supports both distributed and non-
↪distributed training. Refer to https://github.com/open-mmlab/mmengine/blob/main/
↪mmengine/dataset/sampler.py
        shuffle=True), # randomly shuffle the training data in each epoch
    dataset=dict( # Train dataset config
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_train2017.json', # Path of annotation file
        data_prefix=dict(img='train2017/'), # Prefix of image path
        filter_cfg=dict(filter_empty_gt=False, min_size=32), # Config of filtering␣
↪images and annotations
        pipeline=train_pipeline))
```

In the testing phase of YOLOv5, the Letter Resize method resizes all the test images to the same scale, which preserves the aspect ratio of all testing images. Therefore, the validation and testing phases share the same data pipeline.

```
test_pipeline = [ # Validation/ Testing dataloader config
    dict(
        type='LoadImageFromFile'), # First pipeline to load images from file path
    dict(type='YOLOv5KeepRatioResize', # Second pipeline to resize images with the same␣
↪aspect ratio
         scale=img_scale), # Pipeline that resizes the images
    dict(
        type='LetterResize', # Third pipeline to rescale images to meet the requirements␣
↪of different strides
        scale=img_scale, # Target scale of image
        allow_scale_up=False, # Allow scale up when radio > 1
        pad_val=dict(img=114)), # Padding value
    dict(type='LoadAnnotations', with_bbox=True), # Forth pipeline to load annotations␣
↪for current image
    dict(
        type='mmdet.PackDetInputs', # Pipeline that formats the annotation data and␣
↪decides which keys in the data should be packed into data_samples
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

val_dataloader = dict(
    batch_size=val_batch_size_pre_gpu, # Batch size of a single GPU
    num_workers=val_num_workers, # Worker to pre-fetch data for each single GPU
    persistent_workers=True, # If ``True``, the dataloader will not shut down the worker␣
↪processes after an epoch end, which can accelerate training speed.
    pin_memory=True, # If ``True``, the dataloader will allow pinned memory, which can␣
↪reduce copy time between CPU and memory
    drop_last=False, # IF ``True``, the dataloader will drop data, which fails to make a␣
↪batch
    sampler=dict(
        type='DefaultSampler', # Default sampler for both distributed and normal training
        shuffle=False), # not shuffle during validation and testing
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
```

(continues on next page)

(continued from previous page)

```
        test_mode=True, # # Turn on test mode of the dataset to avoid filtering␣
↪annotations or images
        data_prefix=dict(img='val2017/'), # Prefix of image path
        ann_file='annotations/instances_val2017.json', # Path of annotation file
        pipeline=test_pipeline,
        batch_shapes_cfg=dict(  # Config of batch shapes
            type='BatchShapePolicy', # Policy that makes paddings with least pixels␣
↪during batch inference process, which does not require the image scales of all batches␣
↪to be the same throughout validation.
            batch_size=val_batch_size_pre_gpu, # Batch size for batch shapes strategy,␣
↪equals to validation batch size on single GPU
            img_size=img_scale[0], # Image scale
            size_divisor=32, # The image scale of padding should be divided by pad_size_
↪divisor
            extra_pad_ratio=0.5))) # additional paddings for pixel scale

test_dataloader = val_dataloader
```

Evaluators are used to compute the metrics of the trained model on the validation and testing datasets. The config of evaluators consists of one or a list of metric configs:

```
val_evaluator = dict(  # Validation evaluator config
    type='mmdet.CocoMetric',  # The coco metric used to evaluate AR, AP, and mAP for␣
↪detection
    proposal_nums=(100, 1, 10),        # The number of proposal used to evaluate for␣
↪detection
    ann_file=data_root + 'annotations/instances_val2017.json',  # Annotation file path
    metric='bbox',  # Metrics to be evaluated, `bbox` for detection
)
test_evaluator = val_evaluator  # Testing evaluator config
```

Since the test dataset has no annotation files, the test_dataloader and test_evaluator config in MMYOLO are generally the same as the val's. If you want to save the detection results on the test dataset, you can write the config like this:

```
# inference on test dataset and
# format the output results for submission.
test_dataloader = dict(
    batch_size=1,
    num_workers=2,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file=data_root + 'annotations/image_info_test-dev2017.json',
        data_prefix=dict(img='test2017/'),
        test_mode=True,
        pipeline=test_pipeline))
test_evaluator = dict(
    type='mmdet.CocoMetric',
    ann_file=data_root + 'annotations/image_info_test-dev2017.json',
```

(continues on next page)

(continued from previous page)

```
    metric='bbox',
    format_only=True,  # Only format and save the results to coco json file
    outfile_prefix='./work_dirs/coco_detection/test')  # The prefix of output json files
```

45.1.4 Training and testing config

MMEngine's runner uses Loop to control the training, validation, and testing processes. Users can set the maximum training epochs and validation intervals with these fields.

```
max_epochs = 300 # Maximum training epochs: 300 epochs
save_epoch_intervals = 10 # Validation intervals. Run validation every 10 epochs.

train_cfg = dict(
    type='EpochBasedTrainLoop',  # The training loop type. Refer to https://github.com/
↪open-mmlab/mmengine/blob/main/mmengine/runner/loops.py
    max_epochs=max_epochs,  # Maximum training epochs: 300 epochs
    val_interval=save_epoch_intervals)  # Validation intervals. Run validation every 10␣
↪epochs.
val_cfg = dict(type='ValLoop')  # The validation loop type
test_cfg = dict(type='TestLoop')  # The testing loop type
```

MMEngine also supports dynamic intervals for evaluation. For example, you can run validation every 10 epochs on the first 280 epochs, and run validation every epoch on the final 20 epochs. The configurations are as follows.

```
max_epochs = 300 # Maximum training epochs: 300 epochs
save_epoch_intervals = 10 # Validation intervals. Run validation every 10 epochs.

train_cfg = dict(
    type='EpochBasedTrainLoop',  # The training loop type. Refer to https://github.com/
↪open-mmlab/mmengine/blob/main/mmengine/runner/loops.py
    max_epochs=max_epochs,  # Maximum training epochs: 300 epochs
    val_interval=save_epoch_intervals,  # Validation intervals. Run validation every 10␣
↪epochs.
    dynamic_intervals=[(280, 1)]) # Switch evaluation on 280 epoch and switch the␣
↪interval to 1.
val_cfg = dict(type='ValLoop')  # The validation loop type
test_cfg = dict(type='TestLoop')  # The testing loop type
```

45.1.5 Optimization config

`optim_wrapper` is the field to configure optimization-related settings. The optimizer wrapper not only provides the functions of the optimizer but also supports functions such as gradient clipping, mixed precision training, etc. Find out more in the optimizer wrapper tutorial.

```
optim_wrapper = dict(  # Optimizer wrapper config
    type='OptimWrapper',  # Optimizer wrapper type, switch to AmpOptimWrapper to enable␣
↪mixed precision training.
    optimizer=dict(  # Optimizer config. Support all kinds of optimizers in PyTorch.␣
↪Refer to https://pytorch.org/docs/stable/optim.html#algorithms
        type='SGD',  # Stochastic gradient descent optimizer
```

(continues on next page)

(continued from previous page)

```
        lr=0.01,  # The base learning rate
        momentum=0.937, # Stochastic gradient descent with momentum
        weight_decay=0.0005, # Weight decay of SGD
        nesterov=True, # Enable Nesterov momentum, Refer to http://www.cs.toronto.edu/~
↪hinton/absps/momentum.pdf
        batch_size_pre_gpu=train_batch_size_pre_gpu),  # Enable automatic learning rate␣
↪scaling
    clip_grad=None,  # Gradient clip option. Set None to disable gradient clip. Find␣
↪usage in https://mmengine.readthedocs.io/en/latest/tutorials/optim_wrapper.html
    constructor='YOLOv5OptimizerConstructor') # The constructor for YOLOv5 optimizer
```

`param_scheduler` is the field that configures methods of adjusting optimization hyperparameters such as learning rate and momentum. Users can combine multiple schedulers to create a desired parameter adjustment strategy. Find more in the parameter scheduler tutorial. In YOLOv5, parameter scheduling is complex to implement and difficult to implement with `param_scheduler`. So we use `YOLOv5ParamSchedulerHook` to implement it (see next section), which is simpler but less versatile.

```
param_scheduler = None
```

45.1.6 Hook config

Users can attach hooks to training, validation, and testing loops to insert some operations during running. There are two different hook fields, one is `default_hooks` and the other is `custom_hooks`.

`default_hooks` is a dict of hook configs for the hooks that must be required at the runtime. They have default priority which should not be modified. If not set, the runner will use the default values. To disable a default hook, users can set its config to `None`.

```
default_hooks = dict(
    param_scheduler=dict(
        type='YOLOv5ParamSchedulerHook', # MMYOLO uses `YOLOv5ParamSchedulerHook` to␣
↪adjust hyper-parameters in optimizers
        scheduler_type='linear',
        lr_factor=0.01,
        max_epochs=max_epochs),
    checkpoint=dict(
        type='CheckpointHook', # Hook to save model checkpoint on specific intervals
        interval=save_epoch_intervals, # Save model checkpoint every 10 epochs.
        max_keep_ckpts=3)) # The maximum checkpoints to keep.
```

`custom_hooks` is a list of hook configs. Users can develop their hooks and insert them in this field.

```
custom_hooks = [
    dict(
        type='EMAHook', # A Hook to apply Exponential Moving Average (EMA) on the model␣
↪during training.
        ema_type='ExpMomentumEMA', # The type of EMA strategy to use.
        momentum=0.0001, # The momentum of EMA
        update_buffers=True, # # If ``True``, calculate the running averages of model␣
↪parameters
        priority=49) # Priority higher than NORMAL(50)
]
```

45.1.7 Runtime config

```
default_scope = 'mmyolo'  # The default registry scope to find modules. Refer to https://
↪mmengine.readthedocs.io/en/latest/tutorials/registry.html

env_cfg = dict(
    cudnn_benchmark=True,  # Whether to enable cudnn benchmark
    mp_cfg=dict(  # Multi-processing config
        mp_start_method='fork',  # Use fork to start multi-processing threads. 'fork' is␣
↪usually faster than 'spawn' but may be unsafe. See discussion in https://github.com/
↪pytorch/pytorch/issues/1355
        opencv_num_threads=0),  # Disable opencv multi-threads to avoid system being␣
↪overloaded
    dist_cfg=dict(backend='nccl'),  # Distribution configs
)

vis_backends = [dict(type='LocalVisBackend')]  # Visualization backends. Refer to:␣
↪https://mmengine.readthedocs.io/zh_CN/latest/advanced_tutorials/visualization.html
visualizer = dict(
    type='mmdet.DetLocalVisualizer', vis_backends=vis_backends, name='visualizer')
log_processor = dict(
    type='LogProcessor',  # Log processor to process runtime logs
    window_size=50,  # Smooth interval of log values
    by_epoch=True)  # Whether to format logs with epoch style. Should be consistent with␣
↪the train loop's type.

log_level = 'INFO'  # The level of logging.
load_from = None  # Load model checkpoint as a pre-trained model from a given path. This␣
↪will not resume training.
resume = False  # Whether to resume from the checkpoint defined in `load_from`. If `load_
↪from` is None, it will resume the latest checkpoint in the `work_dir`.
```

45.2 Config file inheritance

`config/_base_` contains default runtime. The configs that are composed of 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 be inherited from the *primitive* config. In this way, the maximum inheritance level is 3.

For easy understanding, we recommend contributors inherit from existing methods. For example, if some modification is made based on YOLOv5-s, such as modifying the depth of the network, users may first inherit the `_base_ = ./yolov5_s-v61_syncbn_8xb16-300e_coco.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 `yolov100` under `configs`,

Please refer to the mmengine config tutorial for more details.

By setting the `_base_` field, we can set which files the current configuration file inherits from.

When `_base_` is a string of a file path, it means inheriting the contents of one config file.

```
_base_ = '../_base_/default_runtime.py'
```

When _base_ is a list of multiple file paths, it means inheriting multiple files.

```
_base_ = [
    './yolov5_s-v61_syncbn_8xb16-300e_coco.py',
    '../_base_/default_runtime.py'
]
```

If you wish to inspect the config file, you may run `mim run mmdet print_config /PATH/TO/CONFIG` to see the complete config.

45.2.1 Ignore some fields in the base configs

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

In MMYOLO, for example, to change the backbone of RTMDet with the following config.

```
model = dict(
    type='YOLODetector',
    data_preprocessor=dict(...),
    backbone=dict(
        type='CSPNeXt',
        arch='P5',
        expand_ratio=0.5,
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        channel_attention=True,
        norm_cfg=dict(type='BN'),
        act_cfg=dict(type='SiLU', inplace=True)),
    neck=dict(...),
    bbox_head=dict(...))
```

If you want to change `CSPNeXt` to `YOLOv6EfficientRep` for the RTMDet backbone, because there are different fields (`channel_attention` and `expand_ratio`) in `CSPNeXt` and `YOLOv6EfficientRep`, you need to use `_delete_=True` to replace all the old keys in the `backbone` field with the new keys.

```
_base_ = '../rtmdet/rtmdet_l_syncbn_8xb32-300e_coco.py'
model = dict(
    backbone=dict(
        _delete_=True,
        type='YOLOv6EfficientRep',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='ReLU', inplace=True)),
    neck=dict(...),
    bbox_head=dict(...))
```

45.2.2 Use intermediate variables in configs

Some intermediate variables are used in the configs files, like `train_pipeline` and `test_pipeline` in datasets. It's worth noting that when modifying intermediate variables in the children configs, users need to pass the intermediate variables into corresponding fields again. For example, we would like to change the `image_scale` during training and add `YOLOv5MixUp` data augmentation, `img_scale/train_pipeline/test_pipeline` are intermediate variables we would like to modify.

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

img_scale = (1280, 1280)  # image height, image width
affine_scale = 0.9

mosaic_affine_pipeline = [
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='YOLOv5RandomAffine',
        max_rotate_degree=0.0,
        max_shear_degree=0.0,
        scaling_ratio_range=(1 - affine_scale, 1 + affine_scale),
        border=(-img_scale[0] // 2, -img_scale[1] // 2),
        border_val=(114, 114, 114))
]

train_pipeline = [
    *pre_transform, *mosaic_affine_pipeline,
    dict(
        type='YOLOv5MixUp',         # MixUp augmentation of YOLOv5
        prob=0.1, # the probability of YOLOv5MixUp
        pre_transform=[*pre_transform,*mosaic_affine_pipeline]), # Pre-defined Training␣
↪data pipeline and MixUp augmentation.
    dict(
        type='mmdet.Albu',
        transforms=albu_train_transforms,
        bbox_params=dict(
            type='BboxParams',
            format='pascal_voc',
            label_fields=['gt_bboxes_labels', 'gt_ignore_flags']),
        keymap={
            'img': 'image',
            'gt_bboxes': 'bboxes'
        }),
    dict(type='YOLOv5HSVRandomAug'),
    dict(type='mmdet.RandomFlip', prob=0.5),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'flip',
                   'flip_direction'))
]
```

(continues on next page)

(continued from previous page)

```
test_pipeline = [
    dict(
        type='LoadImageFromFile'),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

train_dataloader = dict(dataset=dict(pipeline=train_pipeline))
val_dataloader = dict(dataset=dict(pipeline=test_pipeline))
test_dataloader = dict(dataset=dict(pipeline=test_pipeline))
```

We first define a new `train_pipeline/test_pipeline` and pass it into `data`.

Likewise, if we want to switch from `SyncBN` to `BN` or `MMSyncBN`, we need to modify every `norm_cfg` in the configuration file.

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
norm_cfg = dict(type='BN', requires_grad=True)
model = dict(
    backbone=dict(norm_cfg=norm_cfg),
    neck=dict(norm_cfg=norm_cfg),
    ...)
```

45.2.3 Reuse variables in _base_ file

If the users want to reuse the variables in the base file, they can get a copy of the corresponding variable by using `{{_base_.xxx}}`. The latest version of MMEngine also supports reusing variables without `{{}}` usage.

E.g:

```
_base_ = '../_base_/default_runtime.py'

pre_transform = _base_.pre_transform # `pre_transform` equals to `pre_transform` in the _
↪base_ config
```

45.3 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.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 training pipeline `train_dataloader.dataset.pipeline` is normally a list, e.g. `[dict(type='LoadImageFromFile'), ...]`. If you want to change `'LoadImageFromFile'` to `'LoadImageFromNDArray'` in the pipeline, you may specify `--cfg-options data.train.pipeline.0.type=LoadImageFromNDArray`.

- Update values of list/tuples.

  Sometimes the value to update is a list or a tuple, for example, the config file normally sets `model.data_preprocessor.mean=[123.675, 116.28, 103.53]`. If you want to change the mean values, you may specify `--cfg-options model.data_preprocessor.mean="[127,127,127]"`. 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.

45.4 Config name style

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

```
{algorithm name}_{model component names [component1]_[component2]_[...]}-[version id]_
↪[norm setting]_[data preprocessor type]_{training settings}_{training dataset␣
↪information}_[testing dataset information].py
```

The file name is divided into 8 name fields, which have 4 required parts and 4 optional parts. All parts and components are connected with _ and words of each part or component should be connected with -. `{}` indicates the required name field, and `[]` indicates the optional name field.

- `{algorithm name}`: The name of the algorithm. It can be a detector name such as `yolov5`, `yolov6`, `yolox`, etc.
- `{component names}`: Names of the components used in the algorithm such as backbone, neck, etc. For example, `yolov5_s` means its `deepen_factor` is `0.33` and its `widen_factor` is `0.5`.
- `[version_id]` (optional): Since the evolution of the YOLO series is much faster than traditional object detection algorithms, `version id` is used to distinguish the differences between different sub-versions. E.g, YOLOv5-3.0 uses the `Focus` layer as the stem layer, and YOLOv5-6.0 uses the `Conv` layer as the stem layer.
- `[norm_setting]` (optional): `bn` indicates `Batch Normalization`, `syncbn` indicates `Synchronized Batch Normalization`
- `[data preprocessor type]` (optional): `fast` incorporates YOLOv5DetDataPreprocessor and yolov5_collate to preprocess data. The training speed is faster than the default `mmdet.DetDataPreprocessor`, while results in extending the overall pipeline to multi-task learning.
- `{training settings}`: Information of training settings such as batch size, augmentations, loss trick, scheduler, and epochs/iterations. For example: `8xb16-300e_coco` means using 8-GPUs x 16-images-per-GPU, and train 300 epochs. Some abbreviations:

  - {gpu x batch_per_gpu}: GPUs and samples per GPU. For example, 4xb4 is the short term of 4-GPUs x 4-images-per-GPU.
  - {schedule}: training schedule, default option in MMYOLO is 300 epochs.
- {training dataset information}: Training dataset names like coco, cityscapes, voc-0712, wider-face, and balloon.
- [testing dataset information] (optional): Testing dataset name for models trained on one dataset but tested on another. If not mentioned, it means the model was trained and tested on the same dataset type.

CHAPTER

FORTYSIX

MIXED IMAGE DATA AUGMENTATION UPDATE

Mixed image data augmentation is similar to Mosaic and MixUp, in which the annotation information of multiple images needs to be obtained for fusion during the running process. In the OpenMMLab data augmentation pipeline, other indexes of the dataset are generally not available. In order to achieve the above function, in the YOLOX reproduced in MMDetection, the concept of MultiImageMixDataset dataset wrapper is proposed.

`MultiImageMixDataset` dataset wrapper will include some data augmentation methods such as `Mosaic` and `RandAffine`, while `CocoDataset` will also need to include a `pipeline` to achieve the image and annotation loading function. In this way, we can achieve mixed data augmentation quickly. The configuration method is as follows:

```
train_pipeline = [
    dict(type='Mosaic', img_scale=img_scale, pad_val=114.0),
    dict(
        type='RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
    dict(
        type='MixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0),
    ...
]
train_dataset = dict(
    # use MultiImageMixDataset wrapper to support mosaic and mixup
    type='MultiImageMixDataset',
    dataset=dict(
        type='CocoDataset',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations', with_bbox=True)
        ]),
    pipeline=train_pipeline)
```

However, this implementation has a disadvantage: users unfamiliar with MMDetection will forget those data augmentation methods like Mosaic must be used together with `MultiImageMixDataset`, increasing the usage complexity. Moreover, it is hard to understand as well.

To address this problem, further simplifications are made in MMYOLO, which directly lets `pipeline` get `dataset`. In this way, the implementation of `Mosaic` and other data augmentation methods can be achieved and used just as the random flip, without a data wrapper anymore. The new configuration method is as follows:

```
pre_transform = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True)
]
train_pipeline = [
    *pre_transform,
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='mmdet.RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
    dict(
        type='YOLOXMixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0,
        pre_transform=pre_transform),
    ...
]
```

A more complex YOLOv5-m configuration including MixUp is shown as follows:

```
mosaic_affine_pipeline = [
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='YOLOv5RandomAffine',
        max_rotate_degree=0.0,
        max_shear_degree=0.0,
        scaling_ratio_range=(1 - affine_scale, 1 + affine_scale),
        border=(-img_scale[0] // 2, -img_scale[1] // 2),
        border_val=(114, 114, 114))
]

# enable mixup
train_pipeline = [
    *pre_transform, *mosaic_affine_pipeline,
    dict(
        type='YOLOv5MixUp',
        prob=0.1,
        pre_transform=[*pre_transform, *mosaic_affine_pipeline]),
    dict(
        type='mmdet.Albu',
        transforms=albu_train_transforms,
        bbox_params=dict(
            type='BboxParams',
```

(continues on next page)

(continued from previous page)

```
            format='pascal_voc',
            label_fields=['gt_bboxes_labels', 'gt_ignore_flags']),
        keymap={
            'img': 'image',
            'gt_bboxes': 'bboxes'
        }),
    dict(type='YOLOv5HSVRandomAug'),
    dict(type='mmdet.RandomFlip', prob=0.5),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'flip',
                   'flip_direction'))
]
```

It is very easy to use, just pass the object of Dataset to the pipeline.

```
def prepare_data(self, idx) -> Any:
   """Pass the dataset to the pipeline during training to support mixed
   data augmentation, such as Mosaic and MixUp."""
   if self.test_mode is False:
        data_info = self.get_data_info(idx)
        data_info['dataset'] = self
        return self.pipeline(data_info)
    else:
        return super().prepare_data(idx)
```

CHAPTER

FORTYSEVEN

CUSTOMIZE INSTALLATION

47.1 CUDA versions

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

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

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

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

47.2 Install MMEngine without MIM

To install MMEngine with pip instead of MIM, please follow [MMEngine installation guides](https://mmengine.readthedocs.io/en/latest/get_started/installation.html).

For example, you can install MMEngine by the following command.

```
pip install "mmengine>=0.6.0"
```

47.3 Install MMCV without MIM

MMCV contains C++ and CUDA extensions, thus depending on PyTorch in a complex way. MIM solves such dependencies automatically and makes the installation easier. However, it is not a must.

To install MMCV with pip instead of MIM, please follow MMCV installation guides. This requires manually specifying a find-url based on the PyTorch version and its CUDA version.

For example, the following command installs MMCV built for PyTorch 1.12.x and CUDA 11.6.

```
pip install "mmcv>=2.0.0rc4" -f https://download.openmmlab.com/mmcv/dist/cu116/torch1.12.
↪0/index.html
```

47.4 Install on CPU-only platforms

MMDetection can be built for the CPU-only environment. In CPU mode you can train (requires MMCV version >= `2.0.0rc1`), test, or infer a model.

However, some functionalities are gone in this mode:

- Deformable Convolution
- Modulated Deformable Convolution
- ROI pooling
- Deformable ROI pooling
- CARAFE
- SyncBatchNorm
- CrissCrossAttention
- MaskedConv2d
- Temporal Interlace Shift
- nms_cuda
- sigmoid_focal_loss_cuda
- bbox_overlaps

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

47.5 Install on Google Colab

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

Step 1. Install MMEngine and MMCV using MIM.

```
!pip3 install openmim
!mim install "mmengine>=0.6.0"
!mim install "mmcv>=2.0.0rc4,<2.1.0"
!mim install "mmdet>=3.0.0,<4.0.0"
```

Step 2. Install MMYOLO from the source.

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

Step 3. Verification.

```
import mmyolo
print(mmyolo.__version__)
# Example output: 0.1.0, or an another version.
```

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

47.6 Develop using multiple MMYOLO versions

The training and testing scripts have been modified in `PYTHONPATH` to ensure that the scripts use MMYOLO in the current directory.

To have the default MMYOLO installed in your environment instead of what is currently in use, you can remove the code that appears in the relevant script:

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

CHAPTER

FORTYEIGHT

COMMON WARNING NOTES

The purpose of this document is to collect warning messages that users often find confusing, and provide explanations to facilitate understanding.

48.1 xxx registry in mmyolo did not set import location

The warning message complete information is that The xxx registry in mmyolo did not set import location. Fallback to call `mmyolo.utils.register_all_modules` instead.

This warning means that a module was not set with an import location when importing it, making it impossible to determine its location. Therefore, `mmyolo.utils.register_all_modules` is automatically called to trigger the package import. This warning belongs to the very low-level module warning in MMEngine, which may be difficult for users to understand, but it has no impact on the actual use and can be ignored directly.

48.2 save_param_schedulers is true but self.param_schedulers is None

The following information is an example using the YOLOv5 algorithm. This is because the parameter scheduler strategy `YOLOv5ParamSchedulerHook` has been rewritten in YOLOv5, so the ParamScheduler designed in MMEngine is not used. However, `save_param_schedulers` is not set to False in the YOLOv5 configuration.

First of all, this warning has no impact on performance and resuming training. If users think this warning affects experience, you can set `default_hooks.checkpoint.save_param_scheduler` to False, or set `--cfg-options default_hooks.checkpoint.save_param_scheduler=False` when training via the command line.

48.3 The loss_cls will be 0. This is a normal phenomenon.

This is related to specific algorithms. Taking YOLOv5 as an example, its classification loss only considers positive samples. If the number of classes is 1, then the classification loss and object loss are functionally redundant. Therefore, in the design, when the number of classes is 1, the loss_cls is not calculated and is always 0. This is a normal phenomenon.

48.4 The model and loaded state dict do not match exactly

Whether this warning will affect performance needs to be determined based on more information. If it occurs during fine-tuning, it is a normal phenomenon that the COCO pre-trained weights of the Head module cannot be loaded due to the user's custom class differences, and it will not affect performance.

CHAPTER

FORTYNINE

FREQUENTLY ASKED QUESTIONS

We list some common problems many users face and their corresponding solutions here. Feel free to enrich the list if you find any frequent issues and have ways to help others to solve them. If the contents here do not cover your issue, please create an issue and make sure you fill in all the required information in the template.

49.1 Why do we need to launch MMYOLO?

Why do we need to launch MMYOLO? Why do we need to open a separate repository instead of putting it directly into MMDetection? Since the open source, we have been receiving similar questions from our community partners, and the answers can be summarized in the following three points.

(1) Unified operation and inference platform

At present, there are very many improved algorithms for YOLO in the field of target detection, and they are very popular, but such algorithms are based on different frameworks for different back-end implementations, and there are significant differences, lacking a unified and convenient fair evaluation process from training to deployment.

(2) Protocol limitations

As we all know, YOLOv5 and its derived algorithms, such as YOLOv6 and YOLOv7 are GPL 3.0 protocols, which differ from the Apache protocol of MMDetection. Therefore, due to the protocol issue, it is not possible to incorporate MMYOLO directly into MMDetection.

(3) Multitasking support

There is another far-reaching reason: **MMYOLO tasks are not limited to MMDetection**, and more tasks will be supported in the future, such as MMPose based keypoint-related applications and MMTracking based tracking related applications, so it is not suitable to be directly incorporated into MMDetection.

49.2 What is the projects folder used for?

The `projects` folder is newly introduced in OpenMMLab 2.0. There are three primary purposes:

1. facilitate community contributors: Since OpenMMLab series codebases have a rigorous code management process, this inevitably leads to long algorithm reproduction cycles, which is not friendly to community contributions.
2. facilitate rapid support for new algorithms: A long development cycle can also lead to another problem users may not be able to experience the latest algorithms as soon as possible.
3. facilitate rapid support for new approaches and features: New approaches or new features may be incompatible with the current design of the codebases and cannot be quickly incorporated.

In summary, the `projects` folder solves the problems of slow support for new algorithms and complicated support for new features due to the long algorithm reproduction cycle. Each folder in `projects` is an entirely independent project, and community users can quickly support some algorithms in the current version through `projects`. This allows the community to quickly use new algorithms and features that are difficult to adapt in the current version. When the design is stable or the code meets the merge specification, it will be considered to merge into the main branch.

49.3 Why does the performance drop significantly by switching the YOLOv5 backbone to Swin?

In *Replace the backbone network*, we provide many tutorials on replacing the backbone module. However, you may not get a desired result once you replace the module and start directly training the model. This is because different networks have very distinct hyperparameters. Take the backbones of Swin and YOLOv5 as an example. Swin belongs to the transformer family, and the YOLOv5 is a convolutional network. Their training optimizers, learning rates, and other hyperparameters are different. If we force using Swin as the backbone of YOLOv5 and try to get a moderate performance, we must modify many parameters.

49.4 How to use the components implemented in all MM series repositories?

In OpenMMLab 2.0, we have enhanced the ability to use different modules across MM series libraries. Currently, users can call any module that has been registered in MM series algorithm libraries via `MM Algorithm Library A. Module Name`. We demonstrated using MMClassification backbones in the *Replace the backbone network*. Other modules can be used in the same way.

49.5 Can pure background pictures be added in MMYOLO for training?

Adding pure background images to training can suppress the false positive rate in most scenarios, and this feature has already been supported for most datasets. Take `YOLOv5CocoDataset` as an example. The control parameter is `train_dataloader.dataset.filter_cfg.filter_empty_gt`. If `filter_empty_gt` is True, the pure background images will be filtered out and not used in training, and vice versa. Most of the algorithms in MMYOLO have added this feature by default.

49.6 Is there a script to calculate the inference FPS in MMYOLO?

MMYOLO is based on MMDet 3.x, which provides a benchmark script to calculate the inference FPS. We recommend using `mim` to run the script in MMDet directly across the library instead of copying them to MMYOLO. More details about `mim` usages can be found at *Use mim to run scripts from other OpenMMLab repositories*.

49.7 What is the difference between MMDeploy and EasyDeploy?

MMDeploy is developed and maintained by the OpenMMLab deployment team to provide model deployment solutions for the OpenMMLab series algorithms, which support various inference backends and customization features. EasyDeploy is an easier and more lightweight deployment project provided by the community. However, it does not support as many features as MMDeploy. Users can choose which one to use in MMYOLO according to their needs.

49.8 How to check the AP of every category in COCOMetric?

Just set `test_evaluator.classwise` to True or add `--cfg-options test_evaluator.classwise=True` when running the test script.

49.9 Why doesn't MMYOLO support the auto-learning rate scaling feature as MMDet?

It is because the YOLO series algorithms are not very well suited for linear scaling. We have verified on several datasets that the performance is better without the auto-scaling based on batch size.

49.10 Why is the weight size of my trained model larger than the official one?

The reason is that user-trained weights usually include extra data such as `optimizer`, `ema_state_dict`, and `message_hub`, which are removed when we publish the models. While on the contrary, the weight users trained by themselves are kept. You can use the publish_model.py to remove these unnecessary components.

49.11 Why does the RTMDet cost more graphics memory during the training than YOLOv5?

It is due to the assigner in RTMDet. YOLOv5 uses a simple and efficient shape-matching assigner, while RTMDet uses a dynamic soft label assigner for entire batch computation. Therefore, it consumes more memory in its internal cost matrix, especially when there are too many labeled bboxes in the current batch. We are considering solving this problem soon.

49.12 Do I need to reinstall MMYOLO after modifying some code?

Without adding any new python code, and if you installed the MMYOLO by `mim install -v -e .`, any new modifications will take effect without reinstalling. However, if you add new python codes and are using them, you need to reinstall with `mim install -v -e .`.

49.13 How to use multiple versions of MMYOLO to develop?

If users have multiple versions of the MMYOLO, such as mmyolo-v1 and mmyolo-v2. They can specify the target version of their MMYOLO by using this command in the shell:

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

Users can unset the PYTHONPATH when they want to reset to the default MMYOLO by this command:

```
unset PYTHONPATH
```

49.14 How to save the best checkpoints during the training?

Users can choose what metrics to filter the best models by setting the `default_hooks.checkpoint.save_best` in the configuration. Take the COCO dataset detection task as an example. Users can customize the `default_hooks.checkpoint.save_best` with these parameters:

1. `auto` works based on the first evaluation metric in the validation set.
2. `coco/bbox_mAP` works based on `bbox_mAP`.
3. `coco/bbox_mAP_50` works based on `bbox_mAP_50`.
4. `coco/bbox_mAP_75` works based on `bbox_mAP_75`.
5. `coco/bbox_mAP_s` works based on `bbox_mAP_s`.
6. `coco/bbox_mAP_m` works based on `bbox_mAP_m`.
7. `coco/bbox_mAP_l` works based on `bbox_mAP_l`.

In addition, users can also choose the filtering logic by setting `default_hooks.checkpoint.rule` in the configuration. For example, `default_hooks.checkpoint.rule=greater` means that the larger the indicator is, the better it is. More details can be found at checkpoint_hook.

49.15 How to train and test with non-square input sizes?

The default configurations of the YOLO series algorithms are mostly squares like 640x640 or 1280x1280. However, if users want to train with a non-square shape, they can modify the `image_scale` to the desired value in the configuration. A more detailed example could be found at yolov5_s-v61_fast_1xb12-40e_608x352_cat.py.

CHAPTER

FIFTY

MMYOLO CROSS-LIBRARY APPLICATION

CHAPTER

FIFTYONE

MODEL ZOO AND BENCHMARK

This page is used to summarize the performance and related evaluation metrics of various models supported in MMYOLO for users to compare and analyze.

51.1 COCO dataset

- All the models are trained on COCO train2017 dataset and evaluated on val2017 dataset.
- TRT-FP16-GPU-Latency(ms) is the GPU Compute time on NVIDIA Tesla T4 device with TensorRT 8.4, a batch size of 1, a test shape of 640x640 and only model forward (The test shape for YOLOX-tiny is 416x416)
- The number of model parameters and FLOPs are obtained using the get_flops script. Different calculation methods may vary slightly
- RTMDet performance is the result of training with MMRazor Knowledge Distillation
- Only YOLOv6 version 2.0 is implemented in MMYOLO for now, and L and M are the results without knowledge distillation
- YOLOv8 results are optimized using mask instance annotation, but YOLOv5, YOLOv6 and YOLOv7 do not use
- PPYOLOE+ uses Obj365 as pre-training weights, so the number of epochs for COCO training only needs 80
- YOLOX-tiny, YOLOX-s and YOLOX-m are trained with the optimizer parameters proposed in RTMDet, with different degrees of performance improvement compared to the original implementation.

Please see below items for more details

- RTMDet
- YOLOv5
- YOLOv6
- YOLOv7
- YOLOv8
- YOLOX
- PPYOLO-E

51.2 VOC dataset

Please see below items for more details

- YOLOv5

51.3 CrowdHuman dataset

Please see below items for more details

- YOLOv5

51.4 DOTA 1.0 dataset

CHAPTER

FIFTYTWO

CHANGELOG

52.1 v0.6.0 (15/8/2023)

52.1.1 Highlights

- Support YOLOv5 instance segmentation
- Support YOLOX-Pose based on MMPose
- Add 15 minutes instance segmentation tutorial.
- YOLOv5 supports using mask annotation to optimize bbox
- Add Multi-scale training and testing docs

52.1.2 New Features

- Add training and testing tricks doc (#659)
- Support setting the cache_size_limit parameter and support mmdet 3.0.0 (#707)
- Support YOLOv5u and YOLOv6 3.0 inference (#624, #744)
- Support model-only inference (#733)
- Add YOLOv8 deepstream config (#633)
- Add ionogram example in MMYOLO application (#643)

52.1.3 Bug Fixes

- Fix the browse_dataset for visualization of test and val (#641)
- Fix installation doc error (#662)
- Fix yolox-l ckpt link (#677)
- Fix typos in the YOLOv7 and YOLOv8 diagram (#621, #710)
- Adjust the order of package imports in `boxam_vis_demo.py` (#655)

52.1.4 Improvements

- Optimize the `convert_kd_ckpt_to_student.py` file (#647)
- Add en doc of `FAQ` and `training_testing_tricks` (#691,#693)

52.1.5 Contributors

A total of 21 developers contributed to this release.

Thank @Lum1104,@azure-wings,@FeiGeChuanShu,@Lingrui Gu,@Nioolek,@huayuan4396,@RangeKing,@danielhonies,@yechenz
Mu,@kikefdezl,@zhangrui-wolf,@xin-li-67,@Ben-Louis,@zgzhengSEU,@VoyagerXvoyagerx,@tang576225574,@hhaAndroid

52.2 v0.5.0 (2/3/2023)

52.2.1 Highlights

1. Support RTMDet-R rotated object detection
2. Support for using mask annotation to improve YOLOv8 object detection performance
3. Support MMRazor searchable NAS sub-network as the backbone of YOLO series algorithm
4. Support calling MMRazor to distill the knowledge of RTMDet
5. MMYOLO document structure optimization, comprehensive content upgrade
6. Improve YOLOX mAP and training speed based on RTMDet training hyperparameters
7. Support calculation of model parameters and FLOPs, provide GPU latency data on T4 devices, and update Model Zoo
8. Support test-time augmentation (TTA)
9. Support RTMDet, YOLOv8 and YOLOv7 assigner visualization

52.2.2 New Features

1. Support inference for RTMDet instance segmentation tasks (#583)
2. Beautify the configuration file in MMYOLO and add more comments (#501, #506, #516, #529, #531, #539)
3. Refactor and optimize documentation (#568, #573, #579, #584, #587, #589, #596, #599, #600)
4. Support fast version of YOLOX (#518)
5. Support DeepStream in EasyDeploy and add documentation (#485, #545, #571)
6. Add confusion matrix drawing script (#572)
7. Add single channel application case (#460)
8. Support auto registration (#597)
9. Support Box CAM of YOLOv7, YOLOv8 and PPYOLOE (#601)
10. Add automated generation of MM series repo registration information and tools scripts (#559)
11. Added YOLOv7 model structure diagram (#504)
12. Add how to specify specific GPU training and inference files (#503)

13. Add check if `metainfo` is all lowercase when training or testing (#535)
14. Add links to Twitter, Discord, Medium, YouTube, etc. (#555)

52.2.3 Bug Fixes

1. Fix isort version issue (#492, #497)
2. Fix type error of assigner visualization (#509)
3. Fix YOLOv8 documentation link error (#517)
4. Fix RTMDet Decoder error in EasyDeploy (#519)
5. Fix some document linking errors (#537)
6. Fix RTMDet-Tiny weight path error (#580)

52.2.4 Improvements

1. Update `contributing.md`
2. Optimize `DetDataPreprocessor` branch to support multitasking (#511)
3. Optimize `gt_instances_preprocess` so it can be used for other YOLO algorithms (#532)
4. Add `yolov7-e6e` weight conversion script (#570)
5. Reference YOLOv8 inference code modification PPYOLOE

52.2.5 Contributors

A total of 22 developers contributed to this release.

Thank @triple-Mu, @isLinXu, @Audrey528, @TianWen580, @yechenzhi, @RangeKing, @lyviva, @Nioolek, @PeterH0323, @tianleiSHI, @aptsunny, @satuoqaq, @vansin, @xin-li-67, @VoyagerXvoyagerx, @landhill, @kitecats, @tang576225574, @HIT-cwh, @AI-Tianlong, @RangiLyu, @hhaAndroid

52.3 v0.4.0 (18/1/2023)

52.3.1 Highlights

1. Implemented YOLOv8 object detection model, and supports model deployment in projects/easydeploy
2. Added Chinese and English versions of Algorithm principles and implementation with YOLOv8

52.3.2 New Features

1. Added YOLOv8 and PPYOLOE model structure diagrams (#459, #471)
2. Adjust the minimum supported Python version from 3.6 to 3.7 (#449)
3. Added a new YOLOX decoder in TensorRT-8 (#450)
4. Add a tool for scheduler visualization (#479)

52.3.3 Bug Fixes

1. Fix `optimize_anchors.py` script import error (#452)
2. Fix the wrong installation steps in `get_started.md` (#474)
3. Fix the neck error when using the `RTMDet` P6 model (#480)

52.3.4 Contributors

A total of 9 developers contributed to this release.

Thank @VoyagerXvoyagerx, @tianleiSHI, @RangeKing, @PeterH0323, @Nioolek, @triple-Mu, @lyviva, @Zheng-LinXiao, @hhaAndroid

52.4 v0.3.0 (8/1/2023)

52.4.1 Highlights

1. Implement fast version of RTMDet. RTMDet-s 8xA100 training takes only 14 hours. The training speed is 2.6 times faster than the previous version.
2. Support PPYOLOE training
3. Support `iscrowd` attribute training in YOLOv5
4. Support YOLOv5 assigner result visualization

52.4.2 New Features

1. Add `crowdhuman` dataset (#368)
2. Easydeploy support TensorRT inference (#377)
3. Add `YOLOX` structure description (#402)
4. Add a feature for the video demo (#392)
5. Support `YOLOv7` easy deploy (#427)
6. Add resume from specific checkpoint in CLI (#393)
7. Set `metainfo` fields to lower case (#362, #412)
8. Add module combination doc (#349, #352, #345)
9. Add docs about how to freeze the weight of backbone or neck (#418)

10. Add don't used pre-training weights doc in how_to.md (#404)
11. Add docs about how to set the random seed (#386)
12. Translate rtmdet_description.md document to English (#353)
13. Add doc of yolov6_description.md (#382, #372)

52.4.3 Bug Fixes

1. Fix bugs in the output annotation file when --class-id-txt is set (#430)
2. Fix batch inference bug in YOLOv5 head (#413)
3. Fix typehint in some heads (#415, #416, #443)
4. Fix RuntimeError of torch.cat() expected a non-empty list of Tensors (#376)
5. Fix the device inconsistency error in YOLOv7 training (#397)
6. Fix the scale_factor and pad_param value in LetterResize (#387)
7. Fix docstring graph rendering error of readthedocs (#400)
8. Fix AssertionError when YOLOv6 from training to val (#378)
9. Fix CI error due to np.int and legacy builder.py (#389)
10. Fix MMDeploy rewriter (#366)
11. Fix MMYOLO unittest scope bug (#351)
12. Fix pad_param error (#354)
13. Fix twice head inference bug (#342)
14. Fix customize dataset training (#428)

52.4.4 Improvements

1. Update useful_tools.md (#384)
2. update the English version of custom_dataset.md (#381)
3. Remove context argument from the rewriter function (#395)
4. deprecating np.bool type alias (#396)
5. Add new video link for custom dataset (#365)
6. Export onnx for model only (#361)
7. Add MMYOLO regression test yml (#359)
8. Update video tutorials in article.md (#350)
9. Add deploy demo (#343)
10. Optimize the vis results of large images in debug mode (#346)
11. Improve args for browse_dataset and support RepeatDataset (#340, #338)

52.4.5 Contributors

A total of 28 developers contributed to this release.

Thank @RangeKing, @PeterH0323, @Nioolek, @triple-Mu, @matrixgame2018, @xin-li-67, @tang576225574, @kitecats, @Seperendity, @diplomatist, @vaew, @wzr-skn, @VoyagerXvoyagerx, @MambaWong, @tianleiSHI, @caj-github, @zhubochao, @lvhan028, @dsghaonan, @lyviva, @yuewangg, @wang-tf, @satuoqaq, @grimoire, @RunningLeon, @hanrui1sensetime, @RangiLyu, @hhaAndroid

52.5 v0.2.01/12/2022)

52.5.1 Highlights

1. Support YOLOv7 P5 and P6 model
2. Support YOLOv6 ML model
3. Support Grad-Based CAM and Grad-Free CAM
4. Support large image inference based on sahi
5. Add easydeploy project under the projects folder
6. Add custom dataset guide

52.5.2 New Features

1. `browse_dataset.py` script supports visualization of original image, data augmentation and intermediate results (#304)
2. Add flag to output labelme label file in `image_demo.py` (#288, #314)
3. Add `labelme2coco` script (#308, #313)
4. Add split COCO dataset script (#311)
5. Add two examples of backbone replacement in `how-to.md` and update `plugin.md` (#291)
6. Add `contributing.md` and `code_style.md` (#322)
7. Add docs about how to use mim to run scripts across libraries (#321)
8. Support `YOLOv5` deployment at RV1126 device (#262)

52.5.3 Bug Fixes

1. Fix MixUp padding error (#319)
2. Fix scale factor order error of `LetterResize` and `YOLOv5KeepRatioResize` (#305)
3. Fix training errors of `YOLOX Nano` model (#285)
4. Fix `RTMDet` deploy error (#287)
5. Fix int8 deploy config (#315)
6. Fix `make_stage_plugins` doc in `basebackbone` (#296)
7. Enable switch to deploy when create pytorch model in deployment (#324)

8. Fix some errors in `RTMDet` model graph (#317)

52.5.4 Improvements

1. Add option of json output in `test.py` (#316)
2. Add area condition in `extract_subcoco.py` script (#286)
3. Deployment doc translation (#289)
4. Add YOLOv6 description overview doc (#252)
5. Improve `config.md` (#297, #303) 6Add mosaic9 graph in docstring (#307)
6. Improve `browse_coco_json.py` script args (#309)
7. Refactor some functions in `dataset_analysis.py` to be more general (#294)

Contributors

A total of 14 developers contributed to this release.

Thank @fcakyon, @matrixgame2018, @MambaWong, @imAzhou, @triple-Mu, @RangeKing, @PeterH0323, @xin-li-67, @kitecats, @hanrui1sensetime, @AllentDan, @Zheng-LinXiao, @hhaAndroid, @wanghonglie

52.6 v0.1.310/11/2022)

52.6.1 New Features

1. Support CBAM plug-in and provide plug-in documentation (#246)
2. Add YOLOv5 P6 model structure diagram and related descriptions (#273)

52.6.2 Bug Fixes

1. Fix training failure when saving best weights based on mmengine 0.3.1
2. Fix `add_dump_metric` error based on mmdet 3.0.0rc3 (#253)
3. Fix backbone does not support `init_cfg` issue (#272)
4. Change typing import method based on mmdet 3.0.0rc3 (#261)

52.6.3 Improvements

1. `featmap_vis_demo` support for folder and url input (#248)
2. Deploy docker file refinement (#242)

Contributors

A total of 10 developers contributed to this release.

Thank @kitecats, @triple-Mu, @RangeKing, @PeterH0323, @Zheng-LinXiao, @tkhe, @weikai520, @zytx121, @wanghonglie, @hhaAndroid

52.7 v0.1.23/11/2022)

52.7.1 Highlights

1. Support YOLOv5/YOLOv6/YOLOX/RTMDet deployments for ONNXRuntime and TensorRT
2. Support YOLOv6 s/t/n model training
3. YOLOv5 supports P6 model training which can input 1280-scale images
4. YOLOv5 supports VOC dataset training
5. Support PPYOLOE and YOLOv7 model inference and official weight conversion
6. Add YOLOv5 replacement backbone tutorial in How-to documentation

52.7.2 New Features

1. Add `optimize_anchors` script (#175)
2. Add `extract_subcoco` script (#186)
3. Add `yolo2coco` conversion script (#161)
4. Add `dataset_analysis` script (#172)
5. Remove Albu version restrictions (#187)

52.7.3 Bug Fixes

1. Fix the problem that `cfg.resume` does not work when set (#221)
2. Fix the problem of not showing bbox in feature map visualization script (#204)
3. uUpdate the metafile of RTMDet (#188)
4. Fix a visualization error in `test_pipeline` (#166)
5. Update badges (#140)

52.7.4 Improvements

1. Optimize Readthedoc display page (#209)
2. Add docstring for module structure diagram for base model (#196)
3. Support for not including any instance logic in LoadAnnotations (#161)
4. Update `image_demo` script to support folder and url paths (#128)
5. Update pre-commit hook (#129)

52.7.5 Documentation

1. Translate `yolov5_description.md`, `yolov5_tutorial.md` and `visualization.md` into English (#138, #198, #206)
2. Add deployment-related Chinese documentation (#220)
3. Update `config.md`, `faq.md` and `pull_request_template.md` (#190, #191, #200)
4. Update the `article` page (#133)

Contributors

A total of 14 developers contributed to this release.

Thank @imAzhou, @triple-Mu, @RangeKing, @PeterH0323, @xin-li-67, @Nioolek, @kitecats, @Bin-ze, @JiayuXu0, @cydiachen, @zhiqwang, @Zheng-LinXiao, @hhaAndroid, @wanghonglie

52.8 v0.1.129/9/2022)

Based on MMDetection's RTMDet high precision and low latency object detection algorithm, we have also released RTMDet and provided a Chinese document on the principle and implementation of RTMDet.

52.8.1 Highlights

1. Support RTMDet
2. Support for backbone customization plugins and update How-to documentation (#75)

52.8.2 Bug Fixes

1. Fix some documentation errors (#66, #72, #76, #83, #86)
2. Fix checkpoints link error (#63)
3. Fix the bug that the output of `LetterResize` does not meet the expectation when using `imscale` (#105)

52.8.3 Improvements

1. Reducing the size of docker images (#67)
2. Simplifying `Compose` Logic in `BaseMixImageTransform` (#71)
3. Supports dump results in `test.py` (#84)

Contributors

A total of 13 developers contributed to this release.

Thank @wanghonglie, @hhaAndroid, @yang-0201, @PeterH0323, @RangeKing, @satuoqaq, @Zheng-LinXiao, @xin-li-67, @suibe-qingtian, @MambaWong, @MichaelCai0912, @rimoire, @Nioolek

52.9 v0.1.021/9/2022)

We have released MMYOLO open source library, which is based on MMEngine, MMCV 2.x and MMDetection 3.x libraries. At present, the object detection has been realized, and it will be expanded to multi-task in the future.

52.9.1 Highlights

1. Support YOLOv5/YOLOX training, support YOLOv6 inference. Deployment will be supported soon.
2. Refactored YOLOX from MMDetection to accelerate training and inference.
3. Detailed introduction and advanced tutorials are provided, see the English tutorial.

CHAPTER

FIFTYTHREE

COMPATIBILITY OF MMYOLO

53.1 MMYOLO 0.3.0

53.1.1 METAINFO modification

To unify with other OpenMMLab repositories, change all keys of `METAINFO` in Dataset from upper case to lower case.

53.1.2 About the order of image shape

In OpenMMLab 2.0, to be consistent with the input argument of OpenCV, the argument about image shape in the data transformation pipeline is always in the `(width, height)` order. On the contrary, for computation convenience, the order of the field going through the data pipeline and the model is `(height, width)`. Specifically, in the results processed by each data transform pipeline, the fields and their value meaning is as below:

- img_shape: (height, width)
- ori_shape: (height, width)
- pad_shape: (height, width)
- batch_input_shape: (height, width)

As an example, the initialization arguments of `Mosaic` are as below:

```
@TRANSFORMS.register_module()
class Mosaic(BaseTransform):
    def __init__(self,
                 img_scale: Tuple[int, int] = (640, 640),
                 center_ratio_range: Tuple[float, float] = (0.5, 1.5),
                 bbox_clip_border: bool = True,
                 pad_val: float = 114.0,
                 prob: float = 1.0) -> None:
        ...

        # img_scale order should be (width, height)
        self.img_scale = img_scale

    def transform(self, results: dict) -> dict:
        ...

        results['img'] = mosaic_img
```

(continues on next page)

(continued from previous page)

```
        # (height, width)
        results['img_shape'] = mosaic_img.shape[:2]
```

CHAPTER

FIFTYFOUR

CONVENTIONS

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

54.1 About the order of image shape

In OpenMMLab 2.0, to be consistent with the input argument of OpenCV, the argument about image shape in the data transformation pipeline is always in the `(width, height)` order. On the contrary, for computation convenience, the order of the field going through the data pipeline and the model is `(height, width)`. Specifically, in the results processed by each data transform pipeline, the fields and their value meaning is as below:

- img_shape: (height, width)
- ori_shape: (height, width)
- pad_shape: (height, width)
- batch_input_shape: (height, width)

As an example, the initialization arguments of `Mosaic` are as below:

```
@TRANSFORMS.register_module()
class Mosaic(BaseTransform):
    def __init__(self,
                 img_scale: Tuple[int, int] = (640, 640),
                 center_ratio_range: Tuple[float, float] = (0.5, 1.5),
                 bbox_clip_border: bool = True,
                 pad_val: float = 114.0,
                 prob: float = 1.0) -> None:
        ...

        # img_scale order should be (width, height)
        self.img_scale = img_scale

    def transform(self, results: dict) -> dict:
        ...

        results['img'] = mosaic_img
        # (height, width)
        results['img_shape'] = mosaic_img.shape[:2]
```

CHAPTER

FIFTYFIVE

CODE STYLE

Coming soon. Please refer to chinese documentation.

CHAPTER

FIFTYSIX

MMYOLO.DATASETS

56.1 datasets

class mmyolo.datasets.**BatchShapePolicy**(*batch_size: int = 32*, *img_size: int = 640*, *size_divisor: int = 32*, *extra_pad_ratio: float = 0.5*)

BatchShapePolicy is only used in the testing phase, which can reduce the number of pad pixels during batch inference.

Parameters

- **batch_size** (*int*) – Single GPU batch size during batch inference. Defaults to 32.
- **img_size** (*int*) – Expected output image size. Defaults to 640.
- **size_divisor** (*int*) – The minimum size that is divisible by size_divisor. Defaults to 32.
- **extra_pad_ratio** (*float*) – Extra pad ratio. Defaults to 0.5.

class mmyolo.datasets.**YOLOv5CocoDataset**(**args*, *batch_shapes_cfg: Optional[dict] = None*, ***kwargs*)

Dataset for YOLOv5 COCO Dataset.

We only add *BatchShapePolicy* function compared with CocoDataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

class mmyolo.datasets.**YOLOv5CrowdHumanDataset**(**args*, *batch_shapes_cfg: Optional[dict] = None*, ***kwargs*)

Dataset for YOLOv5 CrowdHuman Dataset.

We only add *BatchShapePolicy* function compared with CrowdHumanDataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

class mmyolo.datasets.**YOLOv5DOTADataset**(**args*, ***kwargs*)

Dataset for YOLOv5 DOTA Dataset.

We only add *BatchShapePolicy* function compared with DOTADataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

class mmyolo.datasets.**YOLOv5VOCDataset**(**args*, *batch_shapes_cfg: Optional[dict] = None*, ***kwargs*)

Dataset for YOLOv5 VOC Dataset.

We only add *BatchShapePolicy* function compared with VOCDataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

mmyolo.datasets.**yolov5_collate**(*data_batch: Sequence*, *use_ms_training: bool = False*) → dict

Rewrite collate_fn to get faster training speed.

Parameters

- **data_batch** (*Sequence*) – Batch of data.

- **use_ms_training** (*bool*) – Whether to use multi-scale training.

56.2 transforms

class mmyolo.datasets.transforms.**FilterAnnotations**(*by_keypoints: bool = False*, ***kwargs*)

Filter invalid annotations.

In addition to the conditions checked by `FilterDetAnnotations`, this filter adds a new condition requiring instances to have at least one visible keypoints.

class mmyolo.datasets.transforms.**LetterResize**(*scale: Union[int, Tuple[int, int]]*, *pad_val: dict = {'img': 0, 'mask': 0, 'seg': 255}*, *use_mini_pad: bool = False*, *stretch_only: bool = False*, *allow_scale_up: bool = True*, *half_pad_param: bool = False*, ***kwargs*)

Resize and pad image while meeting stride-multiple constraints.

Required Keys:

- img (np.uint8)
- batch_shape (np.int64) (optional)

Modified Keys:

- img (np.uint8)
- img_shape (tuple)
- gt_bboxes (optional)

Added Keys: - pad_param (np.float32)

Parameters

- **scale** (*Union[int, Tuple[int, int]]*) – Images scales for resizing.
- **pad_val** (*dict*) – Padding value. Defaults to dict(img=0, seg=255).
- **use_mini_pad** (*bool*) – Whether using minimum rectangle padding. Defaults to True
- **stretch_only** (*bool*) – Whether stretch to the specified size directly. Defaults to False
- **allow_scale_up** (*bool*) – Allow scale up when ratio > 1. Defaults to True
- **half_pad_param** (*bool*) – If set to True, left and right pad_param will be given by dividing padding_h by 2. If set to False, pad_param is in int format. We recommend setting this to False for object detection tasks, and True for instance segmentation tasks. Default to False.

transform(*results: dict*) → dict

Transform function to resize images, bounding boxes, semantic segmentation map and keypoints.

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

Returns Resized results, 'img', 'gt_bboxes', 'gt_seg_map', 'gt_keypoints', 'scale', 'scale_factor', 'img_shape', and 'keep_ratio' keys are updated in result dict.

Return type dict

class mmyolo.datasets.transforms.**LoadAnnotations**(*mask2bbox: bool = False*, *poly2mask: bool = False*, *merge_polygons: bool = True*, ***kwargs*)

Because the yolo series does not need to consider ignore bboxes for the time being, in order to speed up the pipeline, it can be excluded in advance.

Parameters

- **mask2bbox** (*bool*) – Whether to use mask annotation to get bbox. Defaults to False.
- **poly2mask** (*bool*) – Whether to transform the polygons to bitmaps. Defaults to False.
- **merge_polygons** (*bool*) – Whether to merge polygons into one polygon. If merged, the storage structure is simpler and training is more effcient, especially if the mask inside a bbox is divided into multiple polygons. Defaults to True.

merge_multi_segment(*gt_masks: List[numpy.ndarray]*) → List[numpy.ndarray]
Merge multi segments to one list.

Find the coordinates with min distance between each segment, then connect these coordinates with one thin line to merge all segments into one. :param gt_masks: original segmentations in coco's json file.

like [segmentation1, segmentation2,...], each segmentation is a list of coordinates.

Returns merged gt_masks

Return type gt_masks(List(np.array))

min_index(*arr1: numpy.ndarray*, *arr2: numpy.ndarray*) → Tuple[int, int]
Find a pair of indexes with the shortest distance.

Parameters

- **arr1** – (N, 2).
- **arr2** – (M, 2).

Returns a pair of indexes.

Return type tuple

transform(*results: dict*) → dict
Function to load multiple types annotations.

Parameters **results** (*dict*) – Result dict from :obj:mmengine.BaseDataset.

Returns The dict contains loaded bounding box, label and semantic segmentation.

Return type dict

class mmyolo.datasets.transforms.**Mosaic**(*img_scale: Tuple[int, int] = (640, 640)*, *center_ratio_range: Tuple[float, float] = (0.5, 1.5)*, *bbox_clip_border: bool = True*, *pad_val: float = 114.0*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 40*, *random_pop: bool = True*, *max_refetch: int = 15*)
Mosaic augmentation.

Given 4 images, mosaic transform combines them into one output image. The output image is composed of the parts from each sub- image.

```
                mosaic transform
                   center_x
          +------------------------------+
          |       pad        |           |
          |      +-----------+    pad    |
          |      |           |           |
          |      |  image1   +-----------+
          |      |           |           |
          |      |           |   image2  |
```

(continues on next page)

(continued from previous page)

```
center_y   |----+-+-----------+-----------+
           |    |   cropped   |           |
           |pad |   image3    |   image4  |
           |    |             |           |
           +----|-------------+-----------+
                |             |
                +-------------+

The mosaic transform steps are as follows:

    1. Choose the mosaic center as the intersections of 4 images
    2. Get the left top image according to the index, and randomly
       sample another 3 images from the custom dataset.
    3. Sub image will be cropped if image is larger than mosaic patch
```

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

Parameters

- **img_scale** (*Sequence[int]*) – Image size after mosaic pipeline of single image. The shape order should be (width, height). Defaults to (640, 640).
- **center_ratio_range** (*Sequence[float]*) – Center ratio range of mosaic output. Defaults to (0.5, 1.5).
- **bbox_clip_border** (*bool, optional*) – Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **pad_val** (*int*) – Pad value. Defaults to 114.
- **pre_transform** (*Sequence[dict]*) – Sequence of transform object or config dict to be composed.
- **prob** (*float*) – Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) – Whether to use cache. Defaults to False.

- **max_cached_images** (*int*) – The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 10 caches for each image suffices for randomness. Defaults to 40.
- **random_pop** (*bool*) – Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) – The maximum number of retry iterations for getting valid results from the pipeline. If the number of iterations is greater than *max_refetch*, but results is still None, then the iteration is terminated and raise the error. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → list
Call function to collect indexes.

Parameters **dataset** (Dataset or list) – The dataset or cached list.

Returns indexes.

Return type list

mix_img_transform(*results: dict*) → dict
Mixed image data transformation.

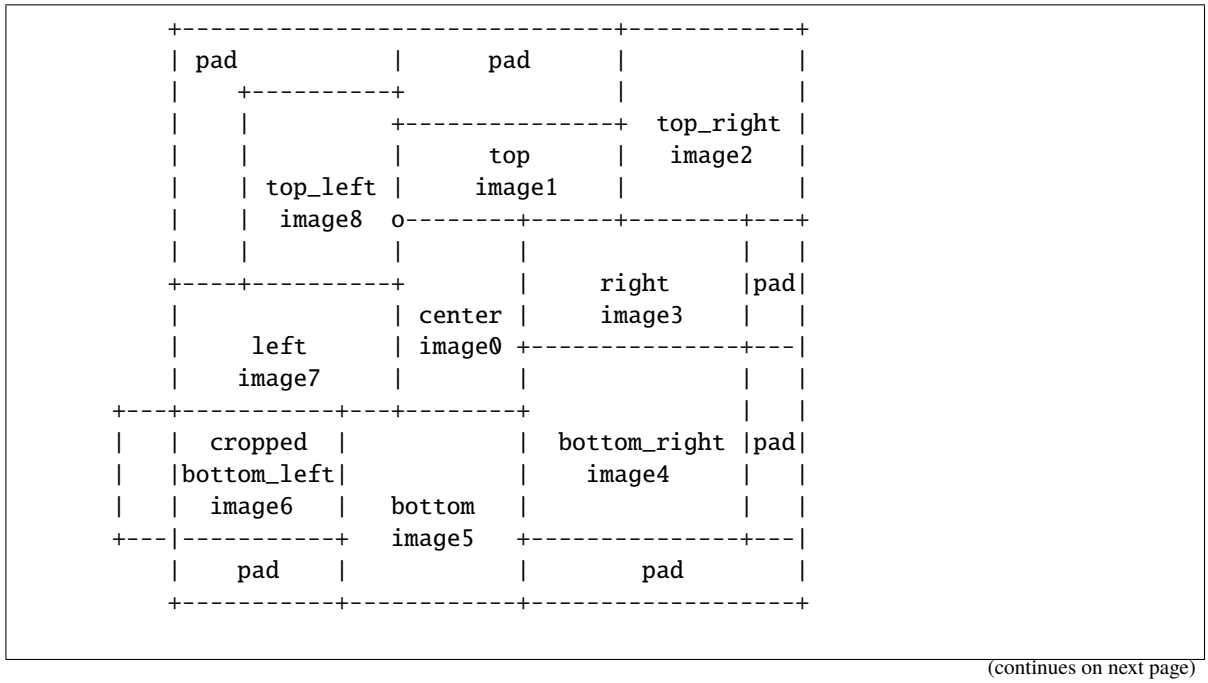
Parameters **results** (*dict*) – Result dict.

Returns Updated result dict.

Return type results (dict)

class mmyolo.datasets.transforms.**Mosaic9**(*img_scale: Tuple[int, int] = (640, 640)*, *bbox_clip_border: bool = True*, *pad_val: Union[float, int] = 114.0*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 50*, *random_pop: bool = True*, *max_refetch: int = 15*)
Mosaic9 augmentation.

Given 9 images, mosaic transform combines them into one output image. The output image is composed of the parts from each sub- image.



(continues on next page)

(continued from previous page)

```
The mosaic transform steps are as follows:

    1. Get the center image according to the index, and randomly
       sample another 8 images from the custom dataset.
    2. Randomly offset the image after Mosaic
```

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

Parameters

- **img_scale** (*Sequence[int]*) – Image size after mosaic pipeline of single image. The shape order should be (width, height). Defaults to (640, 640).
- **bbox_clip_border** (*bool, optional*) – Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **pad_val** (*int*) – Pad value. Defaults to 114.
- **pre_transform** (*Sequence[dict]*) – Sequence of transform object or config dict to be composed.
- **prob** (*float*) – Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) – Whether to use cache. Defaults to False.
- **max_cached_images** (*int*) – The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 5 caches for each image suffices for randomness. Defaults to 50.
- **random_pop** (*bool*) – Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) – The maximum number of retry iterations for getting valid results from the pipeline. If the number of iterations is greater than *max_refetch*, but results is still None, then the iteration is terminated and raise the error. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → list

Call function to collect indexes.

Parameters **dataset** (Dataset or list) – The dataset or cached list.

Returns indexes.

Return type list

mix_img_transform(*results: dict*) → dict
Mixed image data transformation.

Parameters **results** (*dict*) – Result dict.

Returns Updated result dict.

Return type results (dict)

class mmyolo.datasets.transforms.**PPYOLOERandomCrop**(*aspect_ratio: List[float] = [0.5, 2.0]*, *thresholds: List[float] = [0.0, 0.1, 0.3, 0.5, 0.7, 0.9]*, *scaling: List[float] = [0.3, 1.0]*, *num_attempts: int = 50*, *allow_no_crop: bool = True*, *cover_all_box: bool = False*)
Random crop the img and bboxes. Different thresholds are used in PPYOLOE to judge whether the clipped image meets the requirements. This implementation is different from the implementation of RandomCrop in mmdet.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

Added Keys: - pad_param (np.float32)

Parameters

- **aspect_ratio** (*List[float]*) – Aspect ratio of cropped region. Default to [.5, 2].
- **thresholds** (*List[float]*) – Iou thresholds for deciding a valid bbox crop in [min, max] format. Defaults to [.0, .1, .3, .5, .7, .9].
- **scaling** (*List[float]*) – Ratio between a cropped region and the original image in [min, max] format. Default to [.3, 1.].
- **num_attempts** (*int*) – Number of tries for each threshold before giving up. Default to 50.
- **allow_no_crop** (*bool*) – Allow return without actually cropping them. Default to True.
- **cover_all_box** (*bool*) – Ensure all bboxes are covered in the final crop. Default to False.

class mmyolo.datasets.transforms.**PPYOLOERandomDistort**(*hue_cfg: dict = {'max': 18, 'min': - 18, 'prob': 0.5}*, *saturation_cfg: dict = {'max': 1.5, 'min': 0.5, 'prob': 0.5}*, *contrast_cfg: dict = {'max': 1.5, 'min': 0.5, 'prob': 0.5}*, *brightness_cfg: dict = {'max': 1.5, 'min': 0.5, 'prob': 0.5}*, *num_distort_func: int = 4*)

Random hue, saturation, contrast and brightness distortion.

Required Keys:

- img

Modified Keys:

- img (np.float32)

Parameters

- **hue_cfg** (*dict*) – Hue settings. Defaults to dict(min=-18, max=18, prob=0.5).
- **saturation_cfg** (*dict*) – Saturation settings. Defaults to dict(min=0.5, max=1.5, prob=0.5).
- **contrast_cfg** (*dict*) – Contrast settings. Defaults to dict(min=0.5, max=1.5, prob=0.5).
- **brightness_cfg** (*dict*) – Brightness settings. Defaults to dict(min=0.5, max=1.5, prob=0.5).
- **num_distort_func** (*int*) – The number of distort function. Defaults to 4.

transform(*results: dict*) → dict

The hue, saturation, contrast and brightness distortion function.

Parameters **results** (*dict*) – The result dict.

Returns The result dict.

Return type dict

transform_brightness(*results*)

Transform brightness randomly.

transform_contrast(*results*)

Transform contrast randomly.

transform_hue(*results*)

Transform hue randomly.

transform_saturation(*results*)

Transform saturation randomly.

class mmyolo.datasets.transforms.**PackDetInputs**(*meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'scale_factor', 'flip', 'flip_direction')*)

Pack the inputs data for the detection / semantic segmentation / panoptic segmentation.

Compared to mmdet, we just add the *gt_panoptic_seg* field and logic.

transform(*results: dict*) → dict

Method to pack the input data. :param results: Result dict from the data pipeline. :type results: dict

Returns

- 'inputs' (obj:*torch.Tensor*): The forward data of models.
- **'data_sample' (obj:*DetDataSample*): The annotation info of the** sample.

Return type dict

class mmyolo.datasets.transforms.**Polygon2Mask**(*downsample_ratio: int = 4*, *mask_overlap: bool = True*, *coco_style: bool = False*)

Polygons to bitmaps in YOLOv5.

Parameters

- **downsample_ratio** (*int*) – Downsample ratio of mask.
- **mask_overlap** (*bool*) – Whether to use maskoverlap in mask process. When set to True, the implementation here is the same as the official, with higher training speed. If set to True, all gt masks will compress into one overlap mask, the value of mask indicates the index of gt masks. If set to False, one mask is a binary mask. Default to True.
- **coco_style** (*bool*) – Whether to use coco_style to convert the polygons to bitmaps. Note that this option is only used to test if there is an improvement in training speed and we recommend setting it to False.

polygon2mask(*img_shape: Tuple[int, int]*, *polygons: numpy.ndarray*, *color: int = 1*) → numpy.ndarray

Parameters

- **img_shape** (*tuple*) – The image size.
- **polygons** (*np.ndarray*) – [N, M], N is the number of polygons, M is the number of points(Be divided by 2).
- **color** (*int*) – color in fillPoly.

Returns the overlap mask.

Return type np.ndarray

polygons2masks(*img_shape: Tuple[int, int]*, *polygons: mmdet.structures.mask.structures.PolygonMasks*, *color: int = 1*) → numpy.ndarray

Return a list of bitmap masks.

Parameters

- **img_shape** (*tuple*) – The image size.
- **polygons** (*PolygonMasks*) – The mask annotations.
- **color** (*int*) – color in fillPoly.

Returns the list of masks in bitmaps.

Return type List[np.ndarray]

polygons2masks_overlap(*img_shape: Tuple[int, int]*, *polygons: mmdet.structures.mask.structures.PolygonMasks*) → Tuple[numpy.ndarray, numpy.ndarray]

Return a overlap mask and the sorted idx of area.

Parameters

- **img_shape** (*tuple*) – The image size.
- **polygons** (*PolygonMasks*) – The mask annotations.
- **color** (*int*) – color in fillPoly.

Returns the overlap mask and the sorted idx of area.

Return type Tuple[np.ndarray, np.ndarray]

transform(*results: dict*) → dict
The transform function. All subclass of BaseTransform should override this method.

This function takes the result dict as the input, and can add new items to the dict or modify existing items in the dict. And the result dict will be returned in the end, which allows to concate multiple transforms into a pipeline.

Parameters **results** (*dict*) – The result dict.

Returns The result dict.

Return type dict

class mmyolo.datasets.transforms.**RandomAffine**(***kwargs*)

class mmyolo.datasets.transforms.**RandomFlip**(*prob: Optional[Union[float, Iterable[float]]] = None*, *direction: Union[str, Sequence[Optional[str]]] = 'horizontal'*, *swap_seg_labels: Optional[Sequence] = None*)

class mmyolo.datasets.transforms.**RegularizeRotatedBox**(*angle_version='le90'*)
Regularize rotated boxes.

Due to the angle periodicity, one rotated box can be represented in many different (x, y, w, h, t). To make each rotated box unique, `regularize_boxes` will take the remainder of the angle divided by 180 degrees.

For convenience, three angle_version can be used here:

- **'oc': OpenCV Definition. Has the same box representation as** `cv2.minAreaRect` the angle ranges in [-90, 0).
- **'le90': Long Edge Definition (90). the angle ranges in [-90, 90).** The width is always longer than the height.
- **'le135': Long Edge Definition (135). the angle ranges in [-45, 135).** The width is always longer than the height.

Required Keys:

- gt_bboxes (RotatedBoxes[torch.float32])

Modified Keys:

- gt_bboxes

Parameters **angle_version** (*str*) – Angle version. Can only be 'oc', 'le90', or 'le135'. Defaults to 'le90.

transform(*results: dict*) → dict
The transform function. All subclass of BaseTransform should override this method.

This function takes the result dict as the input, and can add new items to the dict or modify existing items in the dict. And the result dict will be returned in the end, which allows to concate multiple transforms into a pipeline.

Parameters **results** (*dict*) – The result dict.

Returns The result dict.

Return type dict

class mmyolo.datasets.transforms.**RemoveDataElement**(*keys: Union[str, Sequence[str]]*)
Remove unnecessary data element in results.

Parameters **keys** (*Union[str, Sequence[str]]*) – Keys need to be removed.

transform(*results: dict*) → dict

The transform function. All subclass of BaseTransform should override this method.

This function takes the result dict as the input, and can add new items to the dict or modify existing items in the dict. And the result dict will be returned in the end, which allows to concate multiple transforms into a pipeline.

Parameters **results** (*dict*) – The result dict.

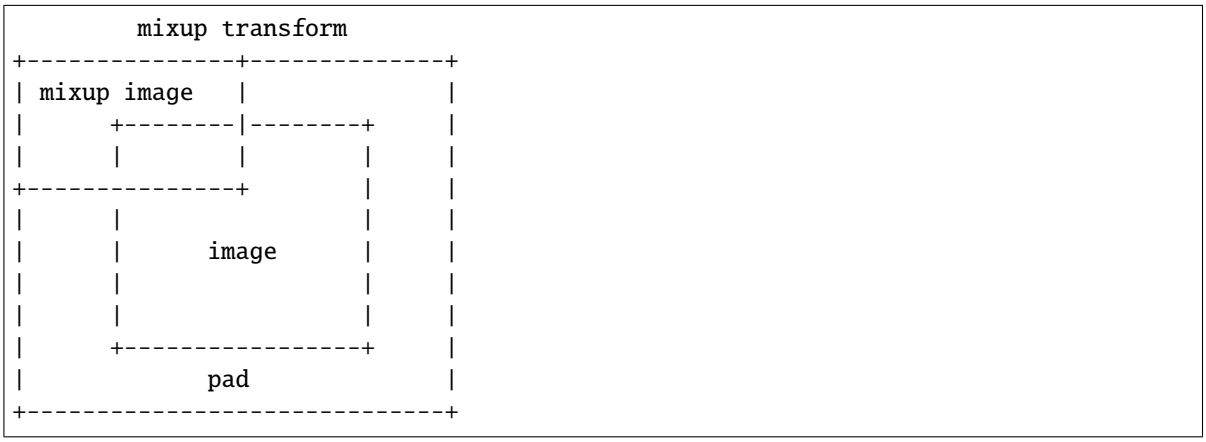
Returns The result dict.

Return type dict

class mmyolo.datasets.transforms.**Resize**(*scale: Optional[Union[int, Tuple[int, int]]] = None*, *scale_factor: Optional[Union[float, Tuple[float, float]]] = None*, *keep_ratio: bool = False*, *clip_object_border: bool = True*, *backend: str = 'cv2'*, *interpolation='bilinear'*)

class mmyolo.datasets.transforms.**YOLOXMixUp**(*img_scale: Tuple[int, int] = (640, 640)*, *ratio_range: Tuple[float, float] = (0.5, 1.5)*, *flip_ratio: float = 0.5*, *pad_val: float = 114.0*, *bbox_clip_border: bool = True*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 20*, *random_pop: bool = True*, *max_refetch: int = 15*)

MixUp data augmentation for YOLOX.



The mixup transform steps are as follows:

1. Another random image is picked by dataset and embedded in the top left patch(after padding and resizing)
2. The target of mixup transform is the weighted average of mixup image and origin image.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img

- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

Parameters

- **img_scale** (*Sequence[int]*) – Image output size after mixup pipeline. The shape order should be (width, height). Defaults to (640, 640).
- **ratio_range** (*Sequence[float]*) – Scale ratio of mixup image. Defaults to (0.5, 1.5).
- **flip_ratio** (*float*) – Horizontal flip ratio of mixup image. Defaults to 0.5.
- **pad_val** (*int*) – Pad value. Defaults to 114.
- **bbox_clip_border** (*bool, optional*) – Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **pre_transform** (*Sequence[dict]*) – Sequence of transform object or config dict to be composed.
- **prob** (*float*) – Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) – Whether to use cache. Defaults to False.
- **max_cached_images** (*int*) – The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 10 caches for each image suffices for randomness. Defaults to 20.
- **random_pop** (*bool*) – Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) – The maximum number of iterations. If the number of iterations is greater than *max_refetch*, but gt_bbox is still empty, then the iteration is terminated. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → int
Call function to collect indexes.

Parameters **dataset** (Dataset or list) – The dataset or cached list.

Returns indexes.

Return type int

mix_img_transform(*results: dict*) → dict
YOLOX MixUp transform function.

Parameters **results** (*dict*) – Result dict.

Returns Updated result dict.

Return type results (dict)

class mmyolo.datasets.transforms.**YOLOv5CopyPaste**(*ioa_thresh: float = 0.3*, *prob: float = 0.5*)
Copy-Paste used in YOLOv5 and YOLOv8.

This transform randomly copy some objects in the image to the mirror position of the image.It is different from the *CopyPaste* in mmdet.

Required Keys:

- img (np.uint8)
- gt_bboxes (BaseBoxes[torch.float32])
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- gt_masks (PolygonMasks) (optional)

Modified Keys:

- img
- gt_bboxes
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (optional)
- gt_masks (optional)

Parameters

- **ioa_thresh** (*float*) – Ioa thresholds for deciding valid bbox.
- **prob** (*float*) – Probability of choosing objects. Defaults to 0.5.

static bbox_ioa(*gt_bboxes_flip: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*, *gt_bboxes: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*, *eps: float = 1e-07*) → numpy.ndarray

Calculate ioa between gt_bboxes_flip and gt_bboxes.

Parameters

- **gt_bboxes_flip** (*HorizontalBoxes*) – Flipped ground truth bounding boxes.
- **gt_bboxes** (*HorizontalBoxes*) – Ground truth bounding boxes.
- **eps** (*float*) – Default to 1e-10.

Returns Ioa.

Return type (Tensor)

class mmyolo.datasets.transforms.**YOLOv5HSVRandomAug**(*hue_delta: Union[int, float] = 0.015*, *saturation_delta: Union[int, float] = 0.7*, *value_delta: Union[int, float] = 0.4*)

Apply HSV augmentation to image sequentially.

Required Keys:

- img

Modified Keys:

- img

Parameters

- **hue_delta** (*[int, float]*) – delta of hue. Defaults to 0.015.
- **saturation_delta** (*[int, float]*) – delta of saturation. Defaults to 0.7.
- **value_delta** (*[int, float]*) – delta of value. Defaults to 0.4.

transform(*results: dict*) → dict
The HSV augmentation transform function.

Parameters **results** (*dict*) – The result dict.

Returns The result dict.

Return type dict

class mmyolo.datasets.transforms.**YOLOv5KeepRatioResize**(*scale: Union[int, Tuple[int, int]]*, *keep_ratio: bool = True*, ***kwargs*)

Resize images & bbox(if existed).

This transform resizes the input image according to `scale`. Bboxes (if existed) are then resized with the same scale factor.

Required Keys:

- img (np.uint8)
- gt_bboxes (BaseBoxes[torch.float32]) (optional)

Modified Keys:

- img (np.uint8)
- img_shape (tuple)
- gt_bboxes (optional)
- scale (float)

Added Keys:

- scale_factor (np.float32)

Parameters **scale** (*Union[int, Tuple[int, int]]*) – Images scales for resizing.

class mmyolo.datasets.transforms.**YOLOv5MixUp**(*alpha: float = 32.0*, *beta: float = 32.0*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 20*, *random_pop: bool = True*, *max_refetch: int = 15*)

MixUp data augmentation for YOLOv5.

The mixup transform steps are as follows:

1. Another random image is picked by dataset.
2. **Randomly obtain the fusion ratio from the beta distribution,** then fuse the target

of the original image and mixup image through this ratio.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img

- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

Parameters

- **alpha** (*float*) – parameter of beta distribution to get mixup ratio. Defaults to 32.
- **beta** (*float*) – parameter of beta distribution to get mixup ratio. Defaults to 32.
- **pre_transform** (*Sequence[dict]*) – Sequence of transform object or config dict to be composed.
- **prob** (*float*) – Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) – Whether to use cache. Defaults to False.
- **max_cached_images** (*int*) – The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 10 caches for each image suffices for randomness. Defaults to 20.
- **random_pop** (*bool*) – Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) – The maximum number of iterations. If the number of iterations is greater than *max_refetch*, but gt_bbox is still empty, then the iteration is terminated. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → int
Call function to collect indexes.

Parameters **dataset** (Dataset or list) – The dataset or cached list.

Returns indexes.

Return type int

mix_img_transform(*results: dict*) → dict
YOLOv5 MixUp transform function.

Parameters **results** (*dict*) – Result dict

Returns Updated result dict.

Return type results (dict)

class mmyolo.datasets.transforms.**YOLOv5RandomAffine**(*max_rotate_degree: float = 10.0*, *max_translate_ratio: float = 0.1*, *scaling_ratio_range: Tuple[float, float] = (0.5, 1.5)*, *max_shear_degree: float = 2.0*, *border: Tuple[int, int] = (0, 0)*, *border_val: Tuple[int, int, int] = (114, 114, 114)*, *bbox_clip_border: bool = True*, *min_bbox_size: int = 2*, *min_area_ratio: float = 0.1*, *use_mask_refine: bool = False*, *max_aspect_ratio: float = 20.0*, *resample_num: int = 1000*)
Random affine transform data augmentation in YOLOv5 and YOLOv8. It is different from the implementation in YOLOX.

This operation randomly generates affine transform matrix which including rotation, translation, shear and scaling transforms. If you set use_mask_refine == True, the code will use the masks annotation to refine the bbox. Our implementation is slightly different from the official. In COCO dataset, a gt may have multiple mask tags. The official YOLOv5 annotation file already combines the masks that an object has, but our code takes into account the fact that an object has multiple masks.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- gt_masks (PolygonMasks) (optional)

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)
- gt_masks (PolygonMasks) (optional)

Parameters

- **max_rotate_degree** (*float*) – Maximum degrees of rotation transform. Defaults to 10.
- **max_translate_ratio** (*float*) – Maximum ratio of translation. Defaults to 0.1.
- **scaling_ratio_range** (*tuple[float]*) – Min and max ratio of scaling transform. Defaults to (0.5, 1.5).
- **max_shear_degree** (*float*) – Maximum degrees of shear transform. Defaults to 2.
- **border** (*tuple[int]*) – Distance from width and height sides of input image to adjust output shape. Only used in mosaic dataset. Defaults to (0, 0).
- **border_val** (*tuple[int]*) – Border padding values of 3 channels. Defaults to (114, 114, 114).
- **bbox_clip_border** (*bool, optional*) – Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **min_bbox_size** (*float*) – Width and height threshold to filter bboxes. If the height or width of a box is smaller than this value, it will be removed. Defaults to 2.
- **min_area_ratio** (*float*) – Threshold of area ratio between original bboxes and wrapped bboxes. If smaller than this value, the box will be removed. Defaults to 0.1.
- **use_mask_refine** (*bool*) – Whether to refine bbox by mask. Deprecated.
- **max_aspect_ratio** (*float*) – Aspect ratio of width and height threshold to filter bboxes. If max(h/w, w/h) larger than this value, the box will be removed. Defaults to 20.
- **resample_num** (*int*) – Number of poly to resample to.

clip_polygons(*gt_masks: mmdet.structures.mask.structures.PolygonMasks*, *height: int*, *width: int*) → mmdet.structures.mask.structures.PolygonMasks

Function to clip points of polygons with height and width.

Parameters

- **gt_masks** (*PolygonMasks*) – Annotations of instance segmentation.
- **height** (*int*) – height of clip border.
- **width** (*int*) – width of clip border.

Returns Clip annotations of instance segmentation.

Return type clipped_masks (PolygonMasks)

filter_gt_bboxes(*origin_bboxes: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*, *wrapped_bboxes: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*) → torch.Tensor

Filter gt bboxes.

Parameters

- **origin_bboxes** (*HorizontalBoxes*) – Origin bboxes.
- **wrapped_bboxes** (*HorizontalBoxes*) – Wrapped bboxes

Returns The result dict.

Return type dict

resample_masks(*gt_masks: mmdet.structures.mask.structures.PolygonMasks*) → mmdet.structures.mask.structures.PolygonMasks

Function to resample each mask annotation with shape (2 * n,) to shape (resample_num * 2,).

Parameters **gt_masks** (*PolygonMasks*) – Annotations of semantic segmentation.

segment2box(*gt_masks: mmdet.structures.mask.structures.PolygonMasks*, *height: int*, *width: int*) → mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes

Convert 1 segment label to 1 box label, applying inside-image constraint i.e. (xy1, xy2, ...) to (xyxy) :param gt_masks: the segment label :type gt_masks: torch.Tensor :param width: the width of the image. Defaults to 640 :type width: int :param height: The height of the image. Defaults to 640 :type height: int

Returns the clip bboxes from gt_masks.

Return type HorizontalBoxes

warp_mask(*gt_masks: mmdet.structures.mask.structures.PolygonMasks*, *warp_matrix: numpy.ndarray*, *img_w: int*, *img_h: int*) → mmdet.structures.mask.structures.PolygonMasks

Warp masks by warp_matrix and retain masks inside image after warping.

Parameters

- **gt_masks** (*PolygonMasks*) – Annotations of semantic segmentation.
- **warp_matrix** (*np.ndarray*) – Affine transformation matrix. Shape: (3, 3).
- **img_w** (*int*) – Width of output image.
- **img_h** (*int*) – Height of output image.

Returns Masks after warping.

Return type PolygonMasks

static warp_poly(*poly: numpy.ndarray*, *warp_matrix: numpy.ndarray*, *img_w: int*, *img_h: int*) → numpy.ndarray

Function to warp one mask and filter points outside image.

Parameters

- **poly** (*np.ndarray*) – Segmentation annotation with shape (n,) and with format (x1, y1, x2, y2, ...).
- **warp_matrix** (*np.ndarray*) – Affine transformation matrix. Shape: (3, 3).
- **img_w** (*int*) – Width of output image.
- **img_h** (*int*) – Height of output image.

CHAPTER

FIFTYSEVEN

MMYOLO.ENGINE

57.1 hooks

57.2 optimizers

CHAPTER

FIFTYEIGHT

MMYOLO.MODELS

58.1 backbones

class mmyolo.models.backbones.**BaseBackbone**(*arch_setting: list*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Sequence[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *plugins: Optional[Union[dict, List[dict]]] = None*, *norm_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *act_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

BaseBackbone backbone used in YOLO series.

```
Backbone model structure diagram
+-----------+
|   input   |
+-----------+
      v
+-----------+
|   stem    |
|   layer   |
+-----------+
      v
+-----------+
|   stage   |
|  layer 1  |
+-----------+
      v
+-----------+
|   stage   |
|  layer 2  |
+-----------+
      v
    ......
      v
+-----------+
|   stage   |
```

(continues on next page)

(continued from previous page)

```
|  layer n  |
+-----------+
In P5 model, n=4
In P6 model, n=5
```

Parameters

- **arch_setting** (*list*) – Architecture of BaseBackbone.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains:
  - cfg (dict, required): Cfg dict to build plugin.
  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as 'num_stages'.
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** – Number of input image channels. Defaults to 3.
- **out_indices** (*Sequence[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to None.
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to None.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

abstract build_stage_layer(*stage_idx: int*, *setting: list*)

Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

abstract build_stem_layer()

Build a stem layer.

forward(*x: torch.Tensor*) → tuple

Forward batch_inputs from the data_preprocessor.

make_stage_plugins(*plugins*, *stage_idx*, *setting*)

Make plugins for backbone `stage_idx` th stage.

Currently we support to insert `context_block`, `empirical_attention_block`, `nonlocal_block`, `dropout_block` into the backbone.

An example of plugins format could be:

Examples

```
>>> plugins=[
...     dict(cfg=dict(type='xxx', arg1='xxx'),
...          stages=(False, True, True, True)),
...     dict(cfg=dict(type='yyy'),
...          stages=(True, True, True, True)),
... ]
>>> model = YOLOv5CSPDarknet()
>>> stage_plugins = model.make_stage_plugins(plugins, 0, setting)
>>> assert len(stage_plugins) == 1
```

Suppose `stage_idx=0`, the structure of blocks in the stage would be:

```
conv1 -> conv2 -> conv3 -> yyy
```

Suppose `stage_idx=1`, the structure of blocks in the stage would be:

```
conv1 -> conv2 -> conv3 -> xxx -> yyy
```

Parameters

- **plugins** (*list[dict]*) – List of plugins cfg to build. The postfix is required if multiple same type plugins are inserted.
- **stage_idx** (*int*) – Index of stage to build If stages is missing, the plugin would be applied to all stages.
- **setting** (*list*) – The architecture setting of a stage layer.

Returns Plugins for current stage

Return type list[nn.Module]

train(*mode: bool = True*)

Convert the model into training mode while keep normalization layer frozen.

class mmyolo.models.backbones.**CSPNeXt**(*arch: str = 'P5', deepen_factor: float = 1.0, widen_factor: float = 1.0, input_channels: int = 3, out_indices: Sequence[int] = (2, 3, 4), frozen_stages: int = - 1, plugins: Optional[Union[dict, List[dict]]] = None, use_depthwise: bool = False, expand_ratio: float = 0.5, arch_ovewrite: Optional[dict] = None, channel_attention: bool = True, conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None, norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}, act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}, norm_eval: bool = False, init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = {'a': 2.23606797749979, 'distribution': 'uniform', 'layer': 'Conv2d', 'mode': 'fan_in', 'nonlinearity': 'leaky_relu', 'type': 'Kaiming'}*)

CSPNeXt backbone used in RTMDet.

Parameters

- **arch** (*str*) – Architecture of CSPNeXt, from {P5, P6}. Defaults to P5.

- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **out_indices** (*Sequence[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin.Defaults to - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as 'num_stages'.
- **use_depthwise** (*bool*) – Whether to use depthwise separable convolution. Defaults to False.
- **expand_ratio** (*float*) – Ratio to adjust the number of channels of the hidden layer. Defaults to 0.5.
- **arch_ovewrite** (*list*) – Overwrite default arch settings. Defaults to None.
- **channel_attention** (*bool*) – Whether to add channel attention in each stage. Defaults to True.
- **conv_cfg** (ConfigDict or dict, optional) – Config dict for convolution layer. Defaults to None.
- **norm_cfg** (ConfigDict or dict) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (ConfigDict or dict) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict]): Initialization config dict.

build_stage_layer(*stage_idx: int*, *setting: list*) → list
Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module
Build a stem layer.

class mmyolo.models.backbones.**PPYOLOECSPResNet**(*arch: str = 'P5'*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *plugins: Optional[Union[dict, List[dict]]] = None*, *arch_ovewrite: Optional[dict] = None*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'shortcut': True, 'type': 'PPYOLOEBasicBlock', 'use_alpha': True}*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *attention_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'act_cfg': {'type': 'HSigmoid'}, 'type': 'EffectiveSELayer'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, *use_large_stem: bool = False*)

CSP-ResNet backbone used in PPYOLOE.

Parameters

- **arch** (*str*) – Architecture of CSPNeXt, from {P5, P6}. Defaults to P5.
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **out_indices** (*Sequence[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as 'num_stages'.
- **arch_ovewrite** (*list*) – Overwrite default arch settings. Defaults to None.
- **block_cfg** (*dict*) – Config dict for block. Defaults to dict(type='PPYOLOEBasicBlock', shortcut=True, use_alpha=True)
- **norm_cfg** (ConfigDict or dict) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', momentum=0.1, eps=1e-5).
- **act_cfg** (ConfigDict or dict) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **attention_cfg** (*dict*) – Config dict for *EffectiveSELayer*. Defaults to dict(type='EffectiveSELayer', act_cfg=dict(type='HSigmoid')).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict]): Initialization config dict. :param use_large_stem: Whether to use large stem layer.

Defaults to False.

build_stage_layer(*stage_idx: int*, *setting: list*) → list
Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module
Build a stem layer.

class mmyolo.models.backbones.**YOLOXCSPDarknet**(*arch: str = 'P5'*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = -1*, *use_depthwise: bool = False*, *spp_kernal_sizes: Tuple[int] = (5, 9, 13)*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSP-Darknet backbone used in YOLOX.

Parameters

- **arch** (*str*) – Architecture of CSP-Darknet, from {P5, P6}. Defaults to P5.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains:
  - cfg (dict, required): Cfg dict to build plugin.
  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as 'num_stages'.
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) – Number of input image channels. Defaults to 3.
- **out_indices** (*Tuple[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **use_depthwise** (*bool*) – Whether to use depthwise separable convolution. Defaults to False.
- **spp_kernal_sizes** – (tuple[int]): Sequential of kernel sizes of SPP layers. Defaults to (5, 9, 13).
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).

- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.
- **init_cfg** (*Union[dict,list[dict]], optional*) – Initialization config dict. Defaults to None.

Example

```
>>> from mmyolo.models import YOLOXCSPDarknet
>>> import torch
>>> model = YOLOXCSPDarknet()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list
Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module
Build a stem layer.

class mmyolo.models.backbones.**YOLOv5CSPDarknet**(*arch: str = 'P5'*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = -1*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSP-Darknet backbone used in YOLOv5. :param arch: Architecture of CSP-Darknet, from {P5, P6}.

Defaults to P5.

Parameters

- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

should be same as 'num_stages'.

- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) – Number of input image channels. Defaults to: 3.
- **out_indices** (*Tuple[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **init_cfg** (*Union[dict,list[dict]], optional*) – Initialization config dict. Defaults to None.

Example

```
>>> from mmyolo.models import YOLOv5CSPDarknet
>>> import torch
>>> model = YOLOv5CSPDarknet()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list
Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module
Build a stem layer.

init_weights()
Initialize the parameters.

class mmyolo.models.backbones.**YOLOv6CSPBep**(*arch: str = 'P5'*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *hidden_ratio: float = 0.5*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *use_cspsppf: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'ConvWrapper'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSPBep backbone used in YOLOv6. :param arch: Architecture of BaseDarknet, from {P5, P6}.

Defaults to P5.

Parameters

- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as 'num_stages'.

- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) – Number of input image channels. Defaults to 3.
- **out_indices** (*Tuple[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='LeakyReLU', negative_slope=0.1).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **block_cfg** (*dict*) – Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **block_act_cfg** (*dict*) – Config dict for activation layer used in each stage. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*Union[dict, list[dict]], optional*) – Initialization config dict. Defaults to None.

Example

```
>>> from mmyolo.models import YOLOv6CSPBep
>>> import torch
>>> model = YOLOv6CSPBep()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list
Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

class mmyolo.models.backbones.**YOLOv6EfficientRep**(*arch: str = 'P5'*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = -1*, *use_cspsppf: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *norm_eval: bool = False*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)
EfficientRep backbone used in YOLOv6. :param arch: Architecture of BaseDarknet, from {P5, P6}.

Defaults to P5.

Parameters

- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as 'num_stages'.
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) – Number of input image channels. Defaults to 3.
- **out_indices** (*Tuple[int]*) – Output from which stages. Defaults to (2, 3, 4).

- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='LeakyReLU', negative_slope=0.1).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **block_cfg** (*dict*) – Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **init_cfg** (*Union[dict, list[dict]], optional*) – Initialization config dict. Defaults to None.

Example

```
>>> from mmyolo.models import YOLOv6EfficientRep
>>> import torch
>>> model = YOLOv6EfficientRep()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list
Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module
Build a stem layer.

init_weights()
Initialize the weights.

class mmyolo.models.backbones.**YOLOv7Backbone**(*arch: str = 'L', deepen_factor: float = 1.0, widen_factor: float = 1.0, input_channels: int = 3, out_indices: Tuple[int] = (2, 3, 4), frozen_stages: int = - 1, plugins: Optional[Union[dict, List[dict]]] = None, norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}, act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}, norm_eval: bool = False, init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Backbone used in YOLOv7.

Parameters

- **arch** (*str*) – Architecture of YOLOv7Defaults to L.
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **out_indices** (*Sequence[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains:
  - cfg (dict, required): Cfg dict to build plugin.
  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as 'num_stages'.
- **norm_cfg** (ConfigDict or dict) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (ConfigDict or dict) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict]): Initialization config dict.

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

class mmyolo.models.backbones.**YOLOv8CSPDarknet**(*arch: str = 'P5'*, *last_stage_out_channels: int = 1024*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSP-Darknet backbone used in YOLOv8.

Parameters

- **arch** (*str*) – Architecture of CSP-Darknet, from {P5}. Defaults to P5.
- **last_stage_out_channels** (*int*) – Final layer output channel. Defaults to 1024.
- **plugins** (*list[dict]*) – List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as 'num_stages'.
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) – Number of input image channels. Defaults to: 3.
- **out_indices** (*Tuple[int]*) – Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **init_cfg** (*Union[dict,list[dict]], optional*) – Initialization config dict. Defaults to None.

Example

```
>>> from mmyolo.models import YOLOv8CSPDarknet
>>> import torch
>>> model = YOLOv8CSPDarknet()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list
Build a stage layer.

Parameters

- **stage_idx** (*int*) – The index of a stage layer.
- **setting** (*list*) – The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module
Build a stem layer.

init_weights()
Initialize the parameters.

58.2 data_preprocessor

58.3 dense_heads

class mmyolo.models.dense_heads.**PPYOLOEHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0.5, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'alpha': 0.75, 'gamma': 2.0, 'iou_weighted': True, 'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.VarifocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xyxy', 'iou_mode': 'giou', 'loss_weight': 2.5, 'reduction': 'mean', 'return_iou': False, 'type': 'IoULoss'}*, *loss_dfl: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 0.125, 'reduction': 'mean', 'type': 'mmdet.DistributionFocalLoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

PPYOLOEHead head used in PPYOLOE. The YOLOv6 head and the PPYOLOE head are only slightly different. Distribution focal loss is extra used in PPYOLOE, but not in YOLOv6.

Parameters

- **head_module** (*ConfigType*) – Base module used for YOLOv5Head
- **prior_generator** (*dict*) – Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) – Config of bbox coder.
- **loss_cls** (ConfigDict or dict) – Config of classification loss.
- **loss_bbox** (ConfigDict or dict) – Config of localization loss.
- **loss_dfl** (ConfigDict or dict) – Config of distribution focal loss.
- **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *bbox_dist_preds: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

Parameters

- **cls_scores** (*Sequence[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **bbox_dist_preds** (*Sequence[Tensor]*) – Box distribution logits for each scale level with shape (bs, reg_max + 1, H*W, 4).
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes `bboxes` and `labels` attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) – Batch of gt_instances_ignore. It includes `bboxes` attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of losses.

Return type dict[str, Tensor]

class mmyolo.models.dense_heads.**PPYOLOEHeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *reg_max: int = 16*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

PPYOLOEHead head module used in `PPYOLOE.

<https://arxiv.org/abs/2203.16250>`_.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) – The number of priors (points) at a point on the feature grid.
- **featmap_strides** (*Sequence[int]*) – Downsample factor of each feature map. Defaults to (8, 16, 32).

- **reg_max** (*int*) – Max value of integral set :math: `{0, ..., reg_max}` in QFL setting. Defaults to 16.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → torch.Tensor
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions.

Return type Tuple[List]

forward_single(*x: torch.Tensor*, *cls_stem: torch.nn.modules.container.ModuleList*, *cls_pred: torch.nn.modules.container.ModuleList*, *reg_stem: torch.nn.modules.container.ModuleList*, *reg_pred: torch.nn.modules.container.ModuleList*) → torch.Tensor
Forward feature of a single scale level.

init_weights(*prior_prob=0.01*)
Initialize the weight and bias of PPYOLOE head.

class mmyolo.models.dense_heads.**RTMDetHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'beta': 2.0, 'loss_weight': 1.0, 'type': 'mmdet.QualityFocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 2.0, 'type': 'mmdet.GIoULoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)
RTMDet head.

Parameters

- **head_module** (*ConfigType*) – Base module used for RTMDetHead
- **prior_generator** – Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) – Config of bbox coder.
- **loss_cls** (ConfigDict or dict) – Config of classification loss.
- **loss_bbox** (ConfigDict or dict) – Config of localization loss.

- **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions, and objectnesses.

Return type Tuple[List]

loss_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: List[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict
Compute losses of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Decoded box for each scale level with shape (N, num_anchors * 4, H, W) in [tl_x, tl_y, br_x, br_y] format.
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], Optional) – Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

special_init()
Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**RTMDetInsSepBNHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'beta': 2.0, 'loss_weight': 1.0, 'type': 'mmdet.QualityFocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 2.0, 'type': 'mmdet.GIoULoss'}*, *loss_mask={'eps': 5e-06, 'loss_weight': 2.0, 'reduction': 'mean', 'type': 'mmdet.DiceLoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

RTMDet Instance Segmentation head.

Parameters

- **head_module** (*ConfigType*) – Base module used for RTMDetInsSepBNHead
- **prior_generator** – Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) – Config of bbox coder.
- **loss_cls** (ConfigDict or dict) – Config of classification loss.
- **loss_bbox** (ConfigDict or dict) – Config of localization loss.
- **loss_mask** (ConfigDict or dict) – Config of mask loss.
- **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: List[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Compute losses of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)

- **bbox_preds** (*list[Tensor]*) – Decoded box for each scale level with shape (N, num_anchors * 4, H, W) in [tl_x, tl_y, br_x, br_y] format.
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes `bboxes` and `labels` attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], Optional) – Batch of gt_instances_ignore. It includes `bboxes` attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

parse_dynamic_params(*flatten_kernels: torch.Tensor*) → tuple

split kernel head prediction to conv weight and bias.

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *kernel_preds: List[torch.Tensor]*, *mask_feats: torch.Tensor*, *score_factors: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted from the head into bbox results.

Note: When score_factors is not None, the cls_scores are usually multiplied by it then obtain the real score used in NMS.

Parameters

- **cls_scores** (*list[Tensor]*) – Classification scores for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **kernel_preds** (*list[Tensor]*) – Kernel predictions of dynamic convs for all scale levels, each is a 4D-tensor, has shape (batch_size, num_params, H, W).
- **mask_feats** (*Tensor*) – Mask prototype features extracted from the mask head, has shape (batch_size, num_prototypes, H, W).
- **score_factors** (*list[Tensor], optional*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, num_priors * 1, H, W). Defaults to None.
- **batch_img_metas** (*list[dict], Optional*) – Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) – Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) – If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Defaults to True.

Returns

Object detection and instance segmentation results of each image after the post process. Each item usually contains following keys.

- scores (Tensor): Classification scores, has a shape (num_instance,)
- labels (Tensor): Labels of bboxes, has a shape (num_instances,).

- bboxes (Tensor): Has a shape (num_instances, 4), the last dimension 4 arrange as (x1, y1, x2, y2).
- masks (Tensor): Has a shape (num_instances, h, w).

Return type list[InstanceData]

class mmyolo.models.dense_heads.**RTMDetInsSepBNHeadModule**(*num_classes: int*, **args*, *num_prototypes: int = 8*, *dyconv_channels: int = 8*, *num_dyconvs: int = 3*, *use_sigmoid_cls: bool = True*, ***kwargs*)

Detection and Instance Segmentation Head of RTMDet.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **num_prototypes** (*int*) – Number of mask prototype features extracted from the mask head. Defaults to 8.
- **dyconv_channels** (*int*) – Channel of the dynamic conv layers. Defaults to 8.
- **num_dyconvs** (*int*) – Number of the dynamic convolution layers. Defaults to 3.
- **use_sigmoid_cls** (*bool*) – Use sigmoid for class prediction. Defaults to True.

forward(*feats: Tuple[torch.Tensor, ...]*) → tuple

Forward features from the upstream network.

Parameters **feats** (*tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns

Usually a tuple of classification scores and bbox prediction - cls_scores (list[Tensor]): Classification scores for all scale

levels, each is a 4D-tensor, the channels number is num_base_priors * num_classes.

- bbox_preds (list[Tensor]): Box energies / deltas for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * 4.
- kernel_preds (list[Tensor]): Dynamic conv kernels for all scale levels, each is a 4D-tensor, the channels number is num_gen_params.
- **mask_feat (Tensor): Mask prototype features.** Has shape (batch_size, num_prototypes, H, W).

Return type tuple

init_weights() → None

Initialize weights of the head.

class mmyolo.models.dense_heads.**RTMDetRotatedHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistanceAnglePointCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'beta': 2.0, 'loss_weight': 1.0, 'type': 'mmdet.QualityFocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 2.0, 'mode': 'linear', 'type': 'mmrotate.RotatedIoULoss'}*, *angle_version: str = 'le90'*, *use_hbbox_loss: bool = False*, *angle_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'mmrotate.PseudoAngleCoder'}*, *loss_angle: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

RTMDet-R head.

Compared with RTMDetHead, RTMDetRotatedHead add some args to support rotated object detection.

- *angle_version* used to limit angle_range during training.
- *angle_coder* used to encode and decode angle, which is similar to bbox_coder.
- *use_hbbox_loss* and *loss_angle* allow custom regression loss calculation for rotated box.

  There are three combination options for regression:

  1. *use_hbbox_loss=False* and loss_angle is None.

```
bbox_pred———(tblr)——

angle_pred          decode——rbox_pred——(xywha)—loss_bbox
    │
    └———decode——(a)—
```

  2. *use_hbbox_loss=False* and loss_angle is specified. A angle loss is added on angle_pred.

```
bbox_pred———(tblr)——

angle_pred          decode——rbox_pred——(xywha)—loss_bbox
    │
    ├———decode——(a)—
```

(continues on next page)

(continued from previous page)

```
    │
    └──────────────────────────────────────────────loss_angle
```

3. *use_hbbox_loss=True* and loss_angle is specified. In this case the loss_angle must be set.

```
bbox_pred──(tblr)──decode──hbox_pred──(xyxy)──loss_bbox

angle_pred──────────────────────────────────────────loss_angle
```

- There's a *decoded_with_angle* flag in test_cfg, which is similar to training process.

  When *decoded_with_angle=True*:

```
bbox_pred────(tblr)──

angle_pred          decode──(xywha)──rbox_pred
    │
    └────decode──(a)─
```

  When *decoded_with_angle=False*:

```
bbox_pred──(tblr)─decode
                      │ (xyxy)

                  format──(xywh)──concat──(xywha)──rbox_pred

angle_pred────────decode────(a)───────
```

  Parameters

  - **head_module** (*ConfigType*) – Base module used for RTMDetRotatedHead.
  - **prior_generator** – Points generator feature maps in 2D points-based detectors.
  - **bbox_coder** (ConfigDict or dict) – Config of bbox coder.
  - **loss_cls** (ConfigDict or dict) – Config of classification loss.
  - **loss_bbox** (ConfigDict or dict) – Config of localization loss.
  - **angle_version** (*str*) – Angle representations. Defaults to 'le90'.
  - **use_hbbox_loss** (*bool*) – If true, use horizontal bbox loss and loss_angle should not be None. Default to False.
  - **angle_coder** (ConfigDict or dict) – Config of angle coder.
  - **loss_angle** (ConfigDict or dict, optional) – Config of angle loss.
  - **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
  - **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *angle_preds: List[torch.Tensor]*, *batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: List[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Compute losses of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Decoded box for each scale level with shape (N, num_anchors * 4, H, W) in [tl_x, tl_y, br_x, br_y] format.
- **angle_preds** (*list[Tensor]*) – Angle prediction for each scale level with shape (N, num_anchors * angle_out_dim, H, W).
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], Optional) – Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *angle_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted by the head into bbox results.

Parameters

- **cls_scores** (*list[Tensor]*) – Classification scores for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **angle_preds** (*list[Tensor]*) – Box angle for each scale level with shape (N, num_points * angle_dim, H, W)
- **objectnesses** (*list[Tensor], Optional*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_img_metas** (*list[dict], Optional*) – Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) – Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) – If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Defaults to True.

Returns

Object detection results of each image after the post process. Each item usually contains following keys. - scores (Tensor): Classification scores, has a shape

(num_instance,)

- labels (Tensor): Labels of bboxes, has a shape (num_instances,).
- bboxes (Tensor): Has a shape (num_instances, 5), the last dimension 4 arrange as (x, y, w, h, angle).

Return type list[InstanceData]

class mmyolo.models.dense_heads.**RTMDetRotatedSepBNHeadModule**(*num_classes: int*, *in_channels: int*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *feat_channels: int = 256*, *stacked_convs: int = 2*, *featmap_strides: Sequence[int] = [8, 16, 32]*, *share_conv: bool = True*, *pred_kernel_size: int = 1*, *angle_out_dim: int = 1*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Detection Head Module of RTMDet-R.

Compared with RTMDet Detection Head Module, RTMDet-R adds a conv for angle prediction. An *angle_out_dim* arg is added, which is generated by the angle_coder module and controls the angle pred dim.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) – The number of priors (points) at a point on the feature grid. Defaults to 1.
- **feat_channels** (*int*) – Number of hidden channels. Used in child classes. Defaults to 256
- **stacked_convs** (*int*) – Number of stacking convs of the head. Defaults to 2.
- **featmap_strides** (*Sequence[int]*) – Downsample factor of each feature map. Defaults to (8, 16, 32).
- **share_conv** (*bool*) – Whether to share conv layers between stages. Defaults to True.

- **pred_kernel_size** (*int*) – Kernel size of `nn.Conv2d`. Defaults to 1.
- **angle_out_dim** (*int*) – Encoded length of angle, will passed by head. Defaults to 1.
- **conv_cfg** (`ConfigDict` or dict, optional) – Config dict for convolution layer. Defaults to None.
- **norm_cfg** (`ConfigDict` or dict) – Config dict for normalization layer. Defaults to `dict(type='BN')`.
- **act_cfg** (`ConfigDict` or dict) – Config dict for activation layer. Default: dict(type='SiLU', inplace=True).

:param init_cfg (`ConfigDict` or list[`ConfigDict`] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*feats: Tuple[torch.Tensor, ...]*) → tuple
Forward features from the upstream network.

Parameters **feats** (*tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns

Usually a tuple of classification scores and bbox prediction - cls_scores (list[Tensor]): Classification scores for all scale

levels, each is a 4D-tensor, the channels number is num_base_priors * num_classes.

- bbox_preds (list[Tensor]): Box energies / deltas for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * 4.
- angle_preds (list[Tensor]): Angle prediction for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * angle_out_dim.

Return type tuple

init_weights() → None
Initialize weights of the head.

class mmyolo.models.dense_heads.**RTMDetSepBNHeadModule**(*num_classes: int*, *in_channels: int*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *feat_channels: int = 256*, *stacked_convs: int = 2*, *featmap_strides: Sequence[int] = [8, 16, 32]*, *share_conv: bool = True*, *pred_kernel_size: int = 1*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Detection Head of RTMDet.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) – The number of priors (points) at a point on the feature grid. Defaults to 1.
- **feat_channels** (*int*) – Number of hidden channels. Used in child classes. Defaults to 256
- **stacked_convs** (*int*) – Number of stacking convs of the head. Defaults to 2.
- **featmap_strides** (*Sequence[int]*) – Downsample factor of each feature map. Defaults to (8, 16, 32).
- **share_conv** (*bool*) – Whether to share conv layers between stages. Defaults to True.
- **pred_kernel_size** (*int*) – Kernel size of `nn.Conv2d`. Defaults to 1.
- **conv_cfg** (`ConfigDict` or dict, optional) – Config dict for convolution layer. Defaults to None.
- **norm_cfg** (`ConfigDict` or dict) – Config dict for normalization layer. Defaults to `dict(type='BN')`.
- **act_cfg** (`ConfigDict` or dict) – Config dict for activation layer. Default: dict(type='SiLU', inplace=True).

:param init_cfg (`ConfigDict` or list[`ConfigDict`] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*feats: Tuple[torch.Tensor, ...]*) → tuple
Forward features from the upstream network.

Parameters **feats** (*tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns

Usually a tuple of classification scores and bbox prediction - cls_scores (list[Tensor]): Classification scores for all scale

levels, each is a 4D-tensor, the channels number is num_base_priors * num_classes.

- bbox_preds (list[Tensor]): Box energies / deltas for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * 4.

Return type tuple

init_weights() → None
Initialize weights of the head.

class mmyolo.models.dense_heads.**YOLOXHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'YOLOXBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-16, 'loss_weight': 5.0, 'mode': 'square', 'reduction': 'sum', 'type': 'mmdet.IoULoss'}*, *loss_obj: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox_aux: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.L1Loss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOXHead head used in YOLOX.

Parameters

- **head_module** (*ConfigType*) – Base module used for YOLOXHead
- **prior_generator** – Points generator feature maps in 2D points-based detectors.
- **loss_cls** (ConfigDict or dict) – Config of classification loss.
- **loss_bbox** (ConfigDict or dict) – Config of localization loss.
- **loss_obj** (ConfigDict or dict) – Config of objectness loss.
- **loss_bbox_aux** (ConfigDict or dict) – Config of bbox aux loss.
- **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions, and objectnesses.

Return type Tuple[List]

static gt_instances_preprocess(*batch_gt_instances: torch.Tensor*, *batch_size: int*) → List[mmengine.structures.instance_data.InstanceData]

Split batch_gt_instances with batch size.

Parameters

- **batch_gt_instances** (*Tensor*) – Ground truth a 2D-Tensor for whole batch, shape [all_gt_bboxes, 6]
- **batch_size** (*int*) – Batch size.

Returns batch gt instances data, shape [batch_size, InstanceData]

Return type List

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *batch_gt_instances: torch.Tensor*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

Parameters

- **cls_scores** (*Sequence[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) – Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of losses.

Return type dict[str, Tensor]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOXHeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *feat_channels: int = 256*, *stacked_convs: int = 2*, *featmap_strides: Sequence[int] = [8, 16, 32]*, *use_depthwise: bool = False*, *dcn_on_last_conv: bool = False*, *conv_bias: Union[bool, str] = 'auto'*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOXHead head module used in `YOLOX.

https://arxiv.org/abs/2107.08430

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) – Number of channels in the input feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) – The number of priors (points) at a point on the feature grid
- **stacked_convs** (*int*) – Number of stacking convs of the head. Defaults to 2.
- **featmap_strides** (*Sequence[int]*) – Downsample factor of each feature map. Defaults to [8, 16, 32].
- **use_depthwise** (*bool*) – Whether to depthwise separable convolution in blocks. Defaults to False.
- **dcn_on_last_conv** (*bool*) – If true, use dcn in the last layer of towers. Defaults to False.
- **conv_bias** (*bool or str*) – If specified as *auto*, it will be decided by the norm_cfg. Bias of conv will be set as True if *norm_cfg* is None, otherwise False. Defaults to "auto".
- **conv_cfg** (ConfigDict or dict, optional) – Config dict for convolution layer. Defaults to None.
- **norm_cfg** (ConfigDict or dict) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (ConfigDict or dict) – Config dict for activation layer. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions, and objectnesses.

Return type Tuple[List]

forward_single(*x: torch.Tensor*, *cls_convs: torch.nn.modules.module.Module*, *reg_convs: torch.nn.modules.module.Module*, *conv_cls: torch.nn.modules.module.Module*, *conv_reg: torch.nn.modules.module.Module*, *conv_obj: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Forward feature of a single scale level.

init_weights()

Initialize weights of the head.

class mmyolo.models.dense_heads.**YOLOXPoseHead**(*loss_pose: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, **args*, ***kwargs*)

YOLOXPoseHead head used in `YOLO-Pose.

<https://arxiv.org/abs/2204.06806>`_. :param loss_pose: Config of keypoint OKS loss. :type loss_pose: ConfigDict, optional

decode_pose(*grids: torch.Tensor*, *offsets: torch.Tensor*, *strides: Union[torch.Tensor, int]*) → torch.Tensor

Decode regression offsets to keypoints.

Parameters

- **grids** (*torch.Tensor*) – The coordinates of the feature map grids.
- **offsets** (*torch.Tensor*) – The predicted offset of each keypoint relative to its corresponding grid.
- **strides** (*torch.Tensor | int*) – The stride of the feature map for each instance.

Returns The decoded keypoints coordinates.

Return type torch.Tensor

static gt_instances_preprocess(*batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, **args*, ***kwargs*) → List[mmengine.structures.instance_data.InstanceData]

Split batch_gt_instances with batch size.

Parameters

- **batch_gt_instances** (*Tensor*) – Ground truth a 2D-Tensor for whole batch, shape [all_gt_bboxes, 6]
- **batch_size** (*int*) – Batch size.

Returns batch gt instances data, shape [batch_size, InstanceData]

Return type List

static gt_kps_instances_preprocess(*batch_gt_instances: torch.Tensor*, *batch_gt_keypoints*, *batch_gt_keypoints_visible*, *batch_size: int*) → List[mmengine.structures.instance_data.InstanceData]

Split batch_gt_instances with batch size.

Parameters

- **batch_gt_instances** (*Tensor*) – Ground truth a 2D-Tensor for whole batch, shape [all_gt_bboxes, 6]
- **batch_size** (*int*) – Batch size.

Returns batch gt instances data, shape [batch_size, InstanceData]

Return type List

loss(*x: Tuple[torch.Tensor]*, *batch_data_samples: Union[list, dict]*) → dict

Perform forward propagation and loss calculation of the detection head on the features of the upstream network.

Parameters

- **x** (*tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.
- **batch_data_samples** (List[DetDataSample], dict) – The Data Samples. It usually includes information such as *gt_instance*, *gt_panoptic_seg* and *gt_sem_seg*.

Returns A dictionary of loss components.

Return type dict

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *kpt_preds: Sequence[torch.Tensor]*, *vis_preds: Sequence[torch.Tensor]*, *batch_gt_instances: torch.Tensor*, *batch_gt_keypoints: torch.Tensor*, *batch_gt_keypoints_visible: torch.Tensor*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

In addition to the base class method, keypoint losses are also calculated in this method.

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *kpt_preds: Optional[List[torch.Tensor]] = None*, *vis_preds: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted by the head into bbox and keypoint results.

In addition to the base class method, keypoint predictions are also calculated in this method.

class mmyolo.models.dense_heads.**YOLOXPoseHeadModule**(*num_keypoints: int*, **args*, ***kwargs*)

YOLOXPoseHeadModule serves as a head module for *YOLOX-Pose*.

In comparison to *YOLOXHeadModule*, this module introduces branches for keypoint prediction.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

init_weights()

Initialize weights of the head.

class mmyolo.models.dense_heads.**YOLOv5Head**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'base_sizes': [[(10, 13), (16, 30), (33, 23)], [(30, 61), (62, 45), (59, 119)], [(116, 90), (156, 198), (373, 326)]], 'strides': [8, 16, 32], 'type': 'mmdet.YOLOAnchorGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'YOLOv5BBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 0.5, 'reduction': 'mean', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xywh', 'eps': 1e-07, 'iou_mode': 'ciou', 'loss_weight': 0.05, 'reduction': 'mean', 'return_iou': True, 'type': 'IoULoss'}*, *loss_obj: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'mean', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *prior_match_thr: float = 4.0*, *near_neighbor_thr: float = 0.5*, *ignore_iof_thr: float = - 1.0*, *obj_level_weights: List[float] = [4.0, 1.0, 0.4]*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv5Head head used in *YOLOv5*.

Parameters

- **head_module** (*ConfigType*) – Base module used for YOLOv5Head
- **prior_generator** (*dict*) – Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) – Config of bbox coder.
- **loss_cls** (ConfigDict or dict) – Config of classification loss.
- **loss_bbox** (ConfigDict or dict) – Config of localization loss.
- **loss_obj** (ConfigDict or dict) – Config of objectness loss.
- **prior_match_thr** (*float*) – Defaults to 4.0.
- **ignore_iof_thr** (*float*) – Defaults to -1.0.
- **obj_level_weights** (*List[float]*) – Defaults to [4.0, 1.0, 0.4].
- **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions, and objectnesses.

Return type Tuple[List]

loss(*x: Tuple[torch.Tensor]*, *batch_data_samples: Union[list, dict]*) → dict

Perform forward propagation and loss calculation of the detection head on the features of the upstream network.

Parameters

- **x** (*tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.
- **batch_data_samples** (List[DetDataSample], dict) – The Data Samples. It usually includes information such as *gt_instance*, *gt_panoptic_seg* and *gt_sem_seg*.

Returns A dictionary of loss components.

Return type dict

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

Parameters

- **cls_scores** (*Sequence[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_gt_instances** (*Sequence[InstanceData]*) – Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*Sequence[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) – Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of losses.

Return type dict[str, Tensor]

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted by the head into bbox results. :param cls_scores: Classification scores for all

scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).

Parameters

- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **objectnesses** (*list[Tensor], Optional*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_img_metas** (*list[dict], Optional*) – Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) – Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) – If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Defaults to True.

Returns

Object detection results of each image after the post process. Each item usually contains following keys.

- scores (Tensor): Classification scores, has a shape (num_instance,)
- labels (Tensor): Labels of bboxes, has a shape (num_instances,).
- bboxes (Tensor): Has a shape (num_instances, 4), the last dimension 4 arrange as (x1, y1, x2, y2).

Return type list[InstanceData]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOv5HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 3*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv5Head head module used in *YOLOv5*.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) – Number of channels in the input feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) – The number of priors (points) at a point on the feature grid.
- **featmap_strides** (*Sequence[int]*) – Downsample factor of each feature map. Defaults to (8, 16, 32).

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions, and objectnesses.

Return type Tuple[List]

forward_single(*x: torch.Tensor*, *convs: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]
Forward feature of a single scale level.

init_weights()
Initialize the bias of YOLOv5 head.

class mmyolo.models.dense_heads.**YOLOv5InsHead**(**args*, *mask_overlap: bool = True*, *loss_mask: Union[mmengine.config.config.ConfigDict, dict] = {'reduction': 'none', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_mask_weight=0.05*, ***kwargs*)
YOLOv5 Instance Segmentation and Detection head.

Parameters

- **mask_overlap** (*bool*) – Defaults to True.
- **loss_mask** (ConfigDict or dict) – Config of mask loss.
- **loss_mask_weight** (*float*) – The weight of mask loss.

crop_mask(*masks: torch.Tensor*, *boxes: torch.Tensor*) → torch.Tensor
Crop mask by the bounding box.

Parameters

- **masks** (*Tensor*) – Predicted mask results. Has shape (1, num_instance, H, W).
- **boxes** (*Tensor*) – Tensor of the bbox. Has shape (num_instance, 4).

Returns The masks are being cropped to the bounding box.

Return type (torch.Tensor)

loss(*x: Tuple[torch.Tensor]*, *batch_data_samples: Union[list, dict]*) → dict
Perform forward propagation and loss calculation of the detection head on the features of the upstream network.

Parameters

- **x** (*tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.
- **batch_data_samples** (List[DetDataSample], dict) – The Data Samples. It usually includes information such as *gt_instance*, *gt_panoptic_seg* and *gt_sem_seg*.

Returns A dictionary of loss components.

Return type dict

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *coeff_preds: Sequence[torch.Tensor]*, *proto_preds: torch.Tensor*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_gt_masks: Sequence[torch.Tensor]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

Parameters

- **cls_scores** (*Sequence[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **coeff_preds** (*Sequence[Tensor]*) – Mask coefficient for each scale level, each is a 4D-tensor, the channel number is num_priors * mask_channels.
- **proto_preds** (*Tensor*) – Mask prototype features extracted from the mask head, has shape (batch_size, mask_channels, H, W).
- **batch_gt_instances** (*Sequence[InstanceData]*) – Batch of gt_instance. It usually includes `bboxes` and `labels` attributes.
- **batch_gt_masks** (*Sequence[Tensor]*) – Batch of gt_mask.
- **batch_img_metas** (*Sequence[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[`InstanceData`], optional) – Batch of gt_instances_ignore. It includes `bboxes` attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of losses.

Return type dict[str, Tensor]

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *coeff_preds: Optional[List[torch.Tensor]] = None*, *proto_preds: Optional[torch.Tensor] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted from the head into bbox results. Note: When score_factors is not None, the cls_scores are usually multiplied by it then obtain the real score used in NMS. :param cls_scores: Classification scores for all

scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).

Parameters

- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **objectnesses** (*list[Tensor], Optional*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **coeff_preds** (*list[Tensor]*) – Mask coefficients predictions for all scale levels, each is a 4D-tensor, has shape (batch_size, mask_channels, H, W).

- **proto_preds** (*Tensor*) – Mask prototype features extracted from the mask head, has shape (batch_size, mask_channels, H, W).
- **batch_img_metas** (*list[dict], Optional*) – Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) – Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) – If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Defaults to True.

Returns

Object detection and instance segmentation results of each image after the post process. Each item usually contains following keys.

- scores (Tensor): Classification scores, has a shape (num_instance,)
- labels (Tensor): Labels of bboxes, has a shape (num_instances,).
- bboxes (Tensor): Has a shape (num_instances, 4), the last dimension 4 arrange as (x1, y1, x2, y2).
- masks (Tensor): Has a shape (num_instances, h, w).

Return type list[InstanceData]

process_mask(*mask_proto: torch.Tensor*, *mask_coeff_pred: torch.Tensor*, *bboxes: torch.Tensor*, *shape: Tuple[int, int]*, *upsample: bool = False*) → torch.Tensor

Generate mask logits results.

Parameters

- **mask_proto** (*Tensor*) – Mask prototype features. Has shape (num_instance, mask_channels).
- **mask_coeff_pred** (*Tensor*) – Mask coefficients prediction for single image. Has shape (mask_channels, H, W)
- **bboxes** (*Tensor*) – Tensor of the bbox. Has shape (num_instance, 4).
- **shape** (*Tuple*) – Batch input shape of image.
- **upsample** (*bool*) – Whether upsample masks results to batch input shape. Default to False.

Returns

Instance segmentation masks for each instance. Has shape (num_instance, H, W).

Return type Tensor

class mmyolo.models.dense_heads.**YOLOv5InsHeadModule**(**args*, *num_classes: int*, *mask_channels: int = 32*, *proto_channels: int = 256*, *widen_factor: float = 1.0*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, ***kwargs*)

Detection and Instance Segmentation Head of YOLOv5.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **mask_channels** (*int*) – Number of channels in the mask feature map. This is the channel count of the mask.
- **proto_channels** (*int*) – Number of channels in the proto feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **norm_cfg** (ConfigDict or dict) – Config dict for normalization layer. Defaults to `dict(type='BN', momentum=0.03, eps=0.001)`.
- **act_cfg** (ConfigDict or dict) – Config dict for activation layer. Default: dict(type='SiLU', inplace=True).

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions, objectnesses, and mask predictions.

Return type Tuple[List]

forward_single(*x: torch.Tensor*, *convs_pred: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Forward feature of a single scale level.

class mmyolo.models.dense_heads.**YOLOv6Head**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0.5, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'alpha': 0.75, 'gamma': 2.0, 'iou_weighted': True, 'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.VarifocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xyxy', 'iou_mode': 'giou', 'loss_weight': 2.5, 'reduction': 'mean', 'return_iou': False, 'type': 'IoULoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv6Head head used in YOLOv6.

Parameters

- **head_module** (*ConfigType*) – Base module used for YOLOv6Head
- **prior_generator** (*dict*) – Points generator feature maps in 2D points-based detectors.
- **loss_cls** (ConfigDict or dict) – Config of classification loss.

- **loss_bbox** (ConfigDict or dict) – Config of localization loss.
- **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *bbox_dist_preds: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

Parameters

- **cls_scores** (*Sequence[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) – Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of losses.

Return type dict[str, Tensor]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOv6HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *reg_max=0*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv6Head head module used in `YOLOv6.

<https://arxiv.org/pdf/2209.02976>`_.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) – Number of channels in the input feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** – (int): The number of priors (points) at a point on the feature grid.
- **featmap_strides** (*Sequence[int]*) –

  Downsample factor of each feature map. Defaults to [8, 16, 32].

  None, otherwise False. Defaults to "auto".
- **norm_cfg** (ConfigDict or dict) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (ConfigDict or dict) – Config dict for activation layer. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions.

Return type Tuple[List]

forward_single(*x: torch.Tensor*, *stem: torch.nn.modules.module.Module*, *cls_conv: torch.nn.modules.module.Module*, *cls_pred: torch.nn.modules.module.Module*, *reg_conv: torch.nn.modules.module.Module*, *reg_pred: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor]
Forward feature of a single scale level.

init_weights()
Initialize the weights.

class mmyolo.models.dense_heads.**YOLOv7Head**(**args*, *simota_candidate_topk: int = 20*, *simota_iou_weight: float = 3.0*, *simota_cls_weight: float = 1.0*, *aux_loss_weights: float = 0.25*, ***kwargs*)
YOLOv7Head head used in YOLOv7.

Parameters

- **simota_candidate_topk** (*int*) – The candidate top-k which used to get top-k ious to calculate dynamic-k in BatchYOLOv7Assigner. Defaults to 10.
- **simota_iou_weight** (*float*) – The scale factor for regression iou cost in BatchYOLOv7Assigner. Defaults to 3.0.
- **simota_cls_weight** (*float*) – The scale factor for classification cost in BatchYOLOv7Assigner. Defaults to 1.0.

loss_by_feat(*cls_scores: Sequence[Union[torch.Tensor, List]]*, *bbox_preds: Sequence[Union[torch.Tensor, List]]*, *objectnesses: Sequence[Union[torch.Tensor, List]]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

Parameters

- **cls_scores** (*Sequence[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) – Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) – Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of losses.

Return type dict[str, Tensor]

class mmyolo.models.dense_heads.**YOLOv7HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 3*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv7Head head module used in YOLOv7.

init_weights()

Initialize the bias of YOLOv7 head.

class mmyolo.models.dense_heads.**YOLOv7p6HeadModule**(**args*, *main_out_channels: Sequence[int] = [256, 512, 768, 1024]*, *aux_out_channels: Sequence[int] = [320, 640, 960, 1280]*, *use_aux: bool = True*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, ***kwargs*)

YOLOv7Head head module used in YOLOv7.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions, and objectnesses.

Return type Tuple[List]

forward_single(*x: torch.Tensor*, *convs: torch.nn.modules.module.Module*, *aux_convs: Optional[torch.nn.modules.module.Module]*) → Tuple[Union[torch.Tensor, List], Union[torch.Tensor, List], Union[torch.Tensor, List]]

Forward feature of a single scale level.

init_weights()

Initialize the bias of YOLOv5 head.

class mmyolo.models.dense_heads.**YOLOv8Head**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0.5, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 0.5, 'reduction': 'none', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xyxy', 'iou_mode': 'ciou', 'loss_weight': 7.5, 'reduction': 'sum', 'return_iou': False, 'type': 'IoULoss'}*, *loss_dfl={'loss_weight': 0.375, 'reduction': 'mean', 'type': 'mmdet.DistributionFocalLoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv8Head head used in *YOLOv8*.

Parameters

- **head_module** (ConfigDict or dict) – Base module used for YOLOv8Head
- **prior_generator** (*dict*) – Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) – Config of bbox coder.
- **loss_cls** (ConfigDict or dict) – Config of classification loss.
- **loss_bbox** (ConfigDict or dict) – Config of localization loss.
- **loss_dfl** (ConfigDict or dict) – Config of Distribution Focal Loss.
- **train_cfg** (ConfigDict or dict, optional) – Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.

Defaults to None.

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *bbox_dist_preds: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

Parameters

- **cls_scores** (*Sequence[Tensor]*) – Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) – Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **bbox_dist_preds** (*Sequence[Tensor]*) – Box distribution logits for each scale level with shape (bs, reg_max + 1, H*W, 4).
- **batch_gt_instances** (list[InstanceData]) – Batch of gt_instance. It usually includes `bboxes` and `labels` attributes.
- **batch_img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) – Batch of gt_instances_ignore. It includes `bboxes` attribute data that is ignored during training and testing. Defaults to None.

Returns A dictionary of losses.

Return type dict[str, Tensor]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOv8HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *reg_max: int = 16*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv8HeadModule head module used in *YOLOv8*.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) – Number of channels in the input feature map.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) – The number of priors (points) at a point on the feature grid.

- **featmap_strides** (*Sequence[int]*) – Downsample factor of each feature map. Defaults to [8, 16, 32].
- **reg_max** (*int*) – Max value of integral set :math: `{0, ..., reg_max-1}` in QFL setting. Defaults to 16.
- **norm_cfg** (ConfigDict or dict) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (ConfigDict or dict) – Config dict for activation layer. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

Parameters **x** (*Tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns A tuple of multi-level classification scores, bbox predictions

Return type Tuple[List]

forward_single(*x: torch.Tensor*, *cls_pred: torch.nn.modules.container.ModuleList*, *reg_pred: torch.nn.modules.container.ModuleList*) → Tuple
Forward feature of a single scale level.

init_weights(*prior_prob=0.01*)
Initialize the weight and bias of PPYOLOE head.

58.4 detectors

class mmyolo.models.detectors.**YOLODetector**(*backbone: Union[mmengine.config.config.ConfigDict, dict]*, *neck: Union[mmengine.config.config.ConfigDict, dict]*, *bbox_head: Union[mmengine.config.config.ConfigDict, dict]*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *data_preprocessor: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, *use_syncbn: bool = True*)
Implementation of YOLO Series

Parameters

- **backbone** (ConfigDict or dict) – The backbone config.
- **neck** (ConfigDict or dict) – The neck config.
- **bbox_head** (ConfigDict or dict) – The bbox head config.
- **train_cfg** (ConfigDict or dict, optional) – The training config of YOLO. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) – The testing config of YOLO. Defaults to None.

- **data_preprocessor** (ConfigDict or dict, optional) – Config of DetDataPreprocessor to process the input data. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

Parameters **use_syncbn** (*bool*) – whether to use SyncBatchNorm. Defaults to True.

58.5 layers

class mmyolo.models.layers.**BepC3StageBlock**(*in_channels: int*, *out_channels: int*, *num_blocks: int = 1*, *hidden_ratio: float = 0.5*, *concat_all_layer: bool = True*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*)

Beer-mug RepC3 Block.

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **num_blocks** (*int*) – Number of blocks. Defaults to 1
- **hidden_ratio** (*float*) – Hidden channel expansion. Default: 0.5
- **concat_all_layer** (*bool*) – Concat all layer when forward calculate. Default: True
- **block_cfg** (*dict*) – Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **norm_cfg** (*ConfigType*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*ConfigType*) – Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).

forward(*x*)

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 mmyolo.models.layers.**BiFusion**(*in_channels0: int*, *in_channels1: int*, *out_channels: int*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*)

BiFusion Block in YOLOv6.

BiFusion fuses current-, high- and low-level features. Compared with concatenation in PAN, it fuses an extra low-level feature.

Parameters

- **in_channels0** (*int*) – The channels of current-level feature.
- **in_channels1** (*int*) – The input channels of lower-level feature.
- **out_channels** (*int*) – The out channels of the BiFusion module.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x: List[torch.Tensor]*) → torch.Tensor

Forward process :param x: The tensor list of length 3.

x[0]: The high-level feature. x[1]: The current-level feature. x[2]: The low-level feature.

class mmyolo.models.layers.**CSPLayerWithTwoConv**(*in_channels: int*, *out_channels: int*, *expand_ratio: float = 0.5*, *num_blocks: int = 1*, *add_identity: bool = True*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Cross Stage Partial Layer with 2 convolutions.

Parameters

- **in_channels** (*int*) – The input channels of the CSP layer.
- **out_channels** (*int*) – The output channels of the CSP layer.
- **expand_ratio** (*float*) – Ratio to adjust the number of channels of the hidden layer. Defaults to 0.5.
- **num_blocks** (*int*) – Number of blocks. Defaults to 1
- **add_identity** (*bool*) – Whether to add identity in blocks. Defaults to True.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Defaults to None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN').
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict], optional): Initialization config dict.
Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor
Forward process.

class mmyolo.models.layers.**DarknetBottleneck**(*in_channels: int*, *out_channels: int*, *expansion: float = 0.5*, *kernel_size: Sequence[int] = (1, 3)*, *padding: Sequence[int] = (0, 1)*, *add_identity: bool = True*, *use_depthwise: bool = False*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

The basic bottleneck block used in Darknet.

Each ResBlock consists of two ConvModules and the input is added to the final output. Each ConvModule is composed of Conv, BN, and LeakyReLU. The first convLayer has filter size of k1Xk1 and the second one has the filter size of k2Xk2.

Note: This DarknetBottleneck is little different from MMDet's, we can change the kernel size and padding for each conv.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The output channels of this Module.
- **expansion** (*float*) – The kernel size for hidden channel. Defaults to 0.5.
- **kernel_size** (*Sequence[int]*) – The kernel size of the convolution. Defaults to (1, 3).
- **padding** (*Sequence[int]*) – The padding size of the convolution. Defaults to (0, 1).
- **add_identity** (*bool*) – Whether to add identity to the out. Defaults to True
- **use_depthwise** (*bool*) – Whether to use depthwise separable convolution. Defaults to False
- **conv_cfg** (*dict*) – Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN').
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='Swish').

class mmyolo.models.layers.**EELANBlock**(*num_elan_block: int*, ***kwargs*)
Expand efficient layer aggregation networks for YOLOv7.

Parameters **num_elan_block** (*int*) – The number of ELANBlock.

forward(*x: torch.Tensor*) → torch.Tensor
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 mmyolo.models.layers.**ELANBlock**(*in_channels: int*, *out_channels: int*, *middle_ratio: float*, *block_ratio: float*, *num_blocks: int = 2*, *num_convs_in_block: int = 1*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Efficient layer aggregation networks for YOLOv7.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The out channels of this Module.
- **middle_ratio** (*float*) – The scaling ratio of the middle layer based on the in_channels.
- **block_ratio** (*float*) – The scaling ratio of the block layer based on the in_channels.
- **num_blocks** (*int*) – The number of blocks in the main branch. Defaults to 2.
- **num_convs_in_block** (*int*) – The number of convs pre block. Defaults to 1.
- **conv_cfg** (*dict*) – Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**EffectiveSELayer**(*channels: int*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'HSigmoid'}*)

Effective Squeeze-Excitation.

From *CenterMask : Real-Time Anchor-Free Instance Segmentation* arxiv (https://arxiv.org/abs/1911.06667) This code referenced to https://github.com/youngwanLEE/CenterMask/blob/72147e8aae673fcaf4103ee90a6a6b73863e7fa1/maskrcnn_benchmark/modeling/backbone/vovnet.py#L108-L121 # noqa

Parameters

- **channels** (*int*) – The input and output channels of this Module.
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='HSigmoid').

forward(*x: torch.Tensor*) → torch.Tensor
Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**ExpMomentumEMA**(*model: torch.nn.modules.module.Module*, *momentum: float = 0.0002*, *gamma: int = 2000*, *interval=1*, *device: Optional[torch.device] = None*, *update_buffers: bool = False*)
Exponential moving average (EMA) with exponential momentum strategy, which is used in YOLO.

Parameters

- **model** (*nn.Module*) – The model to be averaged.
- **momentum** (*float*) –

  The momentum used for updating ema parameter. Ema's parameters are updated with the formula:

  *averaged_param = (1-momentum) * averaged_param + momentum * source_param*. Defaults to 0.0002.
- **gamma** (*int*) – Use a larger momentum early in training and gradually annealing to a smaller value to update the ema model smoothly. The momentum is calculated as *(1 - momentum) * exp(-(1 + steps) / gamma) + momentum*. Defaults to 2000.
- **interval** (*int*) – Interval between two updates. Defaults to 1.
- **device** (*torch.device, optional*) – If provided, the averaged model will be stored on the device. Defaults to None.
- **update_buffers** (*bool*) – if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

avg_func(*averaged_param: torch.Tensor*, *source_param: torch.Tensor*, *steps: int*)
Compute the moving average of the parameters using the exponential momentum strategy.

Parameters

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

update_parameters(*model: torch.nn.modules.module.Module*)
Update the parameters after each training step.

Parameters **model** (*nn.Module*) – The model of the parameter needs to be updated.

class mmyolo.models.layers.**ImplicitA**(*in_channels: int*, *mean: float = 0.0*, *std: float = 0.02*)
Implicit add layer in YOLOv7.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **mean** (*float*) – Mean value of implicit module. Defaults to 0.
- **std** (*float*) – Std value of implicit module. Defaults to 0.02

forward(*x*)
Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**ImplicitM**(*in_channels: int*, *mean: float = 1.0*, *std: float = 0.02*)
Implicit multiplier layer in YOLOv7.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **mean** (*float*) – Mean value of implicit module. Defaults to 1.
- **std** (*float*) – Std value of implicit module. Defaults to 0.02.

forward(*x*)

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**MaxPoolAndStrideConvBlock**(*in_channels: int*, *out_channels: int*, *maxpool_kernel_sizes: int = 2*, *use_in_channels_of_middle: bool = False*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Max pooling and stride conv layer for YOLOv7.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The out channels of this Module.
- **maxpool_kernel_sizes** (*int*) – kernel sizes of pooling layers. Defaults to 2.
- **use_in_channels_of_middle** (*bool*) – Whether to calculate middle channels based on in_channels. Defaults to False.
- **conv_cfg** (*dict*) – Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**PPYOLOEBasicBlock**(*in_channels: int*, *out_channels: int*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *shortcut: bool = True*, *use_alpha: bool = False*)

PPYOLOE Backbone BasicBlock.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The output channels of this Module.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.1, eps=1e-5).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **shortcut** (*bool*) – Whether to add inputs and outputs together
- **the end of this layer. Defaults to True.** (*at*) –
- **use_alpha** (*bool*) – Whether to use *alpha* parameter at 1x1 conv.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process. :param inputs: The input tensor. :type inputs: Tensor

Returns The output tensor.

Return type Tensor

class mmyolo.models.layers.**RepStageBlock**(*in_channels: int*, *out_channels: int*, *num_blocks: int = 1*, *bottle_block: torch.nn.modules.module.Module = <class 'mmyolo.models.layers.yolo_bricks.RepVGGBlock'>*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*)

RepStageBlock is a stage block with rep-style basic block.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The output channels of this Module.
- **num_blocks** (*int, tuple[int]*) – Number of blocks. Defaults to 1.
- **bottle_block** (*nn.Module*) – Basic unit of RepStage. Defaults to RepVGGBlock.
- **block_cfg** (*ConfigType*) – Config of RepStage. Defaults to 'RepVGGBlock'.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process.

Parameters **x** (*Tensor*) – The input tensor.

Returns The output tensor.

Return type Tensor

class mmyolo.models.layers.**RepVGGBlock**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int]] = 3*, *stride: Union[int, Tuple[int]] = 1*, *padding: Union[int, Tuple[int]] = 1*, *dilation: Union[int, Tuple[int]] = 1*, *groups: Optional[int] = 1*, *padding_mode: Optional[str] = 'zeros'*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *use_se: bool = False*, *use_alpha: bool = False*, *use_bn_first=True*, *deploy: bool = False*)

RepVGGBlock is a basic rep-style block, including training and deploy status This code is based on https://github.com/DingXiaoH/RepVGG/blob/main/repvgg.py.

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple*) – Stride of the convolution. Default: 1
- **padding** (*int, tuple*) – Padding added to all four sides of the input. Default: 1
- **dilation** (*int or tuple*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **padding_mode** (*string, optional*) – Default: 'zeros'
- **use_se** (*bool*) – Whether to use se. Default: False
- **use_alpha** (*bool*) – Whether to use *alpha* parameter at 1x1 conv. In PPYOLOE+ model backbone, *use_alpha* will be set to True. Default: False.
- **use_bn_first** (*bool*) – Whether to use bn layer before conv. In YOLOv6 and YOLOv7, this will be set to True. In PPYOLOE, this will be set to False. Default: True.
- **deploy** (*bool*) – Whether in deploy mode. Default: False

forward(*inputs: torch.Tensor*) → torch.Tensor

Forward process. :param inputs: The input tensor. :type inputs: Tensor

Returns The output tensor.

Return type Tensor

get_equivalent_kernel_bias()

Derives the equivalent kernel and bias in a differentiable way.

Returns Equivalent kernel and bias

Return type tuple

switch_to_deploy()

Switch to deploy mode.

class mmyolo.models.layers.**SPPFBottleneck**(*in_channels: int*, *out_channels: int*, *kernel_sizes: Union[int, Sequence[int]] = 5*, *use_conv_first: bool = True*, *mid_channels_scale: float = 0.5*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Spatial pyramid pooling - Fast (SPPF) layer for YOLOv5, YOLOX and PPYOLOE by Glenn Jocher

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The output channels of this Module.

- **kernel_sizes** (*int, tuple[int]*) – Sequential or number of kernel sizes of pooling layers. Defaults to 5.
- **use_conv_first** (*bool*) – Whether to use conv before pooling layer. In YOLOv5 and YOLOX, the para set to True. In PPYOLOE, the para set to False. Defaults to True.
- **mid_channels_scale** (*float*) – Channel multiplier, multiply in_channels by this amount to get mid_channels. This parameter is valid only when use_conv_fist=True.Defaults to 0.5.
- **conv_cfg** (*dict*) – Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**SPPFCSPBlock**(*in_channels: int*, *out_channels: int*, *expand_ratio: float = 0.5*, *kernel_sizes: Union[int, Sequence[int]] = 5*, *is_tiny_version: bool = False*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Spatial pyramid pooling - Fast (SPPF) layer with CSP for YOLOv7

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The output channels of this Module.
- **expand_ratio** (*float*) – Expand ratio of SPPCSPBlock. Defaults to 0.5.
- **kernel_sizes** (*int, tuple[int]*) – Sequential or number of kernel sizes of pooling layers. Defaults to 5.
- **is_tiny_version** (*bool*) – Is tiny version of SPPFCSPBlock. If True, it means it is a yolov7 tiny model. Defaults to False.
- **conv_cfg** (*dict*) – Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*) → torch.Tensor
Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**TinyDownSampleBlock**(*in_channels: int*, *out_channels: int*, *middle_ratio: float = 1.0*, *kernel_sizes: Union[int, Sequence[int]] = 3*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'negative_slope': 0.1, 'type': 'LeakyReLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Down sample layer for YOLOv7-tiny.

Parameters

- **in_channels** (*int*) – The input channels of this Module.
- **out_channels** (*int*) – The out channels of this Module.
- **middle_ratio** (*float*) – The scaling ratio of the middle layer based on the in_channels. Defaults to 1.0.
- **kernel_sizes** (*int, tuple[int]*) – Sequential or number of kernel sizes of pooling layers. Defaults to 3.
- **conv_cfg** (*dict*) – Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='LeakyReLU', negative_slope=0.1).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*) → torch.Tensor
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.

58.6 losses

class mmyolo.models.losses.**IoULoss**(*iou_mode: str = 'ciou'*, *bbox_format: str = 'xywh'*, *eps: float = 1e-07*, *reduction: str = 'mean'*, *loss_weight: float = 1.0*, *return_iou: bool = True*)

IoULoss.

Computing the IoU loss between a set of predicted bboxes and target bboxes. :param iou_mode: Options are "ciou".

Defaults to "ciou".

Parameters

- **bbox_format** (*str*) – Options are "xywh" and "xyxy". Defaults to "xywh".
- **eps** (*float*) – Eps to avoid log(0).
- **reduction** (*str*) – Options are "none", "mean" and "sum".
- **loss_weight** (*float*) – Weight of loss.
- **return_iou** (*bool*) – If True, return loss and iou.

forward(*pred: torch.Tensor*, *target: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, *avg_factor: Optional[float] = None*, *reduction_override: Optional[Union[str, bool]] = None*) → Tuple[torch.Tensor, torch.Tensor]

Forward function.

Parameters

- **pred** (*Tensor*) – Predicted bboxes of format (x1, y1, x2, y2) or (x, y, w, h),shape (n, 4).
- **target** (*Tensor*) – Corresponding gt bboxes, shape (n, 4).
- **weight** (*Tensor, optional*) – Element-wise weights.
- **avg_factor** (*float, optional*) – Average factor when computing the mean of losses.
- **reduction_override** (*str, bool, optional*) – Same as built-in losses of PyTorch. Defaults to None.

Returns

Return type loss or tuple(loss, iou)

class mmyolo.models.losses.**OksLoss**(*metainfo: Optional[str] = None*, *loss_weight: float = 1.0*)

A PyTorch implementation of the Object Keypoint Similarity (OKS) loss as described in the paper "YOLO-Pose: Enhancing YOLO for Multi Person Pose Estimation Using Object Keypoint Similarity Loss" by Debapriya et al.

(2022). The OKS loss is used for keypoint-based object recognition and consists of a measure of the similarity between predicted and ground truth keypoint locations, adjusted by the size of the object in the image. The loss function takes as input the predicted keypoint locations, the ground truth keypoint locations, a mask indicating which keypoints are valid, and bounding boxes for the objects. :param metainfo: Path to a JSON file containing information

about the dataset's annotations.

Parameters **loss_weight** (*float*) – Weight for the loss.

compute_oks(*output: torch.Tensor*, *target: torch.Tensor*, *target_weights: torch.Tensor*, *bboxes: Optional[torch.Tensor] = None*) → torch.Tensor

Calculates the OKS loss.

Parameters

- **output** (*Tensor*) – Predicted keypoints in shape N x k x 2, where N is batch size, k is the number of keypoints, and 2 are the xy coordinates.
- **target** (*Tensor*) – Ground truth keypoints in the same shape as output.
- **target_weights** (*Tensor*) – Mask of valid keypoints in shape N x k, with 1 for valid and 0 for invalid.
- **bboxes** (*Optional[Tensor]*) – Bounding boxes in shape N x 4, where 4 are the xyxy coordinates.

Returns The calculated OKS loss.

Return type Tensor

forward(*output: torch.Tensor*, *target: torch.Tensor*, *target_weights: torch.Tensor*, *bboxes: Optional[torch.Tensor] = None*) → torch.Tensor

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.

---

mmyolo.models.losses.**bbox_overlaps**(*pred: torch.Tensor*, *target: torch.Tensor*, *iou_mode: str = 'ciou'*, *bbox_format: str = 'xywh'*, *siou_theta: float = 4.0*, *eps: float = 1e-07*) → torch.Tensor

Calculate overlap between two set of bboxes. Implementation of paper `Enhancing Geometric Factors into Model Learning and Inference for Object Detection and Instance Segmentation.

In the CIoU implementation of YOLOv5 and MMDetection, there is a slight difference in the way the alpha parameter is computed.

mmdet version: alpha = (ious > 0.5).float() * v / (1 - ious + v)

YOLOv5 version: alpha = v / (v - ious + (1 + eps)

Parameters

- **pred** (*Tensor*) – Predicted bboxes of format (x1, y1, x2, y2) or (x, y, w, h),shape (n, 4).
- **target** (*Tensor*) – Corresponding gt bboxes, shape (n, 4).
- **iou_mode** (*str*) – Options are ('iou', 'ciou', 'giou', 'siou'). Defaults to "ciou".
- **bbox_format** (*str*) – Options are "xywh" and "xyxy". Defaults to "xywh".
- **siou_theta** (*float*) – siou_theta for SIoU when calculate shape cost. Defaults to 4.0.
- **eps** (*float*) – Eps to avoid log(0).

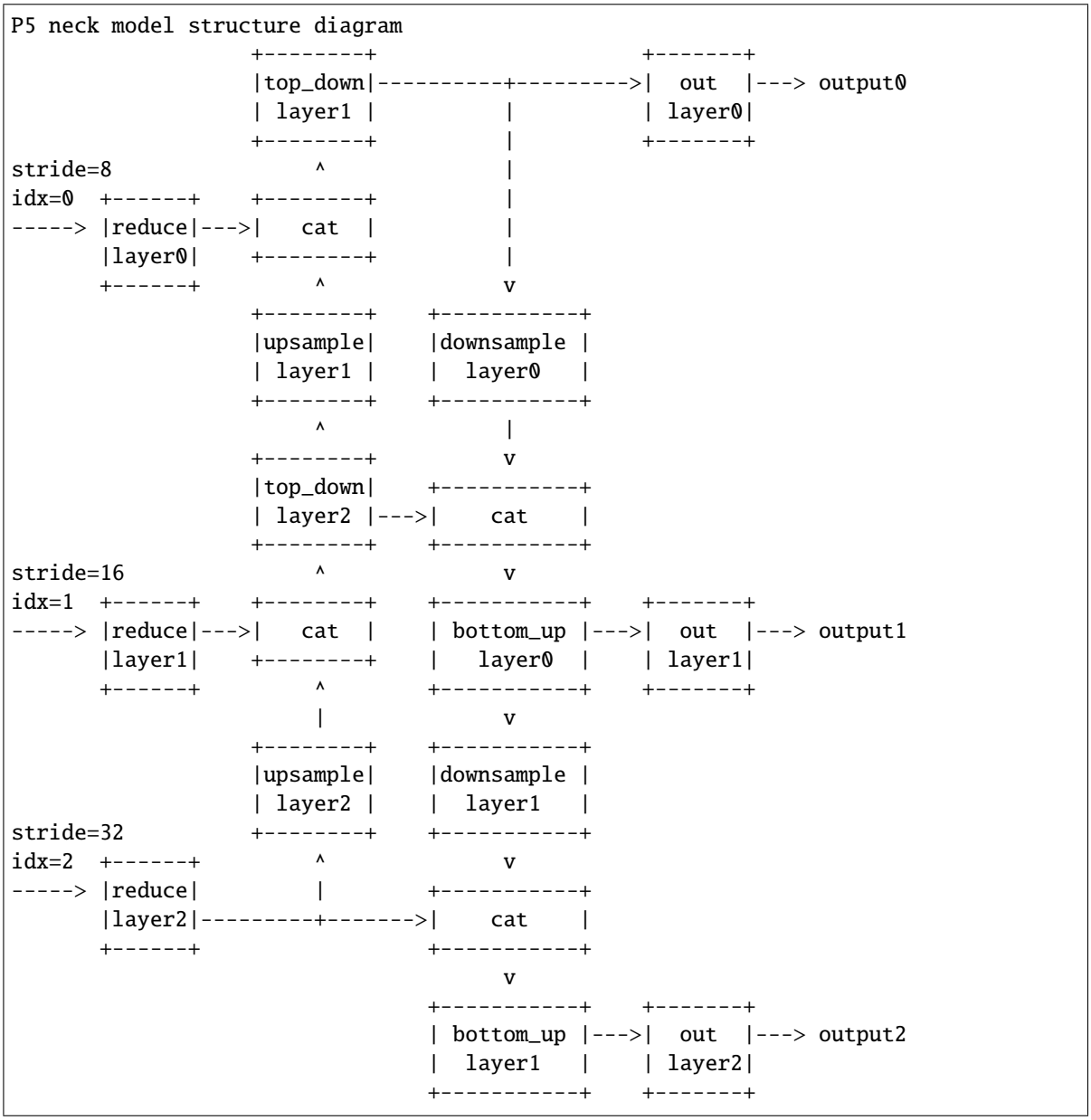
Returns shape (n,).

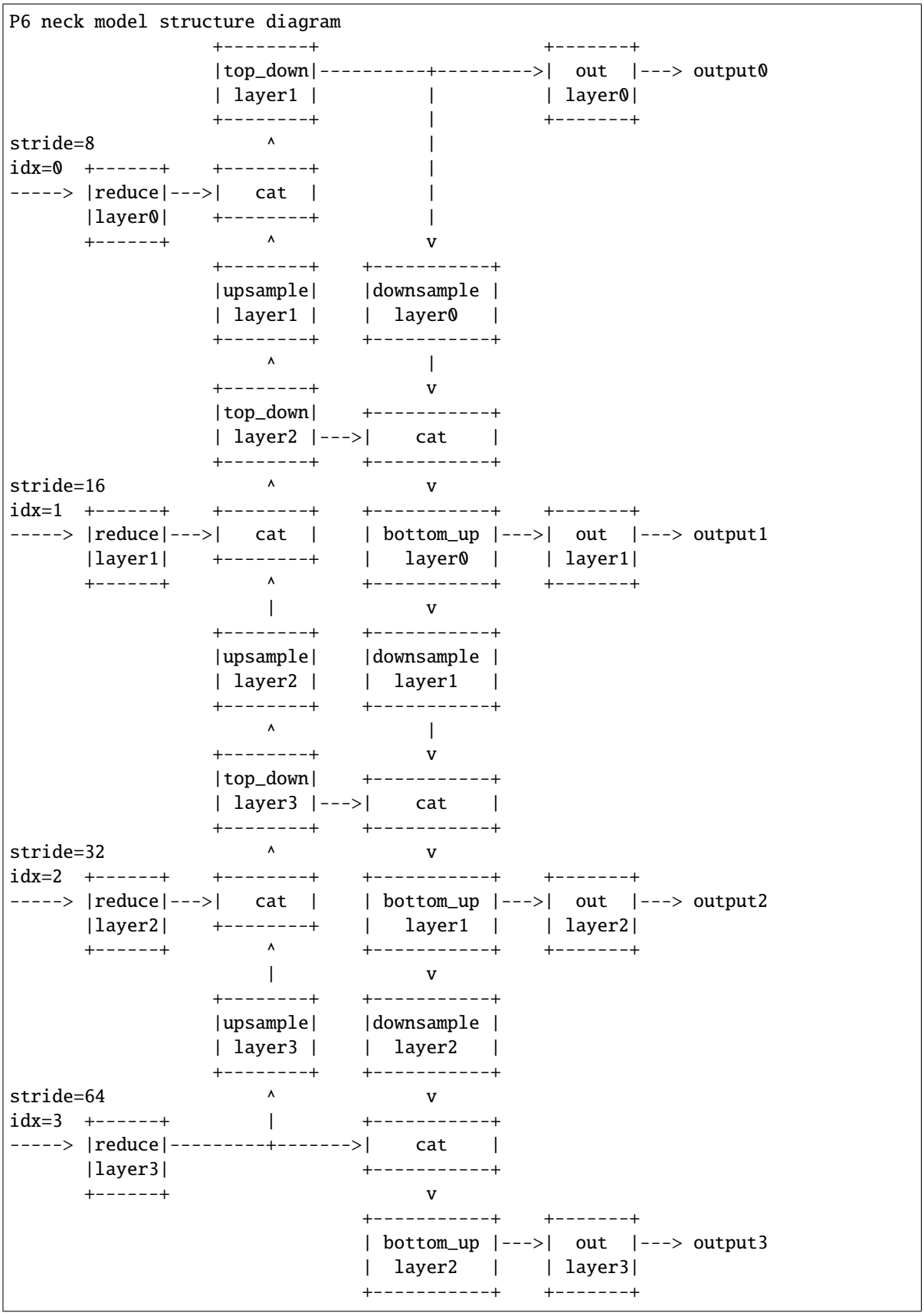
Return type Tensor

58.7 necks

class mmyolo.models.necks.**BaseYOLONeck**(*in_channels: List[int]*, *out_channels: Union[int, List[int]]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *upsample_feats_cat_first: bool = True*, *freeze_all: bool = False*, *norm_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *act_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, ***kwargs*)

Base neck used in YOLO series.




P6 neck model structure diagram
top_down layer1
out layer0
output0
stride=8
idx=0
reduce layer0
cat
upsample layer1
downsample layer0
top_down layer2
cat
stride=16
idx=1
reduce layer1
cat
bottom_up layer0
out layer1
output1
upsample layer2
downsample layer1
top_down layer3
cat
stride=32
idx=2
reduce layer2
cat
bottom_up layer1
out layer2
output2
upsample layer3
downsample layer2
stride=64
idx=3
reduce layer3
cat
bottom_up layer2
out layer3
output3

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **upsample_feats_cat_first** (*bool*) – Whether the output features are concat first after upsampling in the topdown module. Defaults to True. Currently only YOLOv7 is false.
- **freeze_all** (*bool*) – Whether to freeze the model. Defaults to False
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to None.
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to None.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

abstract build_bottom_up_layer(*idx: int*)
build bottom up layer.

abstract build_downsample_layer(*idx: int*)
build downsample layer.

abstract build_out_layer(*idx: int*)
build out layer.

abstract build_reduce_layer(*idx: int*)
build reduce layer.

abstract build_top_down_layer(*idx: int*)
build top down layer.

abstract build_upsample_layer(*idx: int*)
build upsample layer.

forward(*inputs: List[torch.Tensor]*) → tuple
Forward function.

train(*mode=True*)
Convert the model into training mode while keep the normalization layer freezed.

class mmyolo.models.necks.**CSPNeXtPAFPN**(*in_channels: Sequence[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 3*, *freeze_all: bool = False*, *use_depthwise: bool = False*, *expand_ratio: float = 0.5*, *upsample_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'mode': 'nearest', 'scale_factor': 2}*, *conv_cfg: Optional[bool] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = {'a': 2.23606797749979, 'distribution': 'uniform', 'layer': 'Conv2d', 'mode': 'fan_in', 'nonlinearity': 'leaky_relu', 'type': 'Kaiming'}*)

Path Aggregation Network with CSPNeXt blocks.

Parameters

- **in_channels** (*Sequence[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 3.
- **use_depthwise** (*bool*) – Whether to use depthwise separable convolution in blocks. Defaults to False.
- **expand_ratio** (*float*) – Ratio to adjust the number of channels of the hidden layer. Defaults to 0.5.
- **upsample_cfg** (*dict*) – Config dict for interpolate layer. Default: *dict(scale_factor=2, mode='nearest')*
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: dict(type='BN')
- **act_cfg** (*dict*) – Config dict for activation layer. Default: dict(type='SiLU', inplace=True)
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The downsample layer.

Return type nn.Module

build_out_layer(*idx: int*) → torch.nn.modules.module.Module
build out layer.

Parameters **idx** (*int*) – layer idx.

Returns The out layer.

Return type nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module
build reduce layer.

Parameters **idx** (*int*) – layer idx.

Returns The reduce layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

build_upsample_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build upsample layer.

class mmyolo.models.necks.**PPYOLOECSPPAFPN**(*in_channels: List[int] = [256, 512, 1024]*, *out_channels: List[int] = [256, 512, 1024]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *freeze_all: bool = False*, *num_csplayer: int = 1*, *num_blocks_per_layer: int = 3*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'shortcut': False, 'type': 'PPYOLOEBasicBlock', 'use_alpha': False}*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *drop_block_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, *use_spp: bool = False*)

CSPPAN in PPYOLOE.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*List[int]*) – Number of output channels (used at each scale).
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **freeze_all** (*bool*) – Whether to freeze the model.
- **num_csplayer** (*int*) – Number of *CSPResLayer* in per layer. Defaults to 1.

- **num_blocks_per_layer** (*int*) – Number of blocks per *CSPResLayer*. Defaults to 3.
- **block_cfg** (*dict*) – Config dict for block. Defaults to dict(type='PPYOLOEBasicBlock', shortcut=True, use_alpha=False)
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.1, eps=1e-5).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **drop_block_cfg** (*dict, optional*) – Drop block config. Defaults to None. If you want to use Drop block after *CSPResLayer*, you can set this para as dict(type='mmdet.DropBlock', drop_prob=0.1, block_size=3, warm_iters=0).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.
- **use_spp** (*bool*) – Whether to use *SPP* in reduce layer. Defaults to False.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module
build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module
build downsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The downsample layer.

Return type nn.Module

build_out_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build out layer.

build_reduce_layer(*idx: int*)
build reduce layer.

Parameters **idx** (*int*) – layer idx.

Returns The reduce layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module
build upsample layer.

class mmyolo.models.necks.**YOLOXPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 3*, *use_depthwise: bool = False*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOX.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale).
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 1.
- **use_depthwise** (*bool*) – Whether to use depthwise separable convolution. Defaults to False.
- **freeze_all** (*bool*) – Whether to freeze the model. Defaults to False.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The downsample layer.

Return type nn.Module

build_out_layer(*idx: int*) → torch.nn.modules.module.Module

build out layer.

Parameters **idx** (*int*) – layer idx.

Returns The out layer.

Return type nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module
build reduce layer.

Parameters **idx** (*int*) – layer idx.

Returns The reduce layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

build_upsample_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build upsample layer.

class mmyolo.models.necks.**YOLOv5PAFPN**(*in_channels: List[int]*, *out_channels: Union[List[int], int]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 1*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)
Path Aggregation Network used in YOLOv5.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) – Whether to freeze the model
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module
build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module
build downsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The downsample layer.

Return type nn.Module

build_out_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build out layer.

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module
build reduce layer.

Parameters **idx** (*int*) – layer idx.

Returns The reduce layer.

Return type nn.Module

build_top_down_layer(*idx: int*)
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

build_upsample_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build upsample layer.

init_weights()
Initialize the weights.

class mmyolo.models.necks.**YOLOv6CSPRepBiPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *hidden_ratio: float = 0.5*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6 3.0.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.

- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) – Whether to freeze the model.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) – Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **block_act_cfg** (*dict*) – Config dict for activation layer used in each stage. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module
build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

class mmyolo.models.necks.**YOLOv6CSPRepPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *hidden_ratio: float = 0.5*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.

- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) – Whether to freeze the model.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) – Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **block_act_cfg** (*dict*) – Config dict for activation layer used in each stage. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module
build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

class mmyolo.models.necks.**YOLOv6RepBiPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6 3.0.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 1.

- **freeze_all** (*bool*) – Whether to freeze the model.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) – Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module
build upsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The upsample layer.

Return type nn.Module

forward(*inputs: List[torch.Tensor]*) → tuple
Forward function.

class mmyolo.models.necks.**YOLOv6RepPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) – Whether to freeze the model.

- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) – Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module
build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module
build downsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The downsample layer.

Return type nn.Module

build_out_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build out layer.

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module
build reduce layer.

Parameters **idx** (*int*) – layer idx.

Returns The reduce layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module
build upsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The upsample layer.

Return type nn.Module

init_weights()
Initialize the weights.

class mmyolo.models.necks.**YOLOv7PAFPN**(*in_channels: List[int]*, *out_channels: List[int]*, *block_cfg: dict = {'block_ratio': 0.25, 'middle_ratio': 0.5, 'num_blocks': 4, 'num_convs_in_block': 1, 'type': 'ELANBlock'}*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *spp_expand_ratio: float = 0.5*, *is_tiny_version: bool = False*, *use_maxpool_in_downsample: bool = True*, *use_in_channels_in_downsample: bool = False*, *use_repconv_outs: bool = True*, *upsample_feats_cat_first: bool = False*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv7.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale).
- **block_cfg** (*dict*) – Config dict for block.
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **spp_expand_ratio** (*float*) – Expand ratio of SPPCSPBlock. Defaults to 0.5.
- **is_tiny_version** (*bool*) – Is tiny version of neck. If True, it means it is a yolov7 tiny model. Defaults to False.
- **use_maxpool_in_downsample** (*bool*) – Whether maxpooling is used in downsample layers. Defaults to True.
- **use_in_channels_in_downsample** (*bool*) – MaxPoolAndStrideConvBlock module input parameters. Defaults to False.
- **use_repconv_outs** (*bool*) – Whether to use *repconv* in the output layer. Defaults to True.
- **upsample_feats_cat_first** (*bool*) – Whether the output features are concat first after upsampling in the topdown module. Defaults to True. Currently only YOLOv7 is false.
- **freeze_all** (*bool*) – Whether to freeze the model. Defaults to False.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module
build downsample layer.

Parameters **idx** (*int*) – layer idx.

Returns The downsample layer.

Return type nn.Module

build_out_layer(*idx: int*) → torch.nn.modules.module.Module
build out layer.

Parameters **idx** (*int*) – layer idx.

Returns The out layer.

Return type nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module
build reduce layer.

Parameters **idx** (*int*) – layer idx.

Returns The reduce layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module
build upsample layer.

class mmyolo.models.necks.**YOLOv8PAFPN**(*in_channels: List[int]*, *out_channels: Union[List[int], int]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 3*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)
Path Aggregation Network used in YOLOv8.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **deepen_factor** (*float*) – Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) – Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) – Number of bottlenecks in CSPLayer. Defaults to 1.

- **freeze_all** (*bool*) – Whether to freeze the model
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) – Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module
build bottom up layer.

Parameters **idx** (*int*) – layer idx.

Returns The bottom up layer.

Return type nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module
build reduce layer.

Parameters **idx** (*int*) – layer idx.

Returns The reduce layer.

Return type nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

Parameters **idx** (*int*) – layer idx.

Returns The top down layer.

Return type nn.Module

58.8 task_modules

class mmyolo.models.task_modules.**BatchATSSAssigner**(*num_classes: int*, *iou_calculator: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'mmdet.BboxOverlaps2D'}*, *topk: int = 9*)

Assign a batch of corresponding gt bboxes or background to each prior.

This code is based on https://github.com/meituan/YOLOv6/blob/main/yolov6/assigners/atss_assigner.py

Each proposal will be assigned with *0* or a positive integer indicating the ground truth index.

- 0: negative sample, no assigned gt
- positive integer: positive sample, index (1-based) of assigned gt

Parameters

- **num_classes** (*int*) – number of class
- **iou_calculator** (ConfigDict or dict) – Config dict for iou calculator. Defaults to dict(type='BboxOverlaps2D')
- **topk** (*int*) – number of priors selected in each level

forward(*pred_bboxes: torch.Tensor*, *priors: torch.Tensor*, *num_level_priors: List*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *pad_bbox_flag: torch.Tensor*) → dict

Assign gt to priors.

The assignment is done in following steps

1. compute iou between all prior (prior of all pyramid levels) and gt
2. compute center distance between all prior and gt
3. on each pyramid level, for each gt, select k prior whose center are closest to the gt center, so we total select k*l prior as candidates for each gt
4. get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold
5. select these candidates whose iou are greater than or equal to the threshold as positive
6. limit the positive sample's center in gt

Parameters

- **pred_bboxes** (*Tensor*) – Predicted bounding boxes, shape(batch_size, num_priors, 4)
- **priors** (*Tensor*) – Model priors with stride, shape(num_priors, 4)
- **num_level_priors** (*List*) – Number of bboxes in each level, len(3)
- **gt_labels** (*Tensor*) – Ground truth label, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) – Ground truth bbox, shape(batch_size, num_gt, 4)
- **pad_bbox_flag** (*Tensor*) – Ground truth bbox mask, 1 means bbox, 0 means no bbox, shape(batch_size, num_gt, 1)

Returns

Assigned result 'assigned_labels' (Tensor): shape(batch_size, num_gt) 'assigned_bboxes' (Tensor): shape(batch_size, num_gt, 4) 'assigned_scores' (Tensor):

shape(batch_size, num_gt, number_classes)

'fg_mask_pre_prior' (Tensor): shape(bs, num_gt)

Return type assigned_result (dict)

get_targets(*gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *assigned_gt_inds: torch.Tensor*, *fg_mask_pre_prior: torch.Tensor*, *num_priors: int*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Get target info.

Parameters

- **gt_labels** (*Tensor*) – Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) – Ground true bboxes, shape(batch_size, num_gt, 4)
- **assigned_gt_inds** (*Tensor*) – Assigned ground truth indexes, shape(batch_size, num_priors)
- **fg_mask_pre_prior** (*Tensor*) – Force ground truth matching mask, shape(batch_size, num_priors)
- **num_priors** (*int*) – Number of priors.
- **batch_size** (*int*) – Batch size.

- **num_gt** (*int*) – Number of ground truth.

Returns

Assigned labels, shape(batch_size, num_priors)

assigned_bboxes (Tensor): Assigned bboxes, shape(batch_size, num_priors)

assigned_scores (Tensor): Assigned scores, shape(batch_size, num_priors)

Return type assigned_labels (Tensor)

select_topk_candidates(*distances: torch.Tensor*, *num_level_priors: List[int]*, *pad_bbox_flag: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Selecting candidates based on the center distance.

Parameters

- **distances** (*Tensor*) – Distance between all bbox and gt, shape(batch_size, num_gt, num_priors)
- **num_level_priors** (*List[int]*) – Number of bboxes in each level, len(3)
- **pad_bbox_flag** (*Tensor*) – Ground truth bbox mask, shape(batch_size, num_gt, 1)

Returns

Flag show that each level have topk candidates or not, shape(batch_size, num_gt, num_priors)

candidate_idxs (Tensor): Candidates index, shape(batch_size, num_gt, num_gt)

Return type is_in_candidate_list (Tensor)

static threshold_calculator(*is_in_candidate: List*, *candidate_idxs: torch.Tensor*, *overlaps: torch.Tensor*, *num_priors: int*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor]

Get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold.

Parameters

- **is_in_candidate** (*Tensor*) – Flag show that each level have topk candidates or not, shape(batch_size, num_gt, num_priors).
- **candidate_idxs** (*Tensor*) – Candidates index, shape(batch_size, num_gt, num_gt)
- **overlaps** (*Tensor*) – Overlaps area, shape(batch_size, num_gt, num_priors).
- **num_priors** (*int*) – Number of priors.
- **batch_size** (*int*) – Batch size.
- **num_gt** (*int*) – Number of ground truth.

Returns

Overlap threshold of per ground truth, shape(batch_size, num_gt, 1).

candidate_overlaps (Tensor): Candidate overlaps, shape(batch_size, num_gt, num_priors).

Return type overlaps_thr_per_gt (Tensor)

class mmyolo.models.task_modules.**BatchTaskAlignedAssigner**(*num_classes: int*, *topk: int = 13*, *alpha: float = 1.0*, *beta: float = 6.0*, *eps: float = 1e-07*, *use_ciou: bool = False*)

This code referenced to https://github.com/meituan/YOLOv6/blob/main/yolov6/ assigners/tal_assigner.py. Batch Task aligned assigner base on the paper: TOOD: Task-aligned One-stage Object Detection.. Assign a corresponding gt bboxes or background to a batch of predicted bboxes. Each bbox will be assigned with *0* or a positive integer indicating the ground truth index. - 0: negative sample, no assigned gt - positive integer: positive sample, index (1-based) of assigned gt :param num_classes: number of class :type num_classes: int :param topk: number of bbox selected in each level :type topk: int :param alpha: Hyper-parameters related to alignment_metrics.

Defaults to 1.0

Parameters

- **beta** (*float*) – Hyper-parameters related to alignment_metrics. Defaults to 6.
- **eps** (*float*) – Eps to avoid log(0). Default set to 1e-9
- **use_ciou** (*bool*) – Whether to use ciou while calculating iou. Defaults to False.

forward(*pred_bboxes: torch.Tensor*, *pred_scores: torch.Tensor*, *priors: torch.Tensor*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *pad_bbox_flag: torch.Tensor*) → dict

Assign gt to bboxes.

The assignment is done in following steps 1. compute alignment metric between all bbox (bbox of all pyramid

levels) and gt

2. select top-k bbox as candidates for each gt
3. limit the positive sample's center in gt (because the anchor-free detector only can predict positive distance)

Parameters

- **pred_bboxes** (*Tensor*) – Predict bboxes, shape(batch_size, num_priors, 4)
- **pred_scores** (*Tensor*) – Scores of predict bboxes, shape(batch_size, num_priors, num_classes)
- **priors** (*Tensor*) – Model priors, shape (num_priors, 4)
- **gt_labels** (*Tensor*) – Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) – Ground true bboxes, shape(batch_size, num_gt, 4)
- **pad_bbox_flag** (*Tensor*) – Ground truth bbox mask, 1 means bbox, 0 means no bbox, shape(batch_size, num_gt, 1)

Returns

assigned_labels (Tensor): Assigned labels, shape(batch_size, num_priors)

assigned_bboxes (Tensor): Assigned boxes, shape(batch_size, num_priors, 4)

assigned_scores (Tensor): Assigned scores, shape(batch_size, num_priors, num_classes)

fg_mask_pre_prior (Tensor): Force ground truth matching mask, shape(batch_size, num_priors)

Return type assigned_result (dict) Assigned result

get_box_metrics(*pred_bboxes: torch.Tensor*, *pred_scores: torch.Tensor*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor]

Compute alignment metric between all bbox and gt.

Parameters

- **pred_bboxes** (*Tensor*) – Predict bboxes, shape(batch_size, num_priors, 4)
- **pred_scores** (*Tensor*) – Scores of predict bbox, shape(batch_size, num_priors, num_classes)
- **gt_labels** (*Tensor*) – Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) – Ground true bboxes, shape(batch_size, num_gt, 4)
- **batch_size** (*int*) – Batch size.
- **num_gt** (*int*) – Number of ground truth.

Returns

Align metric, shape(batch_size, num_gt, num_priors)

overlaps (Tensor): Overlaps, shape(batch_size, num_gt, num_priors)

Return type alignment_metrics (Tensor)

get_pos_mask(*pred_bboxes: torch.Tensor*, *pred_scores: torch.Tensor*, *priors: torch.Tensor*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *pad_bbox_flag: torch.Tensor*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Get possible mask.

Parameters

- **pred_bboxes** (*Tensor*) – Predict bboxes, shape(batch_size, num_priors, 4)
- **pred_scores** (*Tensor*) – Scores of predict bbox, shape(batch_size, num_priors, num_classes)
- **priors** (*Tensor*) – Model priors, shape (num_priors, 2)
- **gt_labels** (*Tensor*) – Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) – Ground true bboxes, shape(batch_size, num_gt, 4)
- **pad_bbox_flag** (*Tensor*) – Ground truth bbox mask, 1 means bbox, 0 means no bbox, shape(batch_size, num_gt, 1)
- **batch_size** (*int*) – Batch size.
- **num_gt** (*int*) – Number of ground truth.

Returns

Possible mask, shape(batch_size, num_gt, num_priors)

alignment_metrics (Tensor): Alignment metrics, shape(batch_size, num_gt, num_priors)

overlaps (Tensor): Overlaps of gt_bboxes and pred_bboxes, shape(batch_size, num_gt, num_priors)

Return type pos_mask (Tensor)

get_targets(*gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *assigned_gt_idxs: torch.Tensor*, *fg_mask_pre_prior: torch.Tensor*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Get assigner info.

Parameters

- **gt_labels** (*Tensor*) – Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) – Ground true bboxes, shape(batch_size, num_gt, 4)
- **assigned_gt_idxs** (*Tensor*) – Assigned ground truth indexes, shape(batch_size, num_priors)
- **fg_mask_pre_prior** (*Tensor*) – Force ground truth matching mask, shape(batch_size, num_priors)
- **batch_size** (*int*) – Batch size.
- **num_gt** (*int*) – Number of ground truth.

Returns

Assigned labels, shape(batch_size, num_priors)

assigned_bboxes (Tensor): Assigned bboxes, shape(batch_size, num_priors)

assigned_scores (Tensor): Assigned scores, shape(batch_size, num_priors)

Return type assigned_labels (Tensor)

select_topk_candidates(*alignment_gt_metrics: torch.Tensor*, *using_largest_topk: bool = True*, *topk_mask: Optional[torch.Tensor] = None*) → torch.Tensor

Compute alignment metric between all bbox and gt.

Parameters

- **alignment_gt_metrics** (*Tensor*) – Alignment metric of gt candidates, shape(batch_size, num_gt, num_priors)
- **using_largest_topk** (*bool*) – Controls whether to using largest or smallest elements.
- **topk_mask** (*Tensor*) – Topk mask, shape(batch_size, num_gt, self.topk)

Returns

Topk candidates mask, shape(batch_size, num_gt, num_priors)

Return type Tensor

class mmyolo.models.task_modules.**YOLOXBBoxCoder**(*use_box_type: bool = False*, ***kwargs*)

YOLOX BBox coder.

This decoder decodes pred bboxes (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

decode(*priors: torch.Tensor*, *pred_bboxes: torch.Tensor*, *stride: Union[torch.Tensor, int]*) → torch.Tensor

Decode regression results (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

Parameters

- **priors** (*torch.Tensor*) – Basic boxes or points, e.g. anchors.
- **pred_bboxes** (*torch.Tensor*) – Encoded boxes with shape
- **stride** (*torch.Tensor | int*) – Strides of bboxes.

Returns Decoded boxes.

Return type torch.Tensor

encode(***kwargs*)
Encode deltas between bboxes and ground truth boxes.

class mmyolo.models.task_modules.**YOLOv5BBoxCoder**(*use_box_type: bool = False*, ***kwargs*)
YOLOv5 BBox coder.

This decoder decodes pred bboxes (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

decode(*priors: torch.Tensor*, *pred_bboxes: torch.Tensor*, *stride: Union[torch.Tensor, int]*) → torch.Tensor
Decode regression results (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

Parameters

- **priors** (*torch.Tensor*) – Basic boxes or points, e.g. anchors.
- **pred_bboxes** (*torch.Tensor*) – Encoded boxes with shape
- **stride** (*torch.Tensor | int*) – Strides of bboxes.

Returns Decoded boxes.

Return type torch.Tensor

encode(***kwargs*)
Encode deltas between bboxes and ground truth boxes.

58.9 utils

class mmyolo.models.utils.**OutputSaveFunctionWrapper**(*func: Callable*, *spec: Optional[Dict]*)
A class that wraps a function and saves its outputs.

This class can be used to decorate a function to save its outputs. It wraps the function with a *__call__* method that calls the original function and saves the results in a log attribute. :param func: A function to wrap. :type func: Callable :param spec: A dictionary of global variables to use as the

namespace for the wrapper. If *None*, the global namespace of the original function is used.

class mmyolo.models.utils.**OutputSaveObjectWrapper**(*obj: Any*)
A wrapper class that saves the output of function calls on an object.

clear()
Clears the log of function call outputs.

mmyolo.models.utils.**gt_instances_preprocess**(*batch_gt_instances: Union[torch.Tensor, Sequence]*, *batch_size: int*) → torch.Tensor
Split batch_gt_instances with batch size.

From [all_gt_bboxes, box_dim+2] to [batch_size, number_gt, box_dim+1]. For horizontal box, box_dim=4, for rotated box, box_dim=5

If some shape of single batch smaller than gt bbox len, then using zeros to fill.

Parameters

- **batch_gt_instances** (*Sequence[Tensor]*) – Ground truth instances for whole batch, shape [all_gt_bboxes, box_dim+2]
- **batch_size** (*int*) – Batch size.

Returns

batch gt instances data, shape [batch_size, number_gt, box_dim+1]

Return type Tensor

mmyolo.models.utils.**make_divisible**(*x: float*, *widen_factor: float = 1.0*, *divisor: int = 8*) → int
Make sure that x*widen_factor is divisible by divisor.

mmyolo.models.utils.**make_round**(*x: float*, *deepen_factor: float = 1.0*) → int
Make sure that x*deepen_factor becomes an integer not less than 1.

CHAPTER

FIFTYNINE

MMYOLO.UTILS

CHAPTER

SIXTY

ENGLISH

CHAPTER

SIXTYONE

CHAPTER

SIXTYTWO

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

C

D

E

F

G

I

L

M

O

P

R

S

T

U

W

Y