
mmdeploy Documentation

Release 0.14.0

MMDeploy Contributors

May 26, 2023

GET STARTED

1	Get Started	3
1.1	Introduction	3
1.2	Prerequisites	4
1.3	Installation	4
1.4	Convert Model	5
1.5	Inference Model	6
1.6	Evaluate Model	9
2	Build from Source	11
2.1	Download	11
2.2	Build	11
3	Use Docker Image	13
3.1	Build docker image	13
3.2	Run docker container	14
3.3	FAQs	14
4	Build from Script	15
5	CMake Build Option Spec	17
6	How to convert model	19
6.1	How to convert models from Pytorch to other backends	19
6.2	How to evaluate the exported models	21
6.3	List of supported models exportable to other backends	21
7	How to write config	23
7.1	1. How to write onnx config	23
7.2	2. How to write codebase config	25
7.3	3. How to write backend config	25
7.4	4. A complete example of mmcls on TensorRT	26
7.5	5. The name rules of our deployment config	26
7.6	6. How to write model config	27
8	How to evaluate model	29
8.1	Prerequisite	29
8.2	Usage	29
8.3	Description of all arguments	30
8.4	Example	30
8.5	Note	31

9	Quantize model	33
9.1	Why quantization ?	33
9.2	Post training quantization scheme	33
9.3	How to convert model	33
9.4	Custom calibration dataset	34
10	Useful Tools	35
10.1	torch2onnx	35
10.2	extract	36
10.3	onnx2pplnn	36
10.4	onnx2tensorrt	37
10.5	onnx2ncnn	38
10.6	profiler	38
10.7	generate_md_table	40
11	Supported models	41
11.1	Note	41
12	Benchmark	43
12.1	Backends	43
12.2	Latency benchmark	43
12.3	Performance benchmark	44
12.4	Notes	44
13	Test on embedded device	45
13.1	Software and hardware environment	45
13.2	mmcls	45
13.3	mmocr detection	45
13.4	mmpose	45
13.5	mmseg	46
13.6	Notes	46
14	Test on TVM	47
14.1	Supported Models	47
14.2	Test	47
15	Quantization test result	49
15.1	Quantize with ncnn	49
16	MMClassification Support	51
16.1	MMClassification installation tutorial	51
16.2	List of MMClassification models supported by MMDeploy	51
17	MMDetection Support	53
17.1	MMDetection installation tutorial	53
17.2	List of MMDetection models supported by MMDeploy	53
18	MMSegmentation Support	55
18.1	MMSegmentation installation tutorial	55
18.2	List of MMSegmentation models supported by MMDeploy	55
18.3	Reminder	55
19	MMEdition Support	57
19.1	MMEdition installation tutorial	57
19.2	MMEdition models support	57

20	MMOCR Support	59
20.1	MMOCR installation tutorial	59
20.2	List of MMOCR models supported by MMDeploy	59
20.3	Reminder	59
21	MMPose Support	63
21.1	MMPose installation tutorial	63
21.2	MMPose models support	63
22	MMDetection3d Support	65
22.1	MMDetection3d installation tutorial	65
22.2	Example	65
22.3	List of MMDetection3d models supported by MMDeploy	65
23	MMRotate Support	67
23.1	MMRotate installation tutorial	67
23.2	MMRotate models support	67
24	MMAction2 Support	69
24.1	Install mmaction2	69
24.2	Supported models	69
25	Supported ncnn feature	71
26	ONNX Runtime Support	73
26.1	Introduction of ONNX Runtime	73
26.2	Installation	73
26.3	Build custom ops	73
26.4	How to convert a model	74
26.5	How to add a new custom op	74
26.6	Reminder	74
26.7	References	74
27	OpenVINO Support	75
27.1	Installation	75
27.2	Usage	75
27.3	List of supported models exportable to OpenVINO from MMDetection	76
27.4	Deployment config	76
27.5	Troubleshooting	76
28	PPLNN Support	77
28.1	Installation	77
28.2	Usage	77
29	SNPE feature support	79
30	TensorRT Support	81
30.1	Installation	81
30.2	Convert model	82
30.3	FAQs	83
31	TorchScript support	85
31.1	Introduction of TorchScript	85
31.2	Build custom ops	85
31.3	How to convert a model	86
31.4	SDK backend	86

31.5	FAQs	86
32	Supported RKNN feature	87
33	Core ML feature support	89
33.1	Installation	89
33.2	Usage	89
34	ONNX Runtime Ops	91
34.1	grid_sampler	92
34.2	MMCVModulatedDeformConv2d	92
34.3	NMSRotated	92
34.4	RoIAlignRotated	93
35	TensorRT Ops	95
35.1	TRTBatchedNMS	97
35.2	grid_sampler	97
35.3	MMCVInstanceNormalization	98
35.4	MMCVModulatedDeformConv2d	98
35.5	MMCVMultiLevelRoiAlign	98
35.6	MMCVRoIAlign	99
35.7	ScatterND	99
35.8	TRTBatchedRotatedNMS	100
35.9	GridPriorsTRT	100
35.10	ScaledDotProductAttentionTRT	101
35.11	GatherTopk	101
36	ncnn Ops	103
36.1	Expand	104
36.2	Gather	104
36.3	Shape	104
36.4	TopK	105
37	mmdeploy Architecture	107
37.1	Take a general look at the directory structure	107
37.2	Model Conversion	108
37.3	SDK	109
38	How to support new models	111
38.1	Function Rewriter	111
38.2	Module Rewriter	112
38.3	Custom Symbolic	112
39	How to support new backends	115
39.1	Prerequisites	115
39.2	Support backend conversion	115
39.3	Support backend inference	117
39.4	Support new backends using MMDeploy as a third party	119
40	How to add test units for backend ops	121
40.1	Prerequisite	121
40.2	1. Add the test program test_XXXX()	121
40.3	2. Test Methods	123
41	How to test rewritten models	125
41.1	Test rewritten model with small changes	125

41.2	Test rewritten model with big changes	126
41.3	Note	128
42	How to get partitioned ONNX models	129
42.1	Step 1: Mark inputs/outputs	129
42.2	Step 2: Add partition config	130
42.3	Step 3: Get partitioned onnx models	131
43	How to do regression test	133
43.1	1. Python Environment	133
43.2	2. Usage	133
43.3	Example	134
43.4	3. Regression Test Configuration	135
43.5	4. Generated Report	137
43.6	5. Supported Backends	137
43.7	6. Supported Codebase and Metrics	137
44	ONNX export Optimizer	139
44.1	Installation	139
44.2	Usage	139
45	Cross compile snpe inference server on Ubuntu 18	141
45.1	1. Environment	141
45.2	2. Cross compile gRPC with NDK	141
45.3	3. (Skipable) Self-test whether NDK gRPC is available	142
45.4	4. Cross compile snpe inference server	143
45.5	5. Regenerate the proto interface	143
45.6	Reference	144
46	Frequently Asked Questions	145
46.1	TensorRT	145
46.2	Libtorch	145
46.3	Windows	146
46.4	ONNX Runtime	146
46.5	Pip	147
47	English	149
48		151
49	apis	153
50	apis/tensorrt	155
51	apis/onnxruntime	157
52	apis/ncnn	159
53	apis/pplnn	161
54	Indices and tables	163

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

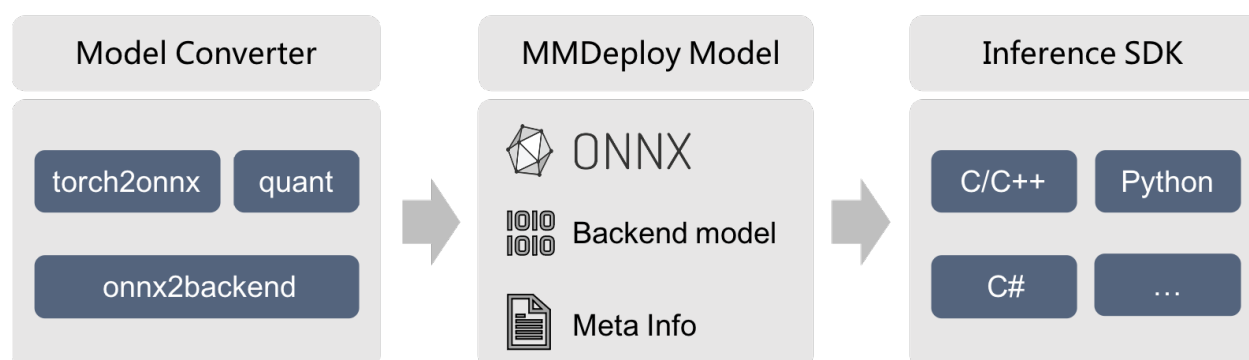
GET STARTED

MMDeploy provides useful tools for deploying OpenMMLab models to various platforms and devices.

With the help of them, you can not only do model deployment using our pre-defined pipelines but also customize your own deployment pipeline.

1.1 Introduction

In MMDeploy, the deployment pipeline can be illustrated by a sequential modules, i.e., Model Converter, MMDeploy Model and Inference SDK.



1.1.1 Model Converter

Model Converter aims at converting training models from OpenMMLab into backend models that can be run on target devices. It is able to transform PyTorch model into IR model, i.e., ONNX, TorchScript, as well as convert IR model to backend model. By combining them together, we can achieve one-click **end-to-end** model deployment.

1.1.2 MMDeploy Model

MMDeploy Model is the result package exported by Model Converter. Beside the backend models, it also includes the model meta info, which will be used by Inference SDK.

1.1.3 Inference SDK

Inference SDK is developed by C/C++, wrapping the preprocessing, model forward and postprocessing modules in model inference. It supports FFI such as C, C++, Python, C#, Java and so on.

1.2 Prerequisites

In order to do an end-to-end model deployment, MMDeploy requires Python 3.6+ and PyTorch 1.5+.

Step 0. Download and install Miniconda from the [official website](#).

Step 1. Create a conda environment and activate it.

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

Step 2. Install PyTorch following [official instructions](#), e.g.

On GPU platforms:

```
conda install pytorch=={pytorch_version} torchvision=={torchvision_version} cudatoolkit=
↪{cudatoolkit_version} -c pytorch -c conda-forge
```

On CPU platforms:

```
conda install pytorch=={pytorch_version} torchvision=={torchvision_version} cpuonly -c
↪pytorch
```

Note: On GPU platform, please ensure that {cudatoolkit_version} matches your host CUDA toolkit version. Otherwise, it probably brings in conflicts when deploying model with TensorRT.

1.3 Installation

We recommend that users follow our best practices installing MMDeploy.

Step 0. Install [MMCV](#).

```
pip install -U openmim
mim install mmcv-full
```

Step 1. Install MMDeploy and inference engine

We recommend using MMDeploy precompiled package as our best practice. Currently, we support model converter and sdk inference pypi package, and the sdk c/cpp library is provided [here](#). You can download them according to your target platform and device.

The supported platform and device matrix is presented as following:

Note: if MMDeploy prebuilt package doesn't meet your target platforms or devices, please [build MMDeploy from source](#)

Take the latest precompiled package as example, you can install it as follows:

```

# 1. install MMDeploy model converter
pip install mmdemo==0.14.0

# 2. install MMDeploy sdk inference
# you can install one to install according whether you need gpu inference
# 2.1 support onnxruntime
pip install mmdemo-runtime==0.14.0
# 2.2 support onnxruntime-gpu, tensorrt
pip install mmdemo-runtime-gpu==0.14.0

# 3. install inference engine
# 3.1 install TensorRT
# !!! If you want to convert a tensorrt model or inference with tensorrt,
# download TensorRT-8.2.3.0 CUDA 11.x tar package from NVIDIA, and extract it to the
# current directory
pip install TensorRT-8.2.3.0/python/tensorrt-8.2.3.0-cp38-none-linux_x86_64.whl
pip install pycuda
export TENSORRT_DIR=$(pwd)/TensorRT-8.2.3.0
export LD_LIBRARY_PATH=${TENSORRT_DIR}/lib:$LD_LIBRARY_PATH
# !!! Moreover, download cuDNN 8.2.1 CUDA 11.x tar package from NVIDIA, and extract it
# to the current directory
export CUDNN_DIR=$(pwd)/cuda
export LD_LIBRARY_PATH=$CUDNN_DIR/lib64:$LD_LIBRARY_PATH

# 3.2 install ONNX Runtime
# you can install one to install according whether you need gpu inference
# 3.2.1 onnxruntime
wget https://github.com/microsoft/onnxruntime/releases/download/v1.8.1/onnxruntime-linux-
x64-1.8.1.tgz
tar -zxvf onnxruntime-linux-x64-1.8.1.tgz
export ONNXRUNTIME_DIR=$(pwd)/onnxruntime-linux-x64-1.8.1
export LD_LIBRARY_PATH=$ONNXRUNTIME_DIR/lib:$LD_LIBRARY_PATH
# 3.2.2 onnxruntime-gpu
pip install onnxruntime-gpu==1.8.1
wget https://github.com/microsoft/onnxruntime/releases/download/v1.8.1/onnxruntime-linux-
x64-gpu-1.8.1.tgz
tar -zxvf onnxruntime-linux-x64-gpu-1.8.1.tgz
export ONNXRUNTIME_DIR=$(pwd)/onnxruntime-linux-x64-gpu-1.8.1
export LD_LIBRARY_PATH=$ONNXRUNTIME_DIR/lib:$LD_LIBRARY_PATH

```

Please learn its prebuilt package from this guide.

1.4 Convert Model

After the installation, you can enjoy the model deployment journey starting from converting PyTorch model to backend model by running tools/deploy.py.

Based on the above settings, we provide an example to convert the Faster R-CNN in MMDetection to TensorRT as below:

```

# clone mmdemo to get the deployment config. `--recursive` is not necessary
git clone https://github.com/open-mmlab/mmdemo.git

```

(continues on next page)

(continued from previous page)

```
# clone mmdetection repo. We have to use the config file to build PyTorch nn module
git clone https://github.com/open-mmlab/mmdetection.git
cd mmdetection
pip install -v -e .
cd ..

# download Faster R-CNN checkpoint
wget -P checkpoints https://download.openmmlab.com/mmdetection/v2.0/faster_rcnn/faster_
rcnn_r50_fpn_1x_coco/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth

# run the command to start model conversion
python mmdeploy/tools/deploy.py \
    mmdeploy/configs/mmdet/detection/detection_tensorrt_dynamic-320x320-1344x1344.py \
    mmdetection/configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py \
    checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth \
    mmdetection/demo/demo.jpg \
    --work-dir mmdeploy_model/faster-rcnn \
    --device cuda \
    --dump-info
```

The converted model and its meta info will be found in the path specified by `--work-dir`. And they make up of MMDeploy Model that can be fed to MMDeploy SDK to do model inference.

For more details about model conversion, you can read [how_to_convert_model](#). If you want to customize the conversion pipeline, you can edit the config file by following [this](#) tutorial.

Tip: You can convert the above model to onnx model and perform ONNX Runtime inference just by changing ‘detection_tensorrt_dynamic-320x320-1344x1344.py’ to ‘detection_onnxruntime_dynamic.py’ and making ‘--device’ as ‘cpu’.

1.5 Inference Model

After model conversion, we can perform inference not only by Model Converter but also by Inference SDK.

1.5.1 Inference by Model Converter

Model Converter provides a unified API named as `inference_model` to do the job, making all inference backends API transparent to users. Take the previous converted Faster R-CNN tensorrt model for example,

```
from mmdeploy.apis import inference_model
result = inference_model(
    model_cfg='mmdetection/configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py',
    deploy_cfg='mmdeploy/configs/mmdet/detection/detection_tensorrt_dynamic-320x320-
1344x1344.py',
    backend_files=['mmdeploy_model/faster-rcnn/end2end.engine'],
    img='mmdetection/demo/demo.jpg',
    device='cuda:0')
```

Note: ‘backend_files’ in this API refers to backend engine file path, which MUST be put in a list, since some inference engines like OpenVINO and ncnn separate the network structure and its weights into two files.

1.5.2 Inference by SDK

You can directly run MMDeploy demo programs in the precompiled package to get inference results.

```
wget https://github.com/open-mmlab/mmddeploy/releases/download/v0.14.0/mmddeploy-0.14.0-
↳ linux-x86_64-cuda11.3.tar.gz
tar xf mmddeploy-0.14.0-linux-x86_64-cuda11.3
cd mmddeploy-0.14.0-linux-x86_64-cuda11.3
# run python demo
python example/python/object_detection.py cuda ../mmddeploy_model/faster-rcnn ../
↳ mmdetection/demo/demo.jpg
# run C/C++ demo
# build the demo according to the README.md in the folder.
./bin/object_detection cuda ../mmddeploy_model/faster-rcnn ../mmdetection/demo/demo.jpg
```

Note: In the above command, the input model is SDK Model path. It is NOT engine file path but actually the path passed to -work-dir. It not only includes engine files but also meta information like ‘deploy.json’ and ‘pipeline.json’.

In the next section, we will provide examples of deploying the converted Faster R-CNN model talked above with SDK different FFI (Foreign Function Interface).

Python API

```
from mmddeploy_runtime import Detector
import cv2

img = cv2.imread('mmdetection/demo/demo.jpg')

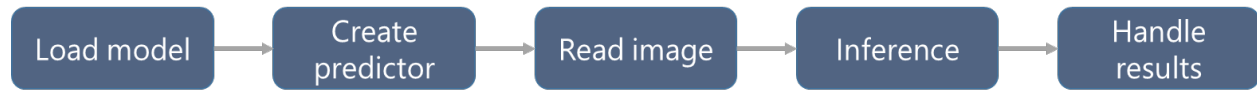
# create a detector
detector = Detector(model_path='mmddeploy_models/faster-rcnn', device_name='cuda', device_
↳ id=0)
# run the inference
bboxes, labels, _ = detector(img)
# Filter the result according to threshold
indices = [i for i in range(len(bboxes))]
for index, bbox, label_id in zip(indices, bboxes, 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('output_detection.png', img)
```

You can find more examples from [here](#).

C++ API

Using SDK C++ API should follow next pattern,



Now let's apply this procedure on the above Faster R-CNN model.

```

#include <cstdlib>
#include <opencv2/opencv.hpp>
#include "mmdet/detector.hpp"

int main() {
    const char* device_name = "cuda";
    int device_id = 0;
    std::string model_path = "mmdet_model/faster-rcnn";
    std::string image_path = "mmdet_demo/demo.jpg";

    // 1. load model
    mmdet::Model model(model_path);
    // 2. create predictor
    mmdet::Detector detector(model, mmdet::Device{device_name, device_id});
    // 3. read image
    cv::Mat img = cv::imread(image_path);
    // 4. inference
    auto dets = detector.Apply(img);
    // 5. deal with the result. Here we choose to visualize it
    for (int i = 0; i < dets.size(); ++i) {
        const auto& box = dets[i].bbox;
        fprintf(stdout, "box %d, left=%.2f, top=%.2f, right=%.2f, bottom=%.2f, label=%d, \n",
            i, box.left, box.top, box.right, box.bottom, dets[i].label_id,
            dets[i].score);
        if (dets[i].score < 0.3) {
            continue;
        }
        cv::rectangle(img, cv::Point{(int)box.left, (int)box.top},
            cv::Point{(int)box.right, (int)box.bottom}, cv::Scalar{0, 255, 0});
    }
    cv::imwrite("output_detection.png", img);
    return 0;
}

```

When you build this example, try to add MMDeploy package in your CMake project as following. Then pass `-DMMDeploy_DIR` to cmake, which indicates the path where `MMDeployConfig.cmake` locates. You can find it in the prebuilt package.

```

find_package(MMDeploy REQUIRED)
target_link_libraries(${name} PRIVATE mmdet ${OpenCV_LIBS})

```

For more SDK C++ API usages, please read these [samples](#).

For the rest C, C# and Java API usages, please read [C demos](#), [C# demos](#) and [Java demos](#) respectively. We'll talk about

them more in our next release.

Accelerate preprocessingExperimental

If you want to fuse preprocess for acceleration please refer to this doc

1.6 Evaluate Model

You can test the performance of deployed model using `tool/test.py`. For example,

```
python ${MMDEPLOY_DIR}/tools/test.py \  
    ${MMDEPLOY_DIR}/configs/detection/detection_tensorrt_dynamic-320x320-1344x1344.py \  
    ${MMDT_DIR}/configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py \  
    --model ${BACKEND_MODEL_FILES} \  
    --metrics ${METRICS} \  
    --device cuda:0
```

Note: Regarding the `--model` option, it represents the converted engine files path when using Model Converter to do performance test. But when you try to test the metrics by Inference SDK, this option refers to the directory path of MMDeploy Model.

You can read [how to evaluate a model](#) for more details.

BUILD FROM SOURCE

2.1 Download

```
git clone -b master git@github.com:open-mmlab/mmddeploy.git --recursive
```

Note:

- If fetching submodule fails, you could get submodule manually by following instructions:

```
cd mmddeploy
git clone git@github.com:NVIDIA/cub.git third_party/cub
cd third_party/cub
git checkout c3cceac115

# go back to third_party directory and git clone pybind11
cd ..
git clone git@github.com:pybind/pybind11.git pybind11
cd pybind11
git checkout 70a58c5
```

- If it fails when `git clone` via SSH, you can try the HTTPS protocol like this:

```
git clone -b master https://github.com/open-mmlab/mmddeploy.git --recursive
```

2.2 Build

Please visit the following links to find out how to build MMDeploy according to the target platform.

- [Linux-x86_64](#)
- [Windows](#)
- [Android-aarch64](#)
- [NVIDIA Jetson](#)
- [SNPE](#)
- [RISC-V](#)
- [Rockchip](#)

USE DOCKER IMAGE

We provide two dockerfiles for CPU and GPU respectively. For CPU users, we install MMDeploy with ONNXRuntime, ncnn and OpenVINO backends. For GPU users, we install MMDeploy with TensorRT backend. Besides, users can install mmdeploy with different versions when building the docker image.

3.1 Build docker image

For CPU users, we can build the docker image with the latest MMDeploy through:

```
cd mmdeploy
docker build docker/CPU/ -t mmdeploy:master-cpu
```

For GPU users, we can build the docker image with the latest MMDeploy through:

```
cd mmdeploy
docker build docker/GPU/ -t mmdeploy:master-gpu
```

For installing MMDeploy with a specific version, we can append `--build-arg VERSION=${VERSION}` to build command. GPU for example:

```
cd mmdeploy
docker build docker/GPU/ -t mmdeploy:0.1.0 --build-arg VERSION=0.1.0
```

For installing libs with the aliyun source, we can append `--build-arg USE_SRC_INSIDE=${USE_SRC_INSIDE}` to build command.

```
# GPU for example
cd mmdeploy
docker build docker/GPU/ -t mmdeploy:inside --build-arg USE_SRC_INSIDE=true

# CPU for example
cd mmdeploy
docker build docker/CPU/ -t mmdeploy:inside --build-arg USE_SRC_INSIDE=true
```

3.2 Run docker container

After building the docker image succeed, we can use `docker run` to launch the docker service. GPU docker image for example:

```
docker run --gpus all -it mmdeploy:master-gpu
```

3.3 FAQs

1. CUDA error: the provided PTX was compiled with an unsupported toolchain:

As described [here](#), update the GPU driver to the latest one for your GPU.

2. docker: Error response from daemon: could not select device driver “” with capabilities: [gpu].

```
# Add the package repositories
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list_
→| sudo tee /etc/apt/sources.list.d/nvidia-docker.list

sudo apt-get update && sudo apt-get install -y nvidia-container-toolkit
sudo systemctl restart docker
```

BUILD FROM SCRIPT

Through user investigation, we know that most users are already familiar with python and torch before using mmdeploy. Therefore we provide scripts to simplify mmdeploy installation.

Assuming you already have

- python3 -m pip (conda or pyenv)
- nvcc (depends on inference backend)
- torch (not compulsory)

run this script to install mmdeploy + ncnn backend, nproc is not compulsory.

```
$ cd /path/to/mmdelay
$ python3 tools/scripts/build_ubuntu_x64_ncnn.py
..
```

A sudo password may be required during this time, and the script will try its best to build and install mmdeploy SDK and demo:

- Detect host OS version, make job number, whether use root and try to fix python3 -m pip
- Find the necessary basic tools, such as g++-7, cmake, wget, etc.
- Compile necessary dependencies, such as pycnn, protobuf

The script will also try to avoid affecting host environment:

- The dependencies of source code compilation are placed in the mmdeploy-dep directory at the same level as mmdeploy
- The script would not modify variables such as PATH, LD_LIBRARY_PATH, PYTHONPATH, etc.
- The environment variables that need to be modified will be printed, **please pay attention to the final output**

The script will eventually execute python3 tools/check_env.py, the successful installation should display the version number of the corresponding backend and ops_is_available: True, for example:

```
$ python3 tools/check_env.py
..
2022-09-13 14:49:13,767 - mmdeploy - INFO - *****Backend information*****
2022-09-13 14:49:14,116 - mmdeploy - INFO - onnxruntime: 1.8.0 ops_is_available :_
↪True
2022-09-13 14:49:14,131 - mmdeploy - INFO - tensorrt: 8.4.1.5 ops_is_available :_
↪True
2022-09-13 14:49:14,139 - mmdeploy - INFO - ncnn: 1.0.20220901 ops_is_available :_
↪True
```

(continues on next page)

(continued from previous page)

```
2022-09-13 14:49:14,150 - mmdeploy - INFO - pplnn_is_avaliabile: True
..
```

Here is the verified installation script. If you want mmdeploy to support multiple backends at the same time, you can execute each script once:

CMAKE BUILD OPTION SPEC

HOW TO CONVERT MODEL

This tutorial briefly introduces how to export an OpenMMLab model to a specific backend using MMDeploy tools.
Notes:

- Supported backends are *ONNXRuntime*, *TensorRT*, *ncnn*, *PPLNN*, *OpenVINO*.
- Supported codebases are *MMClassification*, *MMDetection*, *MMSegmentation*, *MMOCR*, *MMEdition*.

6.1 How to convert models from Pytorch to other backends

6.1.1 Prerequisite

1. Install and build your target backend. You could refer to *ONNXRuntime-install*, *TensorRT-install*, *ncnn-install*, *PPLNN-install*, *OpenVINO-install* for more information.
2. Install and build your target codebase. You could refer to *MMClassification-install*, *MMDetection-install*, *MMSegmentation-install*, *MMOCR-install*, *MMEdition-install*.

6.1.2 Usage

```
python ./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
```

6.1.3 Description of all arguments

- `deploy_cfg` : The deployment configuration of mmdet 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, `mmdeploy/mmcv/classification_ncnn_static.py` is an example.
- `model_cfg` : Model configuration for algorithm library, e.g. `mmclassification/configs/vision_transformer/vit-base-p32_ft-64xb64_in1k-384.py`, regardless of the path to mmdet.
- `checkpoint` : torch model path. It can start with http/https, see the implementation of `mmcv.FileClient` for details.
- `img` : The path to the image or point cloud file used for testing during model conversion.
- `--test-img` : The path of image file that used to test model. If not specified, it will be set to `None`.
- `--work-dir` : The path of work directory that used to save logs and models.
- `--calib-dataset-cfg` : Only valid in int8 mode. Config used for calibration. If not specified, it will be set to `None` and use “val” dataset in model config for calibration.
- `--device` : The device used for model conversion. If not specified, it will be set to `cpu`, for trt use `cuda:0` format.
- `--log-level` : To set log level which in 'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'. If not specified, it will be set to `INFO`.
- `--show` : Whether to show detection outputs.
- `--dump-info` : Whether to output information for SDK.

6.1.4 How to find the corresponding deployment config of a PyTorch model

1. Find model's codebase folder in `configs/`. Example, convert a yolov3 model you need to find `configs/mmdet` folder.
2. Find model's task folder in `configs/codebase_folder/`. Just like yolov3 model, you need to find `configs/mmdet/detection` folder.
3. Find deployment config file in `configs/codebase_folder/task_folder/`. Just like deploy yolov3 model you can use `configs/mmdet/detection/detection_onnxruntime_dynamic.py`.

6.1.5 Example

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

6.2 How to evaluate the exported models

You can try to evaluate model, referring to *how_to_evaluate_a_model*.

6.3 List of supported models exportable to other backends

Refer to *Support model list*

HOW TO WRITE CONFIG

This tutorial describes how to write a config for model conversion and deployment. A deployment config includes `onnx config`, `codebase config`, `backend config`.

- *How to write config*
 - 1. *How to write onnx config*
 - * *Description of onnx config arguments*
 - *Example*
 - * *If you need to use dynamic axes*
 - *Example*
 - 2. *How to write codebase config*
 - * *Description of codebase config arguments*
 - *Example*
 - 3. *How to write backend config*
 - * *Example*
 - 4. *A complete example of mmcls on TensorRT*
 - 5. *The name rules of our deployment config*
 - * *Example*
 - 6. *How to write model config*
 - 7. *Reminder*
 - 8. *FAQs*

7.1 1. How to write onnx config

Onnx config to describe how to export a model from pytorch to onnx.

7.1.1 Description of onnx config arguments

- `type`: Type of config dict. Default is `onnx`.
- `export_params`: If specified, all parameters will be exported. Set this to `False` if you want to export an untrained model.
- `keep_initializers_as_inputs`: If `True`, all the initializers (typically corresponding to parameters) in the exported graph will also be added as inputs to the graph. If `False`, then initializers are not added as inputs to the graph, and only the non-parameter inputs are added as inputs.
- `opset_version`: Opset_version is 11 by default.
- `save_file`: Output onnx file.
- `input_names`: Names to assign to the input nodes of the graph.
- `output_names`: Names to assign to the output nodes of the graph.
- `input_shape`: The height and width of input tensor to the model.

7.1.2 Example

```
onnx_config = dict(  
    type='onnx',  
    export_params=True,  
    keep_initializers_as_inputs=False,  
    opset_version=11,  
    save_file='end2end.onnx',  
    input_names=['input'],  
    output_names=['output'],  
    input_shape=None)
```

7.1.3 If you need to use dynamic axes

If the dynamic shape of inputs and outputs is required, you need to add `dynamic_axes` dict in onnx config.

- `dynamic_axes`: Describe the dimensional information about input and output.

Example

```
dynamic_axes={  
    'input': {  
        0: 'batch',  
        2: 'height',  
        3: 'width'  
    },  
    'dets': {  
        0: 'batch',  
        1: 'num_dets',  
    },  
    'labels': {  
        0: 'batch',  
        1: 'num_dets',  
    },  
}
```

(continues on next page)

(continued from previous page)

```
    },
}
```

7.2 2. How to write codebase config

Codebase config part contains information like codebase type and task type.

7.2.1 Description of codebase config arguments

- `type`: Model's codebase, including `mmcls`, `mmdet`, `mmseg`, `mmocr`, `mmedit`.
- `task`: Model's task type, referring to List of tasks in all codebases.

Example

```
codebase_config = dict(type='mmcls', task='Classification')
```

7.3 3. How to write backend config

The backend config is mainly used to specify the backend on which model runs and provide the information needed when the model runs on the backend, referring to [ONNX Runtime](#), [TensorRT](#), [ncnn](#), [PPLNN](#).

- `type`: Model's backend, including `onnxruntime`, `ncnn`, `pplnn`, `tensorrt`, `openvino`.

7.3.1 Example

```
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, 512, 1024],
                    opt_shape=[1, 3, 1024, 2048],
                    max_shape=[1, 3, 2048, 2048]))))
    ])
```

7.4 4. A complete example of mmcls on TensorRT

Here we provide a complete deployment config from mmcls on TensorRT.

```
codebase_config = dict(type='mmcls', task='Classification')

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, 224, 224],
                    opt_shape=[4, 3, 224, 224],
                    max_shape=[64, 3, 224, 224]))))]

onnx_config = dict(
    type='onnx',
    dynamic_axes={
        'input': {
            0: 'batch',
            2: 'height',
            3: 'width'
        },
        'output': {
            0: 'batch'
        }
    },
    export_params=True,
    keep_initializers_as_inputs=False,
    opset_version=11,
    save_file='end2end.onnx',
    input_names=['input'],
    output_names=['output'],
    input_shape=[224, 224])
```

7.5 5. The name rules of our deployment config

There is a specific naming convention for the filename of deployment config files.

```
(task name)_(backend name)_(dynamic or static).py
```

- **task name**: Model's task type.
- **backend name**: Backend's name. Note if you use the quantization function, you need to indicate the quantization type. Just like `tensorrt-int8`.
- **dynamic or static**: Dynamic or static export. Note if the backend needs explicit shape information, you need to add a description of input size with `height x width` format. Just like `dynamic-512x1024-2048x2048`, it

means that the min input shape is 512x1024 and the max input shape is 2048x2048.

7.5.1 Example

```
detection_tensorrt-int8_dynamic-320x320-1344x1344.py
```

7.6 6. How to write model config

According to model's codebase, write the model config file. Model's config file is used to initialize the model, referring to [MMClassification](#), [MMDetection](#), [MMSegmentation](#), [MMOCR](#), [MMEediting](#).

HOW TO EVALUATE MODEL

After converting a PyTorch model to a backend model, you may evaluate backend models with `tools/test.py`

8.1 Prerequisite

Install MMDeploy according to *get-started* instructions. And convert the PyTorch model or ONNX model to the backend model by following the *guide*.

8.2 Usage

```
python tools/test.py \
  ${DEPLOY_CFG} \
  ${MODEL_CFG} \
  --model ${BACKEND_MODEL_FILES} \
  [--out ${OUTPUT_PKL_FILE}] \
  [--format-only] \
  [--metrics ${METRICS}] \
  [--show] \
  [--show-dir ${OUTPUT_IMAGE_DIR}] \
  [--show-score-thr ${SHOW_SCORE_THR}] \
  --device ${DEVICE} \
  [--cfg-options ${CFG_OPTIONS}] \
  [--metric-options ${METRIC_OPTIONS}]
  [--log2file work_dirs/output.txt]
  [--batch-size ${BATCH_SIZE}]
  [--speed-test] \
  [--warmup ${WARM_UP}] \
  [--log-interval ${LOG_INTERVERL}] \
```

8.3 Description of all arguments

- `deploy_cfg`: The config for deployment.
- `model_cfg`: The config of the model in OpenMMLab codebases.
- `--model`: The backend model file. For example, if we convert a model to TensorRT, we need to pass the model file with “.engine” suffix.
- `--out`: The path to save output results in pickle format. (The results will be saved only if this argument is given)
- `--format-only`: Whether format the output results without evaluation or not. It is useful when you want to format the result to a specific format and submit it to the test server
- `--metrics`: The metrics to evaluate the model defined in OpenMMLab codebases. e.g. “segm”, “proposal” for COCO in mmdet, “precision”, “recall”, “f1_score”, “support” for single label dataset in mmcls.
- `--show`: Whether to show the evaluation result on the screen.
- `--show-dir`: The directory to save the evaluation result. (The results will be saved only if this argument is given)
- `--show-score-thr`: The threshold determining whether to show detection bounding boxes.
- `--device`: The device that the model runs on. Note that some backends restrict the device. For example, TensorRT must run on cuda.
- `--cfg-options`: Extra or overridden settings that will be merged into the current deploy config.
- `--metric-options`: Custom options for evaluation. The key-value pair in `xxx=yyy` format will be kwargs for `dataset.evaluate()` function.
- `--log2file`: log evaluation results (and speed) to file.
- `--batch-size`: the batch size for inference, which would override `samples_per_gpu` in data config. Default is 1. Note that not all models support `batch_size>1`.
- `--speed-test`: Whether to activate speed test.
- `--warmup`: warmup before counting inference elapse, require setting speed-test first.
- `--log-interval`: The interval between each log, require setting speed-test first.
- `--json-file`: The path of json file to save evaluation results. Default is `./results.json`.

* Other arguments in `tools/test.py` are used for speed test. They have no concern with evaluation.

8.4 Example

```
python tools/test.py \
  configs/mmdet/classification_onnxruntime_static.py \
  {MMCLS_DIR}/configs/resnet/resnet50_b32x8_imagenet.py \
  --model model.onnx \
  --out out.pkl \
  --device cpu \
  --speed-test
```

8.5 Note

- The performance of each model in [OpenMMLab](#) codebases can be found in the document of each codebase.

QUANTIZE MODEL

9.1 Why quantization ?

The fixed-point model has many advantages over the fp32 model:

- Smaller size, 8-bit model reduces file size by 75%
- Benefit from the smaller model, the Cache hit rate is improved and inference would be faster
- Chips tend to have corresponding fixed-point acceleration instructions which are faster and less energy consumed (int8 on a common CPU requires only about 10% of energy)

APK file size and heat generation are key indicators while evaluating mobile APP; On server side, quantization means that you can increase model size in exchange for precision and keep the same QPS.

9.2 Post training quantization scheme

Taking ncnn backend as an example, the complete workflow is as follows:

mmdeploy generates quantization table based on static graph (onnx) and uses backend tools to convert fp32 model to fixed point.

mmdeploy currently support ncnn with PTQ.

9.3 How to convert model

After mmdeploy installation, install ppq

```
git clone https://github.com/openppl-public/ppq.git
cd ppq
pip install -r requirements.txt
python3 setup.py install
```

Back in mmdeploy, enable quantization with the option 'tools/deploy.py --quant'.

```
cd /path/to/mmdploy

export MODEL_CONFIG=/home/rg/konghuanjun/miclassification/configs/resnet/resnet18_8xb32_
↪in1k.py
export MODEL_PATH=https://download.openmmlab.com/miclassification/v0/resnet/resnet18_
↪8xb32_in1k_20210831-fbbb1da6.pth
```

(continues on next page)

(continued from previous page)

```
# get some imagenet sample images
git clone https://github.com/nihui/imagenet-sample-images --depth=1

# quantize
python3 tools/deploy.py configs/mmccls/classification_ncnn-int8_static.py ${MODEL_
↪CONFIG} ${MODEL_PATH} /path/to/self-test.png --work-dir work_dir --device cpu --
↪quant --quant-image-dir /path/to/imagenet-sample-images
...
```

Description

9.4 Custom calibration dataset

Calibration set is used to calculate quantization layer parameters. Some DFQ (Data Free Quantization) methods do not even require a dataset.

- Create a folder, just put in some images (no directory structure, no negative example, no special filename format)
- The image needs to be the data comes from real scenario otherwise the accuracy would be drop
- You can not quantize model with test dataset

Type	Train dataset	Validation dataset	Test dataset	Calibration dataset
Usage	QAT	PTQ	Test accuracy	PTQ

It is highly recommended that *verifying model precision* after quantization. *Here* is some quantization model test result.

USEFUL TOOLS

Apart from `deploy.py`, there are other useful tools under the `tools/` directory.

10.1 torch2onnx

This tool can be used to convert PyTorch model from OpenMMLab to ONNX.

10.1.1 Usage

```
python tools/torch2onnx.py \
    ${DEPLOY_CFG} \
    ${MODEL_CFG} \
    ${CHECKPOINT} \
    ${INPUT_IMG} \
    --work-dir ${WORK_DIR} \
    --device cpu \
    --log-level INFO
```

10.1.2 Description of all arguments

- `deploy_cfg` : The path of the deploy config file in MMDeploy codebase.
- `model_cfg` : The path of model config file in OpenMMLab codebase.
- `checkpoint` : The path of the model checkpoint file.
- `img` : The path of the image file used to convert the model.
- `--work-dir` : Directory to save output ONNX models Default is `./work-dir`.
- `--device` : The device used for conversion. If not specified, it will be set to `cpu`.
- `--log-level` : To set log level which in 'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'. If not specified, it will be set to `INFO`.

10.2 extract

ONNX model with Mark nodes in it can be partitioned into multiple subgraphs. This tool can be used to extract the subgraph from the ONNX model.

10.2.1 Usage

```
python tools/extract.py \  
    ${INPUT_MODEL} \  
    ${OUTPUT_MODEL} \  
    --start ${PARTITION_START} \  
    --end ${PARTITION_END} \  
    --log-level INFO
```

10.2.2 Description of all arguments

- `input_model` : The path of input ONNX model. The output ONNX model will be extracted from this model.
- `output_model` : The path of output ONNX model.
- `--start` : The start point of extracted model with format `<function_name>:<input/output>`. The `function_name` comes from the decorator `@mark`.
- `--end` : The end point of extracted model with format `<function_name>:<input/output>`. The `function_name` comes from the decorator `@mark`.
- `--log-level` : To set log level which in 'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'. If not specified, it will be set to INFO.

10.2.3 Note

To support the model partition, you need to add Mark nodes in the ONNX model. The Mark node comes from the `@mark` decorator. For example, if we have marked the `multiclass_nms` as below, we can set `end=multiclass_nms:input` to extract the subgraph before NMS.

```
@mark('multiclass_nms', inputs=['boxes', 'scores'], outputs=['dets', 'labels'])  
def multiclass_nms(*args, **kwargs):  
    """Wrapper function for `_multiclass_nms`."""
```

10.3 onnx2pplnn

This tool helps to convert an ONNX model to an PPLNN model.

10.3.1 Usage

```
python tools/onnx2pplnn.py \
    ${ONNX_PATH} \
    ${OUTPUT_PATH} \
    --device cuda:0 \
    --opt-shapes [224,224] \
    --log-level INFO
```

10.3.2 Description of all arguments

- `onnx_path`: The path of the ONNX model to convert.
- `output_path`: The converted PPLNN algorithm path in json format.
- `device`: The device of the model during conversion.
- `opt-shapes`: Optimal shapes for PPLNN optimization. The shape of each tensor should be wrap with “[]” or “()” and the shapes of tensors should be separated by “,”.
- `--log-level`: To set log level which in 'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'. If not specified, it will be set to INFO.

10.4 onnx2tensorrt

This tool can be used to convert ONNX to TensorRT engine.

10.4.1 Usage

```
python tools/onnx2tensorrt.py \
    ${DEPLOY_CFG} \
    ${ONNX_PATH} \
    ${OUTPUT} \
    --device-id 0 \
    --log-level INFO \
    --calib-file /path/to/file
```

10.4.2 Description of all arguments

- `deploy_cfg`: The path of the deploy config file in MMDeploy codebase.
- `onnx_path`: The ONNX model path to convert.
- `output`: The path of output TensorRT engine.
- `--device-id`: The device index, default to 0.
- `--calib-file`: The calibration data used to calibrate engine to int8.
- `--log-level`: To set log level which in 'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'. If not specified, it will be set to INFO.

10.5 onnx2ncnn

This tool helps to convert an ONNX model to an ncnn model.

10.5.1 Usage

```
python tools/onnx2ncnn.py \  
    ${ONNX_PATH} \  
    ${NCNN_PARAM} \  
    ${NCNN_BIN} \  
    --log-level INFO
```

10.5.2 Description of all arguments

- onnx_path : The path of the ONNX model to convert from.
- output_param : The converted ncnn param path.
- output_bin : The converted ncnn bin path.
- --log-level : To set log level which in 'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'. If not specified, it will be set to INFO.

10.6 profiler

This tool helps to test latency of models with PyTorch, TensorRT and other backends. Note that the pre- and post-processing is excluded when computing inference latency.

10.6.1 Usage

```
python tools/profiler.py \  
    ${DEPLOY_CFG} \  
    ${MODEL_CFG} \  
    ${IMAGE_DIR} \  
    --model ${MODEL} \  
    --device ${DEVICE} \  
    --shape ${SHAPE} \  
    --num-iter ${NUM_ITER} \  
    --warmup ${WARMUP} \  
    --cfg-options ${CFG_OPTIONS} \  
    --batch-size ${BATCH_SIZE} \  
    --img-ext ${IMG_EXT}
```

10.6.2 Description of all arguments

- `deploy_cfg` : The path of the deploy config file in MMDeploy codebase.
- `model_cfg` : The path of model config file in OpenMMLab codebase.
- `image_dir` : The directory to image files that used to test the model.
- `--model` : The path of the model to be tested.
- `--shape` : Input shape of the model by HxW, e.g., 800x1344. If not specified, it would use `input_shape` from deploy config.
- `--num-iter` : Number of iteration to run inference. Default is 100.
- `--warmup` : Number of iteration to warm-up the machine. Default is 10.
- `--device` : The device type. If not specified, it will be set to `cuda:0`.
- `--cfg-options` : Optional key-value pairs to be overrode for model config.
- `--batch-size` : the batch size for test inference. Default is 1. Note that not all models support `batch_size>1`.
- `--img-ext` : the file extensions for input images from `image_dir`. Defaults to `['.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif']`.

10.6.3 Example:

```
python tools/profiler.py \
  configs/mmccls/classification_tensorrt_dynamic-224x224-224x224.py \
  ../mmclassification/configs/resnet/resnet18_8xb32_in1k.py \
  ../mmclassification/demo/ \
  --model work-dirs/mmccls/resnet/trt/end2end.engine \
  --device cuda \
  --shape 224x224 \
  --num-iter 100 \
  --warmup 10 \
  --batch-size 1
```

And the output look like this:

```
----- Settings:
+-----+
| batch size |    1    |
|   shape   | 224x224 |
| iterations |   100   |
|  warmup   |    10   |
+-----+
----- Results:
+-----+-----+
| Stats | Latency/ms |   FPS   |
+-----+-----+
| Mean  |    1.535   | 651.656 |
| Median |    1.665   | 600.569 |
| Min   |    1.308   | 764.341 |
| Max   |    1.689   | 591.983 |
+-----+-----+
```

10.7 generate_md_table

This tool can be used to generate supported-backends markdown table.

10.7.1 Usage

```
python tools/generate_md_table.py \
    ${yml_file} \
    ${output} \
    ${backends}
```

10.7.2 Description of all arguments

- `yml_file`: input yml config path
- `output`: output markdown file path
- `backends`: output backends list. If not specified, it will be set 'onnxruntime' 'tensorrt' 'torchscript' 'pplnn' 'openvino' 'ncnn'.

10.7.3 Example:

Generate backends markdown table from mmocr.yml

```
python tools/generate_md_table.py tests/regression/mmocr.yml tests/regression/mmocr.md
↪ onnxruntime tensorrt torchscript pplnn openvino ncnn
```

And the output look like this:

SUPPORTED MODELS

The table below lists the models that are guaranteed to be exportable to other backends.

11.1 Note

- Tag:
 - static: This model only support static export. Please use static deploy config, just like `$MMDEPLOY_DIR/configs/mmdet/segmentation_tensorrt_static-1024x2048.py`.
- SSD: When you convert SSD model, you need to use min shape deploy config just like `300x300-512x512` rather than `320x320-1344x1344`, for example `$MMDEPLOY_DIR/configs/mmdet/detection_tensorrt_dynamic-300x300-512x512.py`.
- YOLOX: YOLOX with ncnn only supports static shape.
- Swin Transformer: For TensorRT, only version 8.4+ is supported.
- SAR: Chinese text recognition model is not supported as the protobuf size of ONNX is limited.

BENCHMARK

12.1 Backends

CPU: ncnn, ONNXRuntime, OpenVINO

GPU: ncnn, TensorRT, PPLNN

12.2 Latency benchmark

12.2.1 Platform

- Ubuntu 18.04
- ncnn 20211208
- Cuda 11.3
- TensorRT 7.2.3.4
- Docker 20.10.8
- NVIDIA tesla T4 tensor core GPU for TensorRT

12.2.2 Other settings

- Static graph
- Batch size 1
- Synchronize devices after each inference.
- We count the average inference performance of 100 images of the dataset.
- Warm up. For ncnn, we warm up 30 iters for all codebases. As for other backends: for classification, we warm up 1010 iters; for other codebases, we warm up 10 iters.
- Input resolution varies for different datasets of different codebases. All inputs are real images except for `meditating` because the dataset is not large enough.

Users can directly test the speed through *model profiling*. And here is the benchmark in our environment.

12.3 Performance benchmark

Users can directly test the performance through [how_to_evaluate_a_model.md](#). And here is the benchmark in our environment.

12.4 Notes

- As some datasets contain images with various resolutions in codebase like MMDet. The speed benchmark is gained through static configs in MMDeploy, while the performance benchmark is gained through dynamic ones.
- Some int8 performance benchmarks of TensorRT require Nvidia cards with tensor core, or the performance would drop heavily.
- DBNet uses the interpolate mode `nearest` in the neck of the model, which TensorRT-7 applies a quite different strategy from Pytorch. To make the repository compatible with TensorRT-7, we rewrite the neck to use the interpolate mode `bilinear` which improves final detection performance. To get the matched performance with Pytorch, TensorRT-8+ is recommended, which the interpolate methods are all the same as Pytorch.
- Mask AP of Mask R-CNN drops by 1% for the backend. The main reason is that the predicted masks are directly interpolated to original image in PyTorch, while they are at first interpolated to the preprocessed input image of the model and then to original image in other backends.
- MMPose models are tested with `flip_test` explicitly set to `False` in model configs.
- Some models might get low accuracy in fp16 mode. Please adjust the model to avoid value overflow.

TEST ON EMBEDDED DEVICE

Here are the test conclusions of our edge devices. You can directly obtain the results of your own environment with *model profiling*.

13.1 Software and hardware environment

- host OS ubuntu 18.04
- backend SNPE-1.59
- device Mi11 (qcom 888)

13.2 mmcls

tips:

1. The ImageNet-1k dataset is too large to test, only part of the dataset is used (8000/50000)
2. The heating of device will downgrade the frequency, so the time consumption will actually fluctuate. Here are the stable values after running for a period of time. This result is closer to the actual demand.

13.3 mmocr detection

13.4 mmpose

tips:

- Test pose_hrnet using AnimalPose's test dataset instead of val dataset.

13.5 mmseg

tips:

- fcn works fine with 512x1024 size. Cityscapes dataset uses 1024x2048 resolution which causes device to reboot.

13.6 Notes

- We need to manually split the mmdet model into two parts. Because
 - In snpe source code, `onnx_to_ir.py` can only parse onnx input while `ir_to_dlc.py` does not support `topk` operator
 - UDO (User Defined Operator) does not work with `snpe-onnx-to-dlc`
- mmedit model
 - `srcnn` requires cubic resize which snpe does not support
 - `esrgan` converts fine, but loading the model causes the device to reboot
- mmrotate depends on `e2cnn` and needs to be installed manually [its Python3.6 compatible branch](#)

TEST ON TVM

14.1 Supported Models

The table above list the models that we have tested. Models not listed on the table might still be able to converted. Please have a try.

14.2 Test

- Ubuntu 20.04
- tvn 0.9.0

*: We only test model on ssd since dynamic shape is not supported for now.

QUANTIZATION TEST RESULT

Currently mmdeploy support ncnn quantization

15.1 Quantize with ncnn

15.1.1 mmcls

Note:

- Because of the large amount of imagenet-1k data and ncnn has not released Vulkan int8 version, only part of the test set (4000/50000) is used.
- The accuracy will vary after quantization, and it is normal for the classification model to increase by less than 1%.

15.1.2 OCR detection

Note: `mmocr` Uses 'shapely' to compute IoU, which results in a slight difference in accuracy

15.1.3 Pose detection

Note: MMPose models are tested with `flip_test` explicitly set to `False` in model configs.

15.1.4 Super Resolution

15.1.5 mmseg

Note:

- Int8 models of the Fast-SCNN requires `ncnnoptimize`.
- NCNN will extract 512 images from the train as a calibration dataset

MMCLASSIFICATION SUPPORT

`MMClassification` is an open-source image classification toolbox based on PyTorch. It is a part of the `OpenMMLab` project.

16.1 MMClassification installation tutorial

Please refer to [install.md](#) for installation.

16.2 List of MMClassification models supported by MMDeploy

MMDETECTION SUPPORT

MMDetection is an open source object detection toolbox based on PyTorch. It is a part of the [OpenMMLab](#) project.

17.1 MMDetection installation tutorial

Please refer to [get_started.md](#) for installation.

17.2 List of MMDetection models supported by MMDeploy

MMSEGMENTATION SUPPORT

MMSegmentation is an open source object segmentation toolbox based on PyTorch. It is a part of the [OpenMMLab](#) project.

18.1 MMSegmentation installation tutorial

Please refer to [get_started.md](#) for installation.

18.2 List of MMSegmentation models supported by MMDeploy

18.3 Reminder

- Only `whole` inference mode is supported for all mmseg models.
- PSPNet, Fast-SCNN only support static shape, because `nn.AdaptiveAvgPool2d` is not supported in most of backends dynamically.
- For models only supporting static shape, you should use the deployment config file of static shape such as `configs/mmdet/segmentation_tensorrt_static-1024x2048.py`.
- For users prefer deployed models generate probability feature map, put `codebase_config = dict(with_argmax=False)` in deploy configs.

MMEDITING SUPPORT

MMEdit**ing** is an open-source image and video editing toolbox based on PyTorch. It is a part of the [OpenMMLab](#) project.

19.1 MMEdit**ing** installation tutorial

Please refer to [official installation guide](#) to install the codebase.

19.2 MMEdit**ing** models support

1. We skipped quantitative evaluation for image inpainting due to the high computational cost required for testing.

MMOCR SUPPORT

MMOCR is an open-source toolbox based on PyTorch and mmdetection for text detection, text recognition, and the corresponding downstream tasks including key information extraction. It is a part of the [OpenMMLab](#) project.

20.1 MMOCR installation tutorial

Please refer to [install.md](#) for installation.

20.2 List of MMOCR models supported by MMDeploy

20.3 Reminder

Note that ncnn, pplnn, and OpenVINO only support the configs of DBNet18 for DBNet.

For CRNN models with TensorRT-int8 backend, we recommend TensorRT 7.2.3.4 and CUDA 10.2.

For the PANet with the [checkpoint](#) pretrained on ICDAR dataset, if you want to convert the model to TensorRT with 16 bits float point, please try the following script.

```
# Copyright (c) OpenMMLab. All rights reserved.
from typing import Sequence

import torch
import torch.nn.functional as F

from mmddeploy.core import FUNCTION_REWRITER
from mmddeploy.utils.constants import Backend

FACTOR = 32
ENABLE = False
CHANNEL_THRESH = 400

@FUNCTION_REWRITER.register_rewriter(
    func_name='mmocr.models.textdet.necks.FPEM_FFM.forward',
    backend=Backend.TENSORRT.value)
def fpem_ffm__forward__trt(ctx, self, x: Sequence[torch.Tensor], *args,
                           **kwargs) -> Sequence[torch.Tensor]:
```

(continues on next page)

(continued from previous page)

```

"""Rewrite `forward` of FPEM_FFM for tensorrt backend.

Rewrite this function avoid overflow for tensorrt-fp16 with the checkpoint
`https://download.openmmlab.com/mmdet/textdet/panet/panet_r18_fpem_ffm
_sbn_600e_icdar2015_20210219-42dbe46a.pth`

Args:
    ctx (ContextCaller): The context with additional information.
    self: The instance of the class FPEM_FFM.
    x (List[Tensor]): A list of feature maps of shape (N, C, H, W).

Returns:
    outs (List[Tensor]): A list of feature maps of shape (N, C, H, W).
    """
c2, c3, c4, c5 = x
# reduce channel
c2 = self.reduce_conv_c2(c2)
c3 = self.reduce_conv_c3(c3)
c4 = self.reduce_conv_c4(c4)

if ENABLE:
    bn_w = self.reduce_conv_c5[1].weight / torch.sqrt(
        self.reduce_conv_c5[1].running_var + self.reduce_conv_c5[1].eps)
    bn_b = self.reduce_conv_c5[
        1].bias - self.reduce_conv_c5[1].running_mean * bn_w
    bn_w = bn_w.reshape(1, -1, 1, 1).repeat(1, 1, c5.size(2), c5.size(3))
    bn_b = bn_b.reshape(1, -1, 1, 1).repeat(1, 1, c5.size(2), c5.size(3))
    conv_b = self.reduce_conv_c5[0].bias.reshape(1, -1, 1, 1).repeat(
        1, 1, c5.size(2), c5.size(3))
    c5 = FACTOR * (self.reduce_conv_c5[:-1](c5)) - (FACTOR - 1) * (
        bn_w * conv_b + bn_b)
    c5 = self.reduce_conv_c5[-1](c5)
else:
    c5 = self.reduce_conv_c5(c5)

# FPEM
for i, fpem in enumerate(self.fpems):
    c2, c3, c4, c5 = fpem(c2, c3, c4, c5)
    if i == 0:
        c2_ffm = c2
        c3_ffm = c3
        c4_ffm = c4
        c5_ffm = c5
    else:
        c2_ffm += c2
        c3_ffm += c3
        c4_ffm += c4
        c5_ffm += c5

# FFM
c5 = F.interpolate(
    c5_ffm,

```

(continues on next page)

(continued from previous page)

```

        c2_ffm.size()[-2:],
        mode='bilinear',
        align_corners=self.align_corners)
    c4 = F.interpolate(
        c4_ffm,
        c2_ffm.size()[-2:],
        mode='bilinear',
        align_corners=self.align_corners)
    c3 = F.interpolate(
        c3_ffm,
        c2_ffm.size()[-2:],
        mode='bilinear',
        align_corners=self.align_corners)
    outs = [c2_ffm, c3, c4, c5]
    return tuple(outs)

@FUNCTION_REWRITER.register_rewriter(
    func_name='mmdet.models.backbones.resnet.BasicBlock.forward',
    backend=Backend.TENSORRT.value)
def basic_block_forward_trt(ctx, self, x: torch.Tensor) -> torch.Tensor:
    """Rewrite `forward` of BasicBlock for tensorrt backend.

    Rewrite this function avoid overflow for tensorrt-fp16 with the checkpoint
    https://download.openmmlab.com/mmdet/textdet/panet/panet\_r18\_fpem\_ffm\_sbn\_600e\_icdar2015\_20210219-42dbe46a.pth

    Args:
        ctx (ContextCaller): The context with additional information.
        self: The instance of the class FPEM_FFM.
        x (Tensor): The input tensor of shape (N, C, H, W).

    Returns:
        outs (Tensor): The output tensor of shape (N, C, H, W).
    """
    if self.conv1.in_channels < CHANNEL_THRESH:
        return ctx.origin_func(self, x)

    identity = x

    out = self.conv1(x)
    out = self.norm1(out)
    out = self.relu(out)

    out = self.conv2(out)

    if torch.abs(self.norm2(out)).max() < 65504:
        out = self.norm2(out)
        out += identity
        out = self.relu(out)
        return out
    else:

```

(continues on next page)

(continued from previous page)

```
global ENABLE
ENABLE = True
# the output of the last bn layer exceeds the range of fp16
w1 = self.norm2.weight / torch.sqrt(self.norm2.running_var +
                                     self.norm2.eps)
bias = self.norm2.bias - self.norm2.running_mean * w1
w1 = w1.reshape(1, -1, 1, 1).repeat(1, 1, out.size(2), out.size(3))
bias = bias.reshape(1, -1, 1, 1).repeat(1, 1, out.size(2),
                                         out.size(3)) + identity
out = self.relu(w1 * (out / FACTOR) + bias / FACTOR)

return out
```

MMPOSE SUPPORT

MMPose is an open-source toolbox for pose estimation based on PyTorch. It is a part of the [OpenMMLab](#) project.

21.1 MMPose installation tutorial

Please refer to [official installation guide](#) to install the codebase.

21.2 MMPose models support

21.2.1 Example

```
python tools/deploy.py \
configs/mmpose/posedetection_tensorrt_static-256x192.py \
$MMPOSE_DIR/configs/body/2d_kpt_sview_rgb_img/topdown_heatmap/coco/hrnet_w48_coco_
↪256x192.py \
$MMPOSE_DIR/checkpoints/hrnet_w48_coco_256x192-b9e0b3ab_20200708.pth \
$MMDEPLOY_DIR/demo/resources/human-pose.jpg \
--work-dir work-dirs/mmpose/topdown/hrnet/trt \
--device cuda
```

Note

- Usually, mmpose models need some extra information for the input image, but we can't get it directly. So, when exporting the model, you can use `$MMDEPLOY_DIR/demo/resources/human-pose.jpg` as input.

MMDETECTION3D SUPPORT

MMDetection3d is a next-generation platform for general 3D object detection. It is a part of the [OpenMMLab](#) project.

22.1 MMDetection3d installation tutorial

Please refer to [getting_started.md](#) for installation.

22.2 Example

```
export MODEL_PATH=https://download.openmmlab.com/mmdetection3d/v1.0.0_models/
↪pointpillars/hv_pointpillars_secfpn_6x8_160e_kitti-3d-car/hv_pointpillars_secfpn_6x8_
↪160e_kitti-3d-car_20220331_134606-d42d15ed.pth

python tools/deploy.py \
    configs/mmdet3d/voxel-detection/voxel-detection_tensorrt_dynamic.py \
    ${MMDET3D_DIR}/configs/pointpillars/hv_pointpillars_secfpn_6x8_160e_kitti-3d-
↪3class.py \
    ${MODEL_PATH} \
    ${MMDET3D_DIR}/demo/data/kitti/kitti_000008.bin \
    --work-dir \
    work_dir \
    --show \
    --device \
    cuda:0
```

22.3 List of MMDetection3d models supported by MMDeploy

1. mmdet3d models on **cu102+TRT8.4** can be visualized normally. For cuda-11 or TRT8.2 users, these issues should be checked
 - TRT8.2 assertion `is_tensor`
 - TRT8.4 output NaN
2. Voxel detection onnx model excludes model.voxelize layer and model post process, and you can use python api to call these func.

Example:

```
from mmdeploy.codebase.mmdet3d.deploy import VoxelDetectionModel
VoxelDetectionModel.voxelize(...)
VoxelDetectionModel.post_process(...)
```

MMROTATE SUPPORT

MMRotate is an open-source toolbox for rotated object detection based on PyTorch. It is a part of the [OpenMMLab](#) project.

23.1 MMRotate installation tutorial

Please refer to [official installation guide](#) to install the codebase.

23.2 MMRotate models support

23.2.1 Example

```
# convert ort
python tools/deploy.py \
configs/mmrrotate/rotated-detection_onnxruntime_dynamic.py \
$MMROTATE_DIR/configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le135.py \
$MMROTATE_DIR/checkpoints/rotated_retinanet_obb_r50_fpn_1x_dota_le135-e4131166.pth \
$MMROTATE_DIR/demo/demo.jpg \
--work-dir work-dirs/mmrrotate/rotated_retinanet/ort \
--device cpu

# compute metric
python tools/test.py \
    configs/mmrrotate/rotated-detection_onnxruntime_dynamic.py \
    $MMROTATE_DIR/configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le135.
↪py \
    --model work-dirs/mmrrotate/rotated_retinanet/ort/end2end.onnx \
    --metrics mAP

# generate submit file
python tools/test.py \
    configs/mmrrotate/rotated-detection_onnxruntime_dynamic.py \
    $MMROTATE_DIR/configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le135.
↪py \
    --model work-dirs/mmrrotate/rotated_retinanet/ort/end2end.onnx \
    --format-only \
    --metric-options submission_dir=work-dirs/mmrrotate/rotated_retinanet/ort/Task1_
↪results
```

Note

- Usually, mmrotate models need some extra information for the input image, but we can't get it directly. So, when exporting the model, you can use `$MMROTATE_DIR/demo/demo.jpg` as input.

MMACTION2 SUPPORT

[MMAction2](#) is an open-source toolbox for video understanding based on PyTorch. It is a part of the [OpenMMLab](#) project.

24.1 Install mmaction2

Please follow the [installation guide](#) to install mmaction2.

24.2 Supported models

SUPPORTED NCNN FEATURE

The current use of the ncnn feature is as follows:

The following features cannot be automatically enabled by mmdeploy and you need to manually modify the ncnn build options or adjust the running parameters in the SDK

- bfloat16 inference
- nc4hw4 layout
- Profiling per layer
- Turn off NCNN_STRING to reduce .so file size
- Set thread number and CPU affinity

ONNX RUNTIME SUPPORT

26.1 Introduction of ONNX Runtime

ONNX Runtime is a cross-platform inference and training accelerator compatible with many popular ML/DNN frameworks. Check its [github](#) for more information.

26.2 Installation

*Please note that only **onnxruntime>=1.8.1** of CPU version on Linux platform is supported by now.*

- Install ONNX Runtime python package

```
pip install onnxruntime==1.8.1
```

26.3 Build custom ops

26.3.1 Prerequisite

- Download onnxruntime-linux from ONNX Runtime [releases](#), extract it, expose ONNXRUNTIME_DIR and finally add the lib path to LD_LIBRARY_PATH as below:

```
wget https://github.com/microsoft/onnxruntime/releases/download/v1.8.1/onnxruntime-linux-  
x64-1.8.1.tgz  
  
tar -zxvf onnxruntime-linux-x64-1.8.1.tgz  
cd onnxruntime-linux-x64-1.8.1  
export ONNXRUNTIME_DIR=$(pwd)  
export LD_LIBRARY_PATH=$ONNXRUNTIME_DIR/lib:$LD_LIBRARY_PATH
```

26.3.2 Build on Linux

```
cd ${MMDEPLOY_DIR} # To MMDeploy root directory
mkdir -p build && cd build
cmake -DMMDEPLOY_TARGET_BACKENDS=ort -DONNXRUNTIME_DIR=${ONNXRUNTIME_DIR} ..
make -j$(nproc) && make install
```

26.4 How to convert a model

- You could follow the instructions of tutorial *How to convert model*

26.5 How to add a new custom op

26.6 Reminder

- The custom operator is not included in [supported operator list](#) in ONNX Runtime.
- The custom operator should be able to be exported to ONNX.

26.6.1 Main procedures

Take custom operator `roi_align` for example.

1. Create a `roi_align` directory in ONNX Runtime source directory `${MMDEPLOY_DIR}/csrc/backend_ops/onnxruntime/`
2. Add header and source file into `roi_align` directory `${MMDEPLOY_DIR}/csrc/backend_ops/onnxruntime/roi_align/`
3. Add unit test into `tests/test_ops/test_ops.py` Check here for examples.

Finally, welcome to send us PR of adding custom operators for ONNX Runtime in MMDeploy. :nerd_face:

26.7 References

- [How to export Pytorch model with custom op to ONNX and run it in ONNX Runtime](#)
- [How to add a custom operator/kernel in ONNX Runtime](#)

OPENVINO SUPPORT

This tutorial is based on Linux systems like Ubuntu-18.04.

27.1 Installation

It is recommended to create a virtual environment for the project.

1. Install [OpenVINO](#). It is recommended to use the installer or install using pip. Installation example using pip:

```
pip install opencvino-dev>=2022.3.0
```

2. *Optional If you want to use OpenVINO in SDK, you need install OpenVINO with [install_guides](#).
3. Install MMDeploy following the [instructions](#).

To work with models from [MMDetection](#), you may need to install it additionally.

27.2 Usage

Example:

```
python tools/deploy.py \  
  configs/mmdet/detection/detection_opencvino_static-300x300.py \  
  /mmdetection_dir/mmdetection/configs/ssd/ssd300_coco.py \  
  /tmp/snapshots/ssd300_coco_20210803_015428-d231a06e.pth \  
  tests/data/tiger.jpeg \  
  --work-dir ../deploy_result \  
  --device cpu \  
  --log-level INFO
```

27.3 List of supported models exportable to OpenVINO from MMDetection

The table below lists the models that are guaranteed to be exportable to OpenVINO from MMDetection.

Notes:

- Custom operations from OpenVINO use the domain `org.openvinotoolkit`.
- For faster work in OpenVINO in the Faster-RCNN, Mask-RCNN, Cascade-RCNN, Cascade-Mask-RCNN models the RoiAlign operation is replaced with the `ExperimentalDetectronROIFeatureExtractor` operation in the ONNX graph.
- Models “VFNet” and “Faster R-CNN + DCN” use the custom “DeformableConv2D” operation.

27.4 Deployment config

With the deployment config, you can specify additional options for the Model Optimizer. To do this, add the necessary parameters to the `backend_config.mo_options` in the fields `args` (for parameters with values) and `flags` (for flags).

Example:

```
backend_config = dict(  
    mo_options=dict(  
        args=dict(  
            '--mean_values': [0, 0, 0],  
            '--scale_values': [255, 255, 255],  
            '--data_type': 'FP32',  
        }),  
        flags=['--disable_fusing'],  
    )  
)
```

Information about the possible parameters for the Model Optimizer can be found in the [documentation](#).

27.5 Troubleshooting

- `ImportError: libpython3.7m.so.1.0: cannot open shared object file: No such file or directory`

To resolve missing external dependency on Ubuntu*, execute the following command:

```
sudo apt-get install libpython3.7
```

PPLNN SUPPORT

MMDeploy supports ppl.nn v0.9.1 and later. This tutorial is based on Linux systems like Ubuntu-18.04.

28.1 Installation

1. Please install `pypp1` following [install-guide](#).
2. Install MMDeploy following the [instructions](#).

28.2 Usage

Example:

```
python tools/deploy.py \
  configs/mmdet/detection/detection_pplnn_dynamic-800x1344.py \
  /mmdetection_dir/mmdetection/configs/retinanet/retinanet_r50_fpn_1x_coco.py \
  /tmp/snapshots/retinanet_r50_fpn_1x_coco_20200130-c2398f9e.pth \
  tests/data/tiger.jpeg \
  --work-dir ../deploy_result \
  --device cuda \
  --log-level INFO
```


SNPE FEATURE SUPPORT

Currently mmdeploy integrates the onnx2dlc model conversion and SDK inference, but the following features are not yet supported:

- GPU_FP16 mode
- DSP/AIP quantization
- Operator internal profiling
- UDO operator

TENSORRT SUPPORT

30.1 Installation

30.1.1 Install TensorRT

Please install TensorRT 8 follow [install-guide](#).

Note:

- pip Wheel File Installation is not supported yet in this repo.
- We strongly suggest you install TensorRT through tar file
- After installation, you'd better add TensorRT environment variables to bashrc by:

```
cd ${TENSORRT_DIR} # To TensorRT root directory
echo '# set env for TensorRT' >> ~/.bashrc
echo "export TENSORRT_DIR=${TENSORRT_DIR}" >> ~/.bashrc
echo 'export LD_LIBRARY_PATH=$TENSORRT_DIR/lib:$TENSORRT_DIR' >> ~/.bashrc
source ~/.bashrc
```

30.1.2 Build custom ops

Some custom ops are created to support models in OpenMMLab, and the custom ops can be built as follow:

```
cd ${MMDEPLOY_DIR} # To MMDeploy root directory
mkdir -p build && cd build
cmake -DMMDEPLOY_TARGET_BACKENDS=trt ..
make -j$(nproc)
```

If you haven't installed TensorRT in the default path, Please add `-DTENSORRT_DIR` flag in CMake.

```
cmake -DMMDEPLOY_TARGET_BACKENDS=trt -DTENSORRT_DIR=${TENSORRT_DIR} ..
make -j$(nproc) && make install
```

30.2 Convert model

Please follow the tutorial in *How to convert model*. **Note** that the device must be cuda device.

30.2.1 Int8 Support

Since TensorRT supports INT8 mode, a custom dataset config can be given to calibrate the model. Following is an example for MMDetection:

```
# calibration_dataset.py

# dataset settings, same format as the codebase in OpenMMLab
dataset_type = 'CalibrationDataset'
data_root = 'calibration/dataset/root'
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
    val=dict(
        type=dataset_type,
        ann_file=data_root + 'val_annotations.json',
        pipeline=test_pipeline),
    test=dict(
        type=dataset_type,
        ann_file=data_root + 'test_annotations.json',
        pipeline=test_pipeline))
evaluation = dict(interval=1, metric='bbox')
```

Convert your model with this calibration dataset:

```
python tools/deploy.py \
    ...
    --calib-dataset-cfg calibration_dataset.py
```

If the calibration dataset is not given, the data will be calibrated with the dataset in model config.

30.3 FAQs

- Error Cannot found TensorRT headers or Cannot found TensorRT libs

Try cmake with flag `-DTENSORRT_DIR`:

```
cmake -DBUILD_TENSORRT_OPS=ON -DTENSORRT_DIR=${TENSORRT_DIR} ..
make -j$(nproc)
```

Please make sure there are libs and headers in `${TENSORRT_DIR}`.

- Error error: parameter check failed at: engine.cpp::setBindingDimensions::1046, condition: profileMinDims.d[i] <= dimensions.d[i]

There is an input shape limit in deployment config:

```
backend_config = dict(
    # other configs
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 320, 320],
                    opt_shape=[1, 3, 800, 1344],
                    max_shape=[1, 3, 1344, 1344]))))
    ])
    # other configs
```

The shape of the tensor input must be limited between `input_shapes["input"]["min_shape"]` and `input_shapes["input"]["max_shape"]`.

- Error error: [TensorRT] INTERNAL ERROR: Assertion failed: cublasStatus == CUBLAS_STATUS_SUCCESS

TRT 7.2.1 switches to use cuBLASLt (previously it was cuBLAS). cuBLASLt is the default choice for SM version ≥ 7.0 . However, you may need CUDA-10.2 Patch 1 (Released Aug 26, 2020) to resolve some cuBLASLt issues. Another option is to use the new TacticSource API and disable cuBLASLt tactics if you don't want to upgrade.

Read [this](#) for detail.

- Install mmdeploy on Jetson

We provide a tutorial to get start on Jetsons [here](#).

TORCHSCRIPT SUPPORT

31.1 Introduction of TorchScript

TorchScript a way to create serializable and optimizable models from PyTorch code. Any TorchScript program can be saved from a Python process and loaded in a process where there is no Python dependency. Check the [Introduction to TorchScript](#) for more details.

31.2 Build custom ops

31.2.1 Prerequisite

- Download libtorch from the official website [here](#).

Please note that only **Pre-cxx11 ABI** and **version 1.8.1+** on Linux platform are supported by now.

For previous versions of libtorch, users can find through the [issue comment](#). Libtorch1.8.1+cu111 as an example, extract it, expose Torch_DIR and add the lib path to LD_LIBRARY_PATH as below:

```
wget https://download.pytorch.org/libtorch/cu111/libtorch-shared-with-deps-1.8.1%2Bcu111.
↪ zip
unzip libtorch-shared-with-deps-1.8.1+cu111.zip
cd libtorch
export Torch_DIR=$(pwd)
export LD_LIBRARY_PATH=$Torch_DIR/lib:$LD_LIBRARY_PATH
```

Note:

- If you want to save libtorch env variables to bashrc, you could run

```
echo '# set env for libtorch' >> ~/.bashrc
echo "export Torch_DIR=${Torch_DIR}" >> ~/.bashrc
echo 'export LD_LIBRARY_PATH=$Torch_DIR/lib:$LD_LIBRARY_PATH' >> ~/.bashrc
source ~/.bashrc
```

31.2.2 Build on Linux

```
cd ${MMDEPLOY_DIR} # To MMDeploy root directory
mkdir -p build && cd build
cmake -DMMDEPLOY_TARGET_BACKENDS=torchscript -DTorch_DIR=${Torch_DIR} ..
make -j$(nproc) && make install
```

31.3 How to convert a model

- You could follow the instructions of tutorial *How to convert model*

31.4 SDK backend

TorchScript SDK backend may be built by passing `-DMMDEPLOY_TORCHSCRIPT_SDK_BACKEND=ON` to `cmake`.

Notice that `libtorch` is sensitive to C++ ABI versions. On platforms defaulted to C++11 ABI (e.g. Ubuntu 16+) one may pass `-DCMAKE_CXX_FLAGS="-D_GLIBCXX_USE_CXX11_ABI=0"` to `cmake` to use pre-C++11 ABI for building. In this case all dependencies with ABI sensitive interfaces (e.g. OpenCV) must be built with pre-C++11 ABI.

31.5 FAQs

- Error: `projects/thirdparty/libtorch/share/cmake/Caffe2/Caffe2Config.cmake:96 (message):Your installed Caffe2 version uses cuDNN but I cannot find the cuDNN libraries. Please set the proper cuDNN prefixes and / or install cuDNN.`

May export `CUDNN_ROOT=/root/path/to/cudnn` to resolve the build error.

SUPPORTED RKNN FEATURE

Currently, MMDeploy only tests rk3588 and rv1126 with linux platform.

The following features cannot be automatically enabled by mmdeploy and you need to manually modify the configuration in MMDeploy like [here](#).

- target_platform other than default
- quantization settings
- optimization level other than 1

CORE ML FEATURE SUPPORT

MMDeploy support convert Pytorch model to Core ML and inference.

33.1 Installation

To convert the model in mmdet, you need to compile libtorch to support custom operators such as nms.

```
cd ${PYTORCH_DIR}
mkdir build && cd build
cmake .. \
  -DCMAKE_BUILD_TYPE=Release \
  -DPYTHON_EXECUTABLE=`which python` \
  -DCMAKE_INSTALL_PREFIX=install \
  -DDISABLE_SVE=ON # low version like 1.8.0 of pytorch need this option
make install
```

33.2 Usage

```
python tools/deploy.py \
  configs/mmdet/detection/detection_coreml_static-800x1344.py \
  /mmdetection_dir/configs/retinanet/retinanet_r18_fpn_1x_coco.py \
  /checkpoint/retinanet_r18_fpn_1x_coco_20220407_171055-614fd399.pth \
  /mmdetection_dir/demo/demo.jpg \
  --work-dir work_dir/retinanet \
  --device cpu \
  --dump-info
```


ONNX RUNTIME OPS

- *ONNX Runtime Ops*
 - *grid_sampler*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *MMCVModulatedDeformConv2d*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
- *NMSRotated*
 - *Description*
 - *Parameters*
 - *Inputs*
 - *Outputs*
 - *Type Constraints*
 - *RoIAlignRotated*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*

34.1 grid_sampler

34.1.1 Description

Perform sample from `input` with pixel locations from `grid`.

34.1.2 Parameters

34.1.3 Inputs

34.1.4 Outputs

34.1.5 Type Constraints

- T:tensor(float32, Linear)

34.2 MMCVModulatedDeformConv2d

34.2.1 Description

Perform Modulated Deformable Convolution on input feature, read [Deformable ConvNets v2: More Deformable, Better Results](#) for detail.

34.2.2 Parameters

34.2.3 Inputs

34.2.4 Outputs

34.2.5 Type Constraints

- T:tensor(float32, Linear)

34.3 NMSRotated

34.3.1 Description

Non Max Suppression for rotated bboxes.

34.3.2 Parameters

34.3.3 Inputs

34.3.4 Outputs

34.3.5 Type Constraints

- T:tensor(float32, Linear)

34.4 RoIAlignRotated

34.4.1 Description

Perform RoIAlignRotated on output feature, used in bbox_head of most two-stage rotated object detectors.

34.4.2 Parameters

34.4.3 Inputs

34.4.4 Outputs

34.4.5 Type Constraints

- T:tensor(float32)

TENSORRT OPS

- *TensorRT Ops*
 - *TRTBatchedNMS*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *grid_sampler*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *MMCVMInstanceNormalization*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *MMCVModulatedDeformConv2d*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *MMCVMultiLevelRoiAlign*
 - * *Description*

- * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
- *MMCVRoIAlign*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
- *ScatterND*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
- *TRTBatchedRotatedNMS*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
- *GridPriorsTRT*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
- *ScaledDotProductAttentionTRT*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
- *GatherTopk*
 - * *Description*

- * *Parameters*
- * *Inputs*
- * *Outputs*
- * *Type Constraints*

35.1 TRTBatchedNMS

35.1.1 Description

Batched NMS with a fixed number of output bounding boxes.

35.1.2 Parameters

35.1.3 Inputs

35.1.4 Outputs

35.1.5 Type Constraints

- T:tensor(float32, Linear)

35.2 grid_sampler

35.2.1 Description

Perform sample from input with pixel locations from grid.

35.2.2 Parameters

35.2.3 Inputs

35.2.4 Outputs

35.2.5 Type Constraints

- T:tensor(float32, Linear)

35.3 MMCVInstanceNormalization

35.3.1 Description

Carry out instance normalization as described in the paper <https://arxiv.org/abs/1607.08022>.

$y = \text{scale} * (x - \text{mean}) / \sqrt{\text{variance} + \text{epsilon}} + B$, where mean and variance are computed per instance per channel.

35.3.2 Parameters

35.3.3 Inputs

35.3.4 Outputs

35.3.5 Type Constraints

- T:tensor(float32, Linear)

35.4 MMCVModulatedDeformConv2d

35.4.1 Description

Perform Modulated Deformable Convolution on input feature. Read [Deformable ConvNets v2: More Deformable, Better Results](#) for detail.

35.4.2 Parameters

35.4.3 Inputs

35.4.4 Outputs

35.4.5 Type Constraints

- T:tensor(float32, Linear)

35.5 MMCVMultiLevelRoiAlign

35.5.1 Description

Perform RoIAlign on features from multiple levels. Used in bbox_head of most two-stage detectors.

35.5.2 Parameters

35.5.3 Inputs

35.5.4 Outputs

35.5.5 Type Constraints

- T:tensor(float32, Linear)

35.6 MMCVRoIAlign

35.6.1 Description

Perform RoIAlign on output feature, used in bbox_head of most two-stage detectors.

35.6.2 Parameters

35.6.3 Inputs

35.6.4 Outputs

35.6.5 Type Constraints

- T:tensor(float32, Linear)

35.7 ScatterND

35.7.1 Description

ScatterND takes three inputs `data` tensor of rank $r \geq 1$, `indices` tensor of rank $q \geq 1$, and `updates` tensor of rank $q + r - \text{indices.shape}[-1] - 1$. The output of the operation is produced by creating a copy of the input `data`, and then updating its value to values specified by `updates` at specific index positions specified by `indices`. Its output shape is the same as the shape of `data`. Note that `indices` should not have duplicate entries. That is, two or more updates for the same index-location is not supported.

The output is calculated via the following equation:

```
output = np.copy(data)
update_indices = indices.shape[:-1]
for idx in np.ndindex(update_indices):
    output[indices[idx]] = updates[idx]
```

35.7.2 Parameters

None

35.7.3 Inputs

35.7.4 Outputs

35.7.5 Type Constraints

- T:tensor(float32, Linear), tensor(int32, Linear)

35.8 TRTBatchedRotatedNMS

35.8.1 Description

Batched rotated NMS with a fixed number of output bounding boxes.

35.8.2 Parameters

35.8.3 Inputs

35.8.4 Outputs

35.8.5 Type Constraints

- T:tensor(float32, Linear)

35.9 GridPriorsTRT

35.9.1 Description

Generate the anchors for object detection task.

35.9.2 Parameters

35.9.3 Inputs

35.9.4 Outputs

35.9.5 Type Constraints

- T:tensor(float32, Linear)
- TAny: Any

35.10 ScaledDotProductAttentionTRT

35.10.1 Description

Dot product attention used to support multihead attention, read [Attention Is All You Need](#) for more detail.

35.10.2 Parameters

None

35.10.3 Inputs

35.10.4 Outputs

35.10.5 Type Constraints

- T:tensor(float32, Linear)

35.11 GatherTopk

35.11.1 Description

TensorRT 8.2~8.4 would give unexpected result for multi-index gather.

```
data[batch_index, bbox_index, ...]
```

Read [this](#) for more details.

35.11.2 Parameters

None

35.11.3 Inputs

35.11.4 Outputs

35.11.5 Type Constraints

- T:tensor(float32, Linear), tensor(int32, Linear)

NCNN OPS

- *ncnn Ops*
 - *Expand*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *Gather*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *Shape*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*
 - *TopK*
 - * *Description*
 - * *Parameters*
 - * *Inputs*
 - * *Outputs*
 - * *Type Constraints*

36.1 Expand

36.1.1 Description

Broadcast the input blob following the given shape and the broadcast rule of ncnn.

36.1.2 Parameters

Expand has no parameters.

36.1.3 Inputs

36.1.4 Outputs

36.1.5 Type Constraints

- ncnn.Mat: Mat(float32)

36.2 Gather

36.2.1 Description

Given the data and indice blob, gather entries of the axis dimension of data indexed by indices.

36.2.2 Parameters

36.2.3 Inputs

36.2.4 Outputs

36.2.5 Type Constraints

- ncnn.Mat: Mat(float32)

36.3 Shape

36.3.1 Description

Get the shape of the ncnn blobs.

36.3.2 Parameters

Shape has no parameters.

36.3.3 Inputs

36.3.4 Outputs

36.3.5 Type Constraints

- ncnn.Mat: Mat(float32)

36.4 TopK

36.4.1 Description

Get the indices and value(optional) of largest or smallest k data among the axis. This op will map to onnx op TopK, ArgMax, and ArgMin.

36.4.2 Parameters

36.4.3 Inputs

36.4.4 Outputs

36.4.5 Type Constraints

- ncnn.Mat: Mat(float32)

MMDEPLOY ARCHITECTURE

This article mainly introduces the functions of each directory of mmdeploy and how it works from model conversion to real inference.

37.1 Take a general look at the directory structure

The entire mmdeploy can be seen as two independent parts: model conversion and SDK.

We introduce the entire repo directory structure and functions, without having to study the source code, just have an impression.

Peripheral directory features:

```
$ cd /path/to/mmdploy
$ tree -L 1
.
├── CMakeLists.txt      # Compile custom operator and cmake configuration of SDK
├── configs             # Algorithm library configuration for model conversion
├── csrc               # SDK and custom operator
├── demo               # FFI interface examples in various languages, such as
├── csharp, java, python, etc.
├── docker             # docker build
├── mmdploy            # python package for model conversion
├── requirements       # python requirements
├── service            # Some small boards not support python, we use C/S mode
├── for model conversion, here is server code
├── tests              # unittest
├── third_party        # 3rd party dependencies required by SDK and FFI
├── tools              # Tools are also the entrance to all functions, such as
├── onnx2xx.py, profile.py, test.py, etc.
```

It should be clear

- Model conversion mainly depends on tools, mmdploy and small part of csrc directory;
- SDK is consist of three directories: csrc, third_party and demo.

37.2 Model Conversion

Here we take ViT of mmcls as model example, and take ncnn as inference backend example. Other models and inferences are similar.

Let's take a look at the mmdeploy/mmdeploy directory structure and get an impression:

```
.
├── apis                                # The api used by tools is implemented here, such_
├── as onnx2ncnn.py
│   ├── calibration.py                # trt dedicated collection of quantitative data
│   ├── core                          # Software infrastructure
│   ├── extract_model.py             # Use it to export part of onnx
│   ├── inference.py                 # Abstract function, which will actually call torch/
├── ncnn specific inference
│   ├── ncnn                         # ncnn Wrapper
│   └── visualize.py                 # Still an abstract function, which will actually call_
├── torch/ncnn specific inference and visualize
..
├── backend                           # Backend wrapper
│   ├── base                         # Because there are multiple backends, there_
├── must be an OO design for the base class
│   ├── ncnn                         # This calls the ncnn python interface for model_
├── conversion
│   ├── init_plugins.py              # Find the path of ncnn custom operators and ncnn_
├── tools
│   ├── onnx2ncnn.py                 # Wrap `mmdeploy_onnx2ncnn` into a python interface
│   ├── quant.py                     # Wrap `ncnn2int8` as a python interface
│   └── wrapper.py                   # Wrap pyncnn forward API
..
├── codebase                           # Algorithm rewriter
│   ├── base                         # There are multiple algorithms here that we need_
├── a bit of OO design
│   ├── mmcls                       # mmcls related model rewrite
│   │   └── deploy                   # mmcls implementation of base abstract task/
├── model/codebase
│   │   └── models                     # Real model rewrite
│   │       ├── backbones             # Rewrites of backbone network parts, such as_
├── multiheadattention
│   │   ├── heads                     # Such as MultiLabelClsHead
│   │   └── necks                     # Such as GlobalAveragePooling
..
├── core                               # Software infrastructure of rewrite mechanism
├── mmcv                               # Rewrite mmcv
├── pytorch                           # Rewrite pytorch operator for ncnn, such as Gemm
..
```

Each line above needs to be read, don't skip it.

When typing tools/deploy.py to convert ViT, these are 3 things:

1. Rewrite of mmcls ViT forward
2. ncnn does not support gather opr, customize and load it with libncnn.so
3. Run exported ncnn model with real inference, render output, and make sure the result is correct

37.2.1 1. Rewrite forward

Because when exporting ViT to onnx, it generates some operators that ncnn doesn't support perfectly, mmdeploy's solution is to hijack the forward code and change it. The output onnx is suitable for ncnn.

For example, rewrite the process of `conv -> shape -> concat_const -> reshape` to `conv -> reshape` to trim off the redundant `shape` and `concat` operator.

All mmcls algorithm rewriters are in the `mmdeploy/codebase/mmcls/models` directory.

37.2.2 2. Custom Operator

Operators customized for ncnn are in the `csrc/mmdeploy/backend_ops/ncnn/` directory, and are loaded together with `libncnn.so` after compilation. The essence is in hotfix ncnn, which currently implements these operators:

- `topk`
- `tensorslice`
- `shpe`
- `gather`
- `expand`
- `constantofshape`

37.2.3 3. Model Conversion and testing

We first use the modified `mmdeploy_onnx2ncnn` to convert model, then inference with `pyncnn` and custom ops.

When encountering a framework such as snpe that does not support python well, we use C/S mode: wrap a server with protocols such as gRPC, and forward the real inference output.

For Rendering, mmdeploy directly uses the rendering API of upstream algorithm codebase.

37.3 SDK

After the model conversion completed, the SDK compiled with C++ can be used to execute on different platforms.

Let's take a look at the `csrc/mmdeploy` directory structure:

```
.
├── apis                # csharp, java, go, Rust and other FFI interfaces
├── backend_ops         # Custom operators for each inference framework
├── CMakeLists.txt
├── codebase            # The type of results preferred by each algorithm framework, such as
├──                    ↳ multi-use bbox for detection task
├── core                # Abstraction of graph, operator, device and so on
├── device              # Implementation of CPU/GPU device abstraction
├── execution           # Implementation of the execution abstraction
├── graph               # Implementation of graph abstraction
├── model               # Implement both zip-compressed and uncompressed work directory
├── net                 # Implementation of net, such as wrap ncnn forward C API
├── preprocess          # Implement preprocess
└── utils               # OCV tools
```

The essence of the SDK is to design a set of abstraction of the computational graph, and combine the **multiple models**'

- preprocess
- inference
- postprocess

Provide FFI in multiple languages at the same time.

HOW TO SUPPORT NEW MODELS

We provide several tools to support model conversion.

38.1 Function Rewriter

The PyTorch neural network is written in python that eases the development of the algorithm. But the use of Python control flow and third-party libraries make it difficult to export the network to an intermediate representation. We provide a ‘monkey patch’ tool to rewrite the unsupported function to another one that can be exported. Here is an example:

```
from mmdesploy.core import FUNCTION_REWRITER

@FUNCTION_REWRITER.register_rewriter(
    func_name='torch.Tensor.repeat', backend='tensorrt')
def repeat_static(ctx, input, *size):
    origin_func = ctx.origin_func
    if input.dim() == 1 and len(size) == 1:
        return origin_func(input.unsqueeze(0), *([1] + list(size))).squeeze(0)
    else:
        return origin_func(input, *size)
```

It is easy to use the function rewriter. Just add a decorator with arguments:

- `func_name` is the function to override. It can be either a PyTorch function or a custom function. Methods in modules can also be overridden by this tool.
- `backend` is the inference engine. The function will be overridden when the model is exported to this engine. If it is not given, this rewrite will be the default rewrite. The default rewrite will be used if the rewrite of the given backend does not exist.

The arguments are the same as the original function, except a context `ctx` as the first argument. The context provides some useful information such as the deployment config `ctx.cfg` and the original function (which has been overridden) `ctx.origin_func`.

38.2 Module Rewriter

If you want to replace a whole module with another one, we have another rewriter as follows:

```
@MODULE_REWRITER.register_rewrite_module(
    'mmedit.models.backbones.sr_backbones.SRCNN', backend='tensorrt')
class SRCNNWrapper(nn.Module):

    def __init__(self,
                  module,
                  cfg,
                  channels=(3, 64, 32, 3),
                  kernel_sizes=(9, 1, 5),
                  upscale_factor=4):
        super(SRCNNWrapper, self).__init__()

        self._module = module

        module.img_upsampler = nn.Upsample(
            scale_factor=module.upscale_factor,
            mode='bilinear',
            align_corners=False)

    def forward(self, *args, **kwargs):
        """Run forward."""
        return self._module(*args, **kwargs)

    def init_weights(self, *args, **kwargs):
        """Initialize weights."""
        return self._module.init_weights(*args, **kwargs)
```

Just like function rewriter, add a decorator with arguments:

- `module_type` the module class to rewrite.
- `backend` is the inference engine. The function will be overridden when the model is exported to this engine. If it is not given, this rewrite will be the default rewrite. The default rewrite will be used if the rewrite of the given backend does not exist.

All instances of the module in the network will be replaced with instances of this new class. The original module and the deployment config will be passed as the first two arguments.

38.3 Custom Symbolic

The mappings between PyTorch and ONNX are defined in PyTorch with symbolic functions. The custom symbolic function can help us to bypass some ONNX nodes which are unsupported by inference engine.

```
@SYMBOLIC_REWRITER.register_symbolic('squeeze', is_pytorch=True)
def squeeze_default(ctx, g, self, dim=None):
    if dim is None:
        dims = []
        for i, size in enumerate(self.type().sizes()):
            if size == 1:
```

(continues on next page)

(continued from previous page)

```
        dims.append(i)
    else:
        dims = [sym_help._get_const(dim, 'i', 'dim')]
    return g.op('Squeeze', self, axes_i=dims)
```

The decorator arguments:

- **func_name** The function name to add symbolic. Use full path if it is a custom `torch.autograd.Function`. Or just a name if it is a PyTorch built-in function.
- **backend** is the inference engine. The function will be overridden when the model is exported to this engine. If it is not given, this rewrite will be the default rewrite. The default rewrite will be used if the rewrite of the given backend does not exist.
- **is_pytorch** True if the function is a PyTorch built-in function.
- **arg_descriptors** the descriptors of the symbolic function arguments. Will be feed to `torch.onnx.symbolic_helper._parse_arg`.

Just like function rewriter, there is a context `ctx` as the first argument. The context provides some useful information such as the deployment config `ctx.cfg` and the original function (which has been overridden) `ctx.origin_func`. Note that the `ctx.origin_func` can be used only when `is_pytorch==False`.

HOW TO SUPPORT NEW BACKENDS

MMDeploy supports a number of backend engines. We welcome the contribution of new backends. In this tutorial, we will introduce the general procedures to support a new backend in MMDeploy.

39.1 Prerequisites

Before contributing the codes, there are some requirements for the new backend that need to be checked:

- The backend must support ONNX as IR.
- If the backend requires model files or weight files other than a “.onnx” file, a conversion tool that converts the “.onnx” file to model files and weight files is required. The tool can be a Python API, a script, or an executable program.
- It is highly recommended that the backend provides a Python interface to load the backend files and inference for validation.

39.2 Support backend conversion

The backends in MMDeploy must support the ONNX. The backend loads the “.onnx” file directly, or converts the “.onnx” to its own format using the conversion tool. In this section, we will introduce the steps to support backend conversion.

1. Add backend constant in `mmdeploy/utils/constants.py` that denotes the name of the backend.

Example:

```
# mmdeploy/utils/constants.py

class Backend(AdvancedEnum):
    # Take TensorRT as an example
    TENSORRT = 'tensorrt'
```

2. Add a corresponding package (a folder with `__init__.py`) in `mmdeploy/backend/`. For example, `mmdeploy/backend/tensorrt`. If the backend requires model files or weight files other than a “.onnx” file, create a `onnx2backend.py` file in the corresponding folder (e.g., create `mmdeploy/backend/tensorrt/onnx2tensorrt.py`). Then add a conversion function `onnx2backend` in the file. The function should convert a given “.onnx” file to the required backend files in a given work directory. There are no requirements on other parameters of the function and the implementation details. You can use any tools for conversion. Here are some examples:

Use Python script:

```
def onnx2openvino(input_info: Dict[str, Union[List[int], torch.Size]],
                  output_names: List[str], onnx_path: str, work_dir: str):

    input_names = ','.join(input_info.keys())
    input_shapes = ','.join(str(list(elem)) for elem in input_info.values())
    output = ','.join(output_names)

    mo_args = f'--input_model="{onnx_path}" '\
              f'--output_dir="{work_dir}" ' \
              f'--output="{output}" ' \
              f'--input="{input_names}" ' \
              f'--input_shape="{input_shapes}" ' \
              f'--disable_fusing '
    command = f'mo.py {mo_args}'
    mo_output = run(command, stdout=PIPE, stderr=PIPE, shell=True, check=True)
```

Use executable program:

```
def onnx2ncnn(onnx_path: str, work_dir: str):
    onnx2ncnn_path = get_onnx2ncnn_path()
    save_param, save_bin = get_output_model_file(onnx_path, work_dir)
    call([onnx2ncnn_path, onnx_path, save_param, save_bin])\
```

3. Create a backend manager class and implement the interface to support model conversion, version check and other features.

Example:

```
# register the backend manager
# the backend manager derive from BaseBackendManager
@BACKEND MANAGERS.register('tensorrt')
class TensorRTManager(BaseBackendManager):

    @classmethod
    def is_available(cls, with_custom_ops: bool = False) -> bool:
        ....

    @classmethod
    def get_version(cls) -> str:
        ....

    @classmethod
    def to_backend(cls,
                   ir_files: Sequence[str],
                   work_dir: str,
                   deploy_cfg: Any,
                   log_level: int = logging.INFO,
                   device: str = 'cpu',
                   **kwargs) -> Sequence[str]:
        ...
```

4. Create a config file in configs/_base_/backends (e.g., configs/_base_/backends/tensorrt.py). If the backend just takes the '.onnx' file as input, the new config can be simple. The config of the backend only

consists of one field denoting the name of the backend (which should be same as the name in `mmdeploy/utils/constants.py`).

Example:

```
backend_config = dict(type='onnxruntime')
```

If the backend requires other files, then the arguments for the conversion from “.onnx” file to backend files should be included in the config file.

Example:

```
backend_config = dict(
    type='tensorrt',
    common_config=dict(
        fp16_mode=False, max_workspace_size=0))
```

After possessing a base backend config file, you can easily construct a complete deploy config through inheritance. Please refer to our [config tutorial](#) for more details. Here is an example:

```
_base_ = ['../_base_/backends/onnxruntime.py']

codebase_config = dict(type='mmcls', task='Classification')
onnx_config = dict(input_shape=None)
```

5. Define APIs in a new package in `mmdeploy/apis`.

Example:

```
# mmdeploy/apis/ncnn/__init__.py

from mmdeploy.backend.ncnn import is_available

__all__ = ['is_available']

if is_available():
    from mmdeploy.backend.ncnn.onnx2ncnn import (onnx2ncnn,
                                                  get_output_model_file)
    __all__ += ['onnx2ncnn', 'get_output_model_file']
```

6. Convert the models of OpenMMLab to backends (if necessary) and inference on backend engine. If you find some incompatible operators when testing, you can try to rewrite the original model for the backend following the [rewriter tutorial](#) or add custom operators.
7. Add docstring and unit tests for new code :).

39.3 Support backend inference

Although the backend engines are usually implemented in C/C++, it is convenient for testing and debugging if the backend provides Python inference interface. We encourage the contributors to support backend inference in the Python interface of MMDeploy. In this section we will introduce the steps to support backend inference.

1. Add a file named `wrapper.py` to corresponding folder in `mmdeploy/backend/{backend}`. For example, `mmdeploy/backend/tensorrt/wrapper.py`. This module should implement and register a wrapper class that inherits the base class `BaseWrapper` in `mmdeploy/backend/base/base_wrapper.py`.

Example:

```
from mmdeploy.utils import Backend
from ..base import BACKEND_WRAPPER, BaseWrapper

@BACKEND_WRAPPER.register_module(Backend.TENSORRT.value)
class TRTWrapper(BaseWrapper):
```

2. The wrapper class can initialize the engine in `__init__` function and inference in `forward` function. Note that the `__init__` function must take a parameter `output_names` and pass it to base class to determine the orders of output tensors. The input and output variables of `forward` should be dictionaries denoting the name and value of the tensors.
3. For the convenience of performance testing, the class should define a “execute” function that only calls the inference interface of the backend engine. The `forward` function should call the “execute” function after pre-processing the data.

Example:

```
from mmdeploy.utils import Backend
from mmdeploy.utils.timer import TimeCounter
from ..base import BACKEND_WRAPPER, BaseWrapper

@BACKEND_WRAPPER.register_module(Backend.ONNXRUNTIME.value)
class ORTWrapper(BaseWrapper):

    def __init__(self,
                 onnx_file: str,
                 device: str,
                 output_names: Optional[Sequence[str]] = None):
        # Initialization
        # ...
        super().__init__(output_names)

    def forward(self, inputs: Dict[str,
                                   torch.Tensor]) -> Dict[str, torch.Tensor]:
        # Fetch data
        # ...

        self.__ort_execute(self.io_binding)

        # Postprocess data
        # ...

    @TimeCounter.count_time('onnxruntime')
    def __ort_execute(self, io_binding: ort.IOBinding):
        # Only do the inference
        self.sess.run_with_iobinding(io_binding)
```

4. Implement `build_wrapper` method in the backend manager.

Example:

```
@BACKEND MANAGERS.register('onnxruntime')
class ONNXRuntimeManager(BaseBackendManager):
```

(continues on next page)

(continued from previous page)

```

@classmethod
def build_wrapper(cls,
                  backend_files: Sequence[str],
                  device: str = 'cpu',
                  input_names: Optional[Sequence[str]] = None,
                  output_names: Optional[Sequence[str]] = None,
                  deploy_cfg: Optional[Any] = None,
                  **kwargs):
    from .wrapper import ORTWrapper
    return ORTWrapper(
        onnx_file=backend_files[0],
        device=device,
        output_names=output_names)

```

5. Add docstring and unit tests for new code :).

39.4 Support new backends using MMDeploy as a third party

Previous parts show how to add a new backend in MMDeploy, which requires changing its source codes. However, if we treat MMDeploy as a third party, the methods above are no longer efficient. To this end, adding a new backend requires us pre-install another package named `aenum`. We can install it directly through `pip install aenum`.

After installing `aenum` successfully, we can use it to add a new backend through:

```

from mmdeploy.utils.constants import Backend
from aenum import extend_enum

try:
    Backend.get('backend_name')
except Exception:
    extend_enum(Backend, 'BACKEND', 'backend_name')

```

We can run the codes above before we use the rewrite logic of MMDeploy.

HOW TO ADD TEST UNITS FOR BACKEND OPS

This tutorial introduces how to add unit test for backend ops. When you add a custom op under `backend_ops`, you need to add the corresponding test unit. Test units of ops are included in `tests/test_ops/test_ops.py`.

40.1 Prerequisite

- **Compile new ops:** After adding a new custom op, needs to recompile the relevant backend, referring to *build.md*.

40.2 1. Add the test program `test_XXXX()`

You can put unit test for ops in `tests/test_ops/`. Usually, the following program template can be used for your custom op.

40.2.1 example of ops unit test

```
@pytest.mark.parametrize('backend', [TEST_TENSORRT, TEST_ONNXRT])           # 1.1 backend_
↪test class
@pytest.mark.parametrize('pool_h,pool_w,spatial_scale,sampling_ratio',      # 1.2 set_
↪parameters of op
                        [(2, 2, 1.0, 2), (4, 4, 2.0, 4)])                  # [(# Examples_
↪of op test parameters),...]
def test_roi_align(backend,
                    pool_h,                                                # set_
↪parameters of op
                    pool_w,
                    spatial_scale,
                    sampling_ratio,
                    input_list=None,
                    save_dir=None):
    backend.check_env()

    if input_list is None:
        input = torch.rand(1, 1, 16, 16, dtype=torch.float32)           # 1.3 op input_
↪data initialization
        single_roi = torch.tensor([[0, 0, 0, 4, 4]], dtype=torch.float32)
    else:
```

(continues on next page)

(continued from previous page)

```

    input = torch.tensor(input_list[0], dtype=torch.float32)
    single_roi = torch.tensor(input_list[1], dtype=torch.float32)

    from mmcv.ops import roi_align

    def wrapped_function(torch_input, torch_rois):
        # 1.4
        ↪ initialize op model to be tested
        return roi_align(torch_input, torch_rois, (pool_w, pool_h),
                        spatial_scale, sampling_ratio, 'avg', True)

    wrapped_model = WrapFunction(wrapped_function).eval()

    with RewriterContext(cfg={}, backend=backend.backend_name, opset=11): # 1.5 call the
    ↪ backend test class interface
        backend.run_and_validate(
            wrapped_model, [input, single_roi],
            'roi_align',
            input_names=['input', 'rois'],
            output_names=['roi_feat'],
            save_dir=save_dir)

```

40.2.2 1.1 backend test class

We provide some functions and classes for difference backends, such as `TestOnnxRTExporter`, `TestTensorRTExporter`, `TestNCNNExporter`.

40.2.3 1.2 set parameters of op

Set some parameters of op, such as 'pool_h', 'pool_w', 'spatial_scale', 'sampling_ratio' in `roi_align`. You can set multiple parameters to test op.

40.2.4 1.3 op input data initialization

Initialization required input data.

40.2.5 1.4 initialize op model to be tested

The model containing custom op usually has two forms.

- `torch` model: Torch model with custom operators. Python code related to op is required, refer to `roi_align` unit test.
- `onnx` model: Onnx model with custom operators. Need to call onnx api to build, refer to `multi_level_roi_align` unit test.

40.2.6 1.5 call the backend test class interface

Call the backend test class `run_and_validate` to run and verify the result output by the op on the backend.

```
def run_and_validate(self,
                    model,
                    input_list,
                    model_name='tmp',
                    tolerate_small_mismatch=False,
                    do_constant_folding=True,
                    dynamic_axes=None,
                    output_names=None,
                    input_names=None,
                    expected_result=None,
                    save_dir=None):
```

Parameter Description

- `model`: Input model to be tested and it can be torch model or any other backend model.
- `input_list`: List of test data, which is mapped to the order of `input_names`.
- `model_name`: The name of the model.
- `tolerate_small_mismatch`: Whether to allow small errors in the verification of results.
- `do_constant_folding`: Whether to use constant light folding to optimize the model.
- `dynamic_axes`: If you need to use dynamic dimensions, enter the dimension information.
- `output_names`: The node name of the output node.
- `input_names`: The node name of the input node.
- `expected_result`: Expected ground truth values for verification.
- `save_dir`: The folder used to save the output files.

40.3 2. Test Methods

Use `pytest` to call the test function to test ops.

```
pytest tests/test_ops/test_ops.py::test_XXXX
```


HOW TO TEST REWRITTEN MODELS

After you create a rewritten model using our *rewriter*, it's better to write a unit test for the model to validate if the model rewrite would come into effect. Generally, we need to get outputs of the original model and rewritten model, then compare them. The outputs of the original model can be acquired directly by calling the forward function of the model, whereas the way to generate the outputs of the rewritten model depends on the complexity of the rewritten model.

41.1 Test rewritten model with small changes

If the changes to the model are small (e.g., only change the behavior of one or two variables and don't introduce side effects), you can construct the input arguments for the rewritten functions/modules run model's inference in `RewriteContext` and check the results.

```
# mmcls.models.classifiers.base.py
class BaseClassifier(BaseModule, metaclass=ABCMeta):
    def forward(self, img, return_loss=True, **kwargs):
        if return_loss:
            return self.forward_train(img, **kwargs)
        else:
            return self.forward_test(img, **kwargs)

# Custom rewritten function
@FUNCTION_REWRITER.register_rewriter(
    'mmcls.models.classifiers.BaseClassifier.forward', backend='default')
def forward_of_base_classifier(ctx, self, img, *args, **kwargs):
    """Rewrite `forward` for default backend."""
    return self.simple_test(img, {})
```

In the example, we only change the function that `forward` calls. We can test this rewritten function by writing the following test function:

```
def test_baseclassifier_forward():
    input = torch.rand(1)
    from mmcls.models.classifiers import BaseClassifier
    class DummyClassifier(BaseClassifier):

        def __init__(self, init_cfg=None):
            super().__init__(init_cfg=init_cfg)

        def extract_feat(self, imgs):
```

(continues on next page)

(continued from previous page)

```

    pass

    def forward_train(self, imgs):
        return 'train'

    def simple_test(self, img, tmp, **kwargs):
        return 'simple_test'

model = DummyClassifier().eval()

model_output = model(input)
with RewriterContext(cfg=dict()), torch.no_grad():
    backend_output = model(input)

assert model_output == 'train'
assert backend_output == 'simple_test'

```

In this test function, we construct a derived class of `BaseClassifier` to test if the rewritten model would work in the rewrite context. We get outputs of the original model by directly calling `model(input)` and get the outputs of the rewritten model by calling `model(input)` in `RewriteContext`. Finally, we can check the outputs by asserting their value.

41.2 Test rewritten model with big changes

In the first example, the output is generated in Python. Sometimes we may make big changes to original model functions (e.g., eliminate branch statements to generate correct computing graph). Even if the outputs of a rewritten model running in Python are correct, we cannot assure that the rewritten model can work as expected in the backend. Therefore, we need to test the rewritten model in the backend.

```

# Custom rewritten function
@FUNCTION_REWRITER.register_rewriter(
    func_name='mmseg.models.segmentors.BaseSegmentor.forward')
def base_segmentor__forward(ctx, self, img, img_metas=None, **kwargs):
    if img_metas is None:
        img_metas = {}
    assert isinstance(img_metas, dict)
    assert isinstance(img, torch.Tensor)

    deploy_cfg = ctx.cfg
    is_dynamic_flag = is_dynamic_shape(deploy_cfg)
    img_shape = img.shape[2:]
    if not is_dynamic_flag:
        img_shape = [int(val) for val in img_shape]
    img_metas['img_shape'] = img_shape
    return self.simple_test(img, img_metas, **kwargs)

```

The behavior of this rewritten function is complex. We should test it as follows:

```

def test_basesegmentor_forward():
    from mmdeploy.utils.test import (WrapModel, get_model_outputs,

```

(continues on next page)

(continued from previous page)

```

get_rewrite_outputs)

segmentor = get_model()
segmentor.cpu().eval()

# Prepare data
# ...

# Get the outputs of original model
model_inputs = {
    'img': [imgs],
    'img metas': [img_metas],
    'return_loss': False
}
model_outputs = get_model_outputs(segmentor, 'forward', model_inputs)

# Get the outputs of rewritten model
wrapped_model = WrapModel(segmentor, 'forward', img_metas = None, return_loss =
↪False)
rewrite_inputs = {'img': imgs}
rewrite_outputs, is_backend_output = get_rewrite_outputs(
    wrapped_model=wrapped_model,
    model_inputs=rewrite_inputs,
    deploy_cfg=deploy_cfg)
if is_backend_output:
    # If the backend plugins have been installed, the rewrite outputs are
    # generated by backend.
    rewrite_outputs = torch.tensor(rewrite_outputs)
    model_outputs = torch.tensor(model_outputs)
    model_outputs = model_outputs.unsqueeze(0).unsqueeze(0)
    assert torch.allclose(rewrite_outputs, model_outputs)
else:
    # Otherwise, the outputs are generated by python.
    assert rewrite_outputs is not None

```

We provide some utilities to test rewritten functions. At first, you can construct a model and call `get_model_outputs` to get outputs of the original model. Then you can wrap the rewritten function with `WrapModel`, which serves as a partial function, and get the results with `get_rewrite_outputs`. `get_rewrite_outputs` returns two values that indicate the content of outputs and whether the outputs come from the backend. Because we cannot assume that everyone has installed the backend, we should check if the results are generated by a Python or backend engine. The unit test must cover both conditions. Finally, we should compare the original and rewritten outputs, which may be done simply by calling `torch.allclose`.

41.3 Note

To learn the complete usage of the test utilities, please refer to our apis document.

HOW TO GET PARTITIONED ONNX MODELS

MMDeploy supports exporting PyTorch models to partitioned onnx models. With this feature, users can define their partition policy and get partitioned onnx models at ease. In this tutorial, we will briefly introduce how to support partition a model step by step. In the example, we would break YOLOV3 model into two parts and extract the first part without the post-processing (such as anchor generating and NMS) in the onnx model.

42.1 Step 1: Mark inputs/outputs

To support the model partition, we need to add Mark nodes in the ONNX model. This could be done with `mmdeploy`'s `@mark` decorator. Note that to make the mark work, the marking operation should be included in a rewriting function.

At first, we would mark the model input, which could be done by marking the input tensor `img` in the forward method of `BaseDetector` class, which is the parent class of all detector classes. Thus we name this marking point as `detector_forward` and mark the inputs as `input`. Since there could be three outputs for detectors such as Mask RCNN, the outputs are marked as `dets`, `labels`, and `masks`. The following code shows the idea of adding mark functions and calling the mark functions in the rewrite. For source code, you could refer to [mmdet-models/detectors/base.py](https://github.com/open-mmlab/mmdetection/blob/master/mmdet/models/detectors/base.py)

```
from mmdeploy.core import FUNCTION_REWRITER, mark

@mark(
    'detector_forward', inputs=['input'], outputs=['dets', 'labels', 'masks'])
def __forward_impl(ctx, self, img, img_metas=None, **kwargs):
    ...

@FUNCTION_REWRITER.register_rewriter(
    'mmdet.models.detectors.base.BaseDetector.forward')
def base_detector__forward(ctx, self, img, img_metas=None, **kwargs):
    ...
    # call the mark function
    return __forward_impl(...)
```

Then, we have to mark the output feature of `YOLOV3Head`, which is the input argument `pred_maps` in `get_bboxes` method of `YOLOV3Head` class. We could add a internal function to only mark the `pred_maps` inside `yolov3_head__get_bboxes` function as following.

```
from mmdeploy.core import FUNCTION_REWRITER, mark

@FUNCTION_REWRITER.register_rewriter(
```

(continues on next page)

(continued from previous page)

```

func_name='mmdet.models.dense_heads.YOLOV3Head.get_bboxes')
def yolov3_head__get_bboxes(ctx,
                             self,
                             pred_maps,
                             img metas,
                             cfg=None,
                             rescale=False,
                             with_nms=True):
    # mark pred_maps
    @mark('yolo_head', inputs=['pred_maps'])
    def __mark_pred_maps(pred_maps):
        return pred_maps
    pred_maps = __mark_pred_maps(pred_maps)
    ...

```

Note that `pred_maps` is a list of `Tensor` and it has three elements. Thus, three Mark nodes with op name as `pred_maps.0`, `pred_maps.1`, `pred_maps.2` would be added in the onnx model.

42.2 Step 2: Add partition config

After marking necessary nodes that would be used to split the model, we could add a deployment config file `configs/mmdet/detection/yolov3_partition_onnxruntime_static.py`. If you are not familiar with how to write config, you could check [write_config.md](#).

In the config file, we need to add `partition_config`. The key part is `partition_cfg`, which contains elements of dict that designates the start nodes and end nodes of each model segments. Since we only want to keep YOLOV3 without post-processing, we could set the start as `['detector_forward:input']`, and end as `['yolo_head:input']`. Note that start and end can have multiple marks.

```

_base_ = ['./detection_onnxruntime_static.py']

onnx_config = dict(input_shape=[608, 608])
partition_config = dict(
    type='yolov3_partition', # the partition policy name
    apply_marks=True, # should always be set to True
    partition_cfg=[
        dict(
            save_file='yolov3.onnx', # filename to save the partitioned onnx model
            start=['detector_forward:input'], # [mark_name:input/output, ...]
            end=['yolo_head:input'], # [mark_name:input/output, ...]
            output_names=[f'pred_maps.{i}' for i in range(3)]) # output names
    ])

```

42.3 Step 3: Get partitioned onnx models

Once we have marks of nodes and the deployment config with `partition_config` being set properly, we could use the *tool* `torch2onnx` to export the model to onnx and get the partition onnx files.

```
python tools/torch2onnx.py \
configs/mmdet/detection/yolov3_partition_onnxruntime_static.py \
../mmdetection/configs/yolo/yolov3_d53_mstrain-608_273e_coco.py \
https://download.openmmlab.com/mmdetection/v2.0/yolo/yolov3_d53_mstrain-608_273e_coco/
↪ yolov3_d53_mstrain-608_273e_coco_20210518_115020-a2c3acb8.pth \
../mmdetection/demo/demo.jpg \
--work-dir ./work-dirs/mmdet/yolov3/ort/partition
```

After run the script above, we would have the partitioned onnx file `yolov3.onnx` in the `work-dir`. You can use the visualization tool `netron` to check the model structure.

With the partitioned onnx file, you could refer to *useful_tools.md* to do the following procedures such as `mmdeploy_onnx2ncnn`, `onnx2tensorrt`.

HOW TO DO REGRESSION TEST

This tutorial describes how to do regression test. The deployment configuration file contains codebase config and inference config.

43.1 1. Python Environment

```
pip install -r requirements/tests.txt
```

If pip throw an exception, try to upgrade numpy.

```
pip install -U numpy
```

43.2 2. Usage

```
python ./tools/regression_test.py \
  --codebase "${CODEBASE_NAME}" \
  --backends "${BACKEND}" \
  [--models "${MODELS}"] \
  --work-dir "${WORK_DIR}" \
  --device "${DEVICE}" \
  --log-level INFO \
  [--performance -p] \
  [--checkpoint-dir "${CHECKPOINT_DIR}"]
```

43.2.1 Description

- `--codebase` : The codebase to test, eg.`mmdet`. If you want to test multiple codebase, use `mmcls mmdet ...`
- `--backends` : The backend to test. By default, all backends would be tested. You can use `onnxruntime` `tesensorrt` to choose several backends. If you also need to test the SDK, you need to configure the `sdk_config` in `tests/regression/${codebase}.yaml`.
- `--models` : Specify the model to be tested. All models in `yaml` are tested by default. You can also give some model names. For the model name, please refer to the relevant `yaml` configuration file. For example `ResNet SE-ResNet "Mask R-CNN"`. Model name can only contain numbers and letters.
- `--work-dir` : The directory of model convert and report, use `../mmdploy_regression_working_dir` by default.

- `--checkpoint-dir`: The path of downloaded torch model, use `../mmdeploy_checkpoints` by default.
- `--device`: device type, use cuda by default
- `--log-level`: These options are available: 'CRITICAL', 'FATAL', 'ERROR', 'WARN', 'WARNING', 'INFO', 'DEBUG', 'NOTSET'. The default value is INFO.
- `-p` or `--performance`: Test precision or not. If not enabled, only model convert would be tested.

43.2.2 Notes

For Windows user:

1. To use the `&&` connector in shell commands, you need to download PowerShell 7 Preview 5+.
2. If you are using conda env, you may need to change `python3` to `python` in `regression_test.py` because there is `python3.exe` in `%USERPROFILE%\AppData\Local\Microsoft\WindowsApps` directory.

43.3 Example

1. Test all backends of mmdet and mmpose for **model convert and precision**

```
python ./tools/regression_test.py \  
--codebase mmdet mmpose \  
--work-dir "../mmdeploy_regression_working_dir" \  
--device "cuda" \  
--log-level INFO \  
--performance
```

2. Test **model convert and precision** of some backends of mmdet and mmpose

```
python ./tools/regression_test.py \  
--codebase mmdet mmpose \  
--backends onnxruntime tensorrt \  
--work-dir "../mmdeploy_regression_working_dir" \  
--device "cuda" \  
--log-level INFO \  
-p
```

3. Test some backends of mmdet and mmpose, **only test model convert**

```
python ./tools/regression_test.py \  
--codebase mmdet mmpose \  
--backends onnxruntime tensorrt \  
--work-dir "../mmdeploy_regression_working_dir" \  
--device "cuda" \  
--log-level INFO
```

4. Test some models of mmdet and mmcls, **only test model convert**

```
python ./tools/regression_test.py \  
--codebase mmdet mmpose \  
--models ResNet SE-ResNet "Mask R-CNN" \  
--work-dir "../mmdeploy_regression_working_dir" \  

```

(continues on next page)

(continued from previous page)

```
--device "cuda" \
--log-level INFO
```

43.4 3. Regression Test Configuration

43.4.1 Example and parameter description

```
globals:
  codebase_dir: ../mmocr # codebase path to test
  checkpoint_force_download: False # whether to redownload the model even if it already_
↪ exists
  images:
    img_densetext_det: &img_densetext_det ../mmocr/demo/demo_densetext_det.jpg
    img_demo_text_det: &img_demo_text_det ../mmocr/demo/demo_text_det.jpg
    img_demo_text_ocr: &img_demo_text_ocr ../mmocr/demo/demo_text_ocr.jpg
    img_demo_text_recog: &img_demo_text_recog ../mmocr/demo/demo_text_recog.jpg
  metric_info: &metric_info
    hmean-iou: # metafile.Results.Metrics
      eval_name: hmean-iou # test.py --metrics args
      metric_key: 0_hmean-iou:hmean # the key name of eval log
      tolerance: 0.1 # tolerated threshold interval
      task_name: Text Detection # the name of metafile.Results.Task
      dataset: ICDAR2015 # the name of metafile.Results.Dataset
    word_acc: # same as hmean-iou, also a kind of metric
      eval_name: acc
      metric_key: 0_word_acc_ignore_case
      tolerance: 0.2
      task_name: Text Recognition
      dataset: IIIT5K
  convert_image_det: &convert_image_det # the image that will be used by detection model_
↪ convert
    input_img: *img_densetext_det
    test_img: *img_demo_text_det
  convert_image_rec: &convert_image_rec
    input_img: *img_demo_text_recog
    test_img: *img_demo_text_recog
  backend_test: &default_backend_test True # whether test model precision for backend
  sdk: # SDK config
    sdk_detection_dynamic: &sdk_detection_dynamic configs/mmocr/text-detection/text-
↪ detection_sdk_dynamic.py
    sdk_recognition_dynamic: &sdk_recognition_dynamic configs/mmocr/text-recognition/
↪ text-recognition_sdk_dynamic.py

onnxruntime:
  pipeline_ort_recognition_static_fp32: &pipeline_ort_recognition_static_fp32
  convert_image: *convert_image_rec # the image used by model conversion
  backend_test: *default_backend_test # whether inference on the backend
  sdk_config: *sdk_recognition_dynamic # test SDK or not. If it exists, use a specific_
↪ SDK config for testing
```

(continues on next page)

(continued from previous page)

```

    deploy_config: configs/mmcocr/text-recognition/text-recognition_onnxruntime_static.py
↪ # the deploy cfg path to use, based on mmdelay path

pipeline_ort_recognition_dynamic_fp32: &pipeline_ort_recognition_dynamic_fp32
    convert_image: *convert_image_rec
    backend_test: *default_backend_test
    sdk_config: *sdk_recognition_dynamic
    deploy_config: configs/mmcocr/text-recognition/text-recognition_onnxruntime_dynamic.py

pipeline_ort_detection_dynamic_fp32: &pipeline_ort_detection_dynamic_fp32
    convert_image: *convert_image_det
    deploy_config: configs/mmcocr/text-detection/text-detection_onnxruntime_dynamic.py

tensorrt:
    pipeline_trt_recognition_dynamic_fp16: &pipeline_trt_recognition_dynamic_fp16
        convert_image: *convert_image_rec
        backend_test: *default_backend_test
        sdk_config: *sdk_recognition_dynamic
        deploy_config: configs/mmcocr/text-recognition/text-recognition_tensorrt-fp16_dynamic-
↪ 1x32x32-1x32x640.py

    pipeline_trt_detection_dynamic_fp16: &pipeline_trt_detection_dynamic_fp16
        convert_image: *convert_image_det
        backend_test: *default_backend_test
        sdk_config: *sdk_detection_dynamic
        deploy_config: configs/mmcocr/text-detection/text-detection_tensorrt-fp16_dynamic-
↪ 320x320-2240x2240.py

openvino:
    # same as onnxruntime backend configuration
ncnn:
    # same as onnxruntime backend configuration
pplnn:
    # same as onnxruntime backend configuration
torchscript:
    # same as onnxruntime backend configuration

models:
- name: crnn # model name
    metafile: configs/textrecog/crnn/metafile.yml # the path of model metafile, based on ↵
↪ codebase path
    codebase_model_config_dir: configs/textrecog/crnn # the basepath of `model_configs`, ↵
↪ based on codebase path
    model_configs: # the config name to teset
        - crnn_academic_dataset.py
    pipelines: # pipeline name
        - *pipeline_ort_recognition_dynamic_fp32

- name: dbnet
    metafile: configs/textdet/dbnet/metafile.yml
    codebase_model_config_dir: configs/textdet/dbnet

```

(continues on next page)

(continued from previous page)

```
model_configs:
- dbnet_r18_fpnc_1200e_icdar2015.py
pipelines:
- *pipeline_ort_detection_dynamic_fp32
- *pipeline_trt_detection_dynamic_fp16

# special pipeline can be added like this
- convert_image: xxx
  backend_test: xxx
  sdk_config: xxx
  deploy_config: configs/mmdcr/text-detection/xxx
```

43.5 4. Generated Report

This is an example of mmocr regression test report.

43.6 5. Supported Backends

- [x] ONNX Runtime
- [x] TensorRT
- [x] PPLNN
- [x] ncnn
- [x] OpenVINO
- [x] TorchScript
- [x] SNPE
- [x] MMDeploy SDK

43.7 6. Supported Codebase and Metrics

ONNX EXPORT OPTIMIZER

This is a tool to optimize ONNX model when exporting from PyTorch.

44.1 Installation

Build MMDeploy with torchscript support:

```
export Torch_DIR=$(python -c "import torch;print(torch.utils.cmake_prefix_path + '/Torch
↪')")

cmake \
  -DTorch_DIR=${Torch_DIR} \
  -DMMDEPLOY_TARGET_BACKENDS="${your_backend};torchscript" \
  .. # You can also add other build flags if you need

cmake --build . -- -j$(nproc) && cmake --install .
```

44.2 Usage

```
# import model_to_graph_custom_optimizer so we can hijack onnx.export
from mmddeploy.apis.onnx.optimizer import model_to_graph_custom_optimizer # noqa
from mmddeploy.core import RewriterContext
from mmddeploy.apis.onnx.passes import optimize_onnx

# load you model here
model = create_model()

# export with ONNX Optimizer
x = create_dummy_input()
with RewriterContext({}, onnx_custom_passes=optimize_onnx):
    torch.onnx.export(model, x, output_path)
```

The model would be optimized after export.

You can also define your own optimizer:

```
# create the optimize callback
def _optimize_onnx(graph, params_dict, torch_out):
```

(continues on next page)

(continued from previous page)

```
from mmdeploy.backend.torchscript import ts_optimizer
ts_optimizer.onnx._jit_pass_onnx_peephole(graph)
return graph, params_dict, torch_out

with RewriterContext({}, onnx_custom_passes=_optimize_onnx):
    # export your model
```

CROSS COMPILE SNPE INFERENCE SERVER ON UBUNTU 18

mmdeploy has provided a prebuilt package, if you want to compile it by self, or need to modify the .proto file, you can refer to this document.

Note that the official gRPC documentation does not have complete support for the NDK.

45.1 1. Environment

45.2 2. Cross compile gRPC with NDK

1. Pull gRPC repo, compile protoc and grpc_cpp_plugin on host

```
# Install dependencies
$ apt-get update && apt-get install -y libssl-dev
# Compile
$ git clone https://github.com/grpc/grpc --recursive=1 --depth=1
$ mkdir -p cmake/build
$ pushd cmake/build

$ cmake \
  -DCMAKE_BUILD_TYPE=Release \
  -DgRPC_INSTALL=ON \
  -DgRPC_BUILD_TESTS=OFF \
  -DgRPC_SSL_PROVIDER=package \
  ../../..
# Install to host
$ make -j
$ sudo make install
```

2. Download the NDK and cross-compile the static libraries with android aarch64 format

```
$ wget https://dl.google.com/android/repository/android-ndk-r17c-linux-x86_64.zip
$ unzip android-ndk-r17c-linux-x86_64.zip

$ export ANDROID_NDK=/path/to/android-ndk-r17c

$ cd /path/to/grpc
$ mkdir -p cmake/build_aarch64 && pushd cmake/build_aarch64

$ cmake ../../.. \
```

(continues on next page)

(continued from previous page)

```

-DCMAKE_TOOLCHAIN_FILE=${ANDROID_NDK}/build/cmake/android.toolchain.cmake \
-DANDROID_ABI=arm64-v8a \
-DANDROID_PLATFORM=android-26 \
-DANDROID_TOOLCHAIN=clang \
-DANDROID_STL=c++_shared \
-DCMAKE_BUILD_TYPE=Release \
-DCMAKE_INSTALL_PREFIX=/tmp/android_grpc_install_shared

$ make -j
$ make install

```

3. At this point /tmp/android_grpc_install should have the complete installation file

```

$ cd /tmp/android_grpc_install
$ tree -L 1
.
├── bin
├── include
├── lib
└── share

```

45.3 3. (Skipable) Self-test whether NDK gRPC is available

1. Compile the helloworld that comes with gRPC

```

$ cd /path/to/grpc/examples/cpp/helloworld/
$ mkdir cmake/build_aarch64 -p && pushd cmake/build_aarch64

$ cmake ../.. \
-DANDROID_TOOLCHAIN_FILE=${ANDROID_NDK}/build/cmake/android.toolchain.cmake \
-DANDROID_ABI=arm64-v8a \
-DANDROID_PLATFORM=android-26 \
-DANDROID_STL=c++_shared \
-DANDROID_TOOLCHAIN=clang \
-DCMAKE_BUILD_TYPE=Release \
-Dabsl_DIR=/tmp/android_grpc_install_shared/lib/cmake/absl \
-DProtobuf_DIR=/tmp/android_grpc_install_shared/lib/cmake/protobuf \
-Dgrpc_DIR=/tmp/android_grpc_install_shared/lib/cmake/grpc

$ make -j
$ ls greeter*
greeter_async_client  greeter_async_server  greeter_callback_server  greeter_server
greeter_async_client2  greeter_callback_client  greeter_client

```

2. Turn on debug mode on your phone, push the binary to /data/local/tmp

```
$ adb push greeter* /data/local/tmp
```

3. adb shell into the phone, execute client/server

```
/data/local/tmp $ ./greeter_client
Greeter received: Hello world
```

45.4 4. Cross compile snpe inference server

1. Open the [snpe tools website](#) and download version 1.59. Unzip and set environment variables

Note that snpe >= 1.60 starts using clang-8.0, which may cause incompatibility with libc++_shared. so on older devices.

```
$ export SNPE_ROOT=/path/to/snpe-1.59.0.3230
```

2. Open the snpe server directory within mmdeploy, use the options when cross-compiling gRPC

```
$ cd /path/to/mmdploy
$ cd service/snpe/server

$ mkdir -p build && cd build
$ export ANDROID_NDK=/path/to/android-ndk-r17c
$ cmake .. \
-DMAKE_TOOLCHAIN_FILE=${ANDROID_NDK}/build/cmake/android.toolchain.cmake \
-DANDROID_ABI=arm64-v8a \
-DANDROID_PLATFORM=android-26 \
-DANDROID_STL=c++_shared \
-DANDROID_TOOLCHAIN=clang \
-DCMAKE_BUILD_TYPE=Release \
-Dabsl_DIR=/tmp/android_grpc_install_shared/lib/cmake/absl \
-DProtobuf_DIR=/tmp/android_grpc_install_shared/lib/cmake/protobuf \
-DgRPC_DIR=/tmp/android_grpc_install_shared/lib/cmake/grpc

$ make -j
$ file inference_server
inference_server: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV),
↳dynamically linked, interpreter /system/bin/linker64,
↳BuildID[sha1]=252aa04e2b982681603dacb74b571be2851176d2, with debug_info, not stripped
```

Finally, you can see infernece_server, adb push it to the device and execute.

45.5 5. Regenerate the proto interface

If you have changed inference.proto, you need to regenerate the .cpp and .py interfaces

```
$ python3 -m pip install grpc_tools --user
$ python3 -m grpc_tools.protoc -I./ --python_out=./client/ --grpc_python_out=./client/
↳inference.proto

$ ln -s `which protoc-gen-grpc`
$ protoc --cpp_out=./ --grpc_out=./ --plugin=protoc-gen-grpc=grpc_cpp_plugin inference.
↳proto
```

45.6 Reference

- snpe tutorial https://developer.qualcomm.com/sites/default/files/docs/snpe/cplus_plus_tutorial.html
- gRPC cross build script https://raw.githubusercontent.com/grpc/grpc/master/test/distrib/cpp/run_distrib_test_cmake_aarch64_cross
- stackoverflow <https://stackoverflow.com/questions/54052229/build-grpc-c-for-android-using-ndk-arm-linux-androideabi-clang-compiler>

FREQUENTLY ASKED QUESTIONS

46.1 TensorRT

- “WARNING: Half2 support requested on hardware without native FP16 support, performance will be negatively affected.”

Fp16 mode requires a device with full-rate fp16 support.

- “error: parameter check failed at: engine.cpp::setBindingDimensions::1046, condition: profileMinDims.d[i] <= dimensions.d[i]”

When building an ICudaEngine from an INetworkDefinition that has dynamically resizable inputs, users need to specify at least one optimization profile. Which can be set in deploy config:

```
backend_config = dict(
    common_config=dict(max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 320, 320],
                    opt_shape=[1, 3, 800, 1344],
                    max_shape=[1, 3, 1344, 1344]))))
    ])
```

The input tensor shape should be limited between min_shape and max_shape.

- “error: [TensorRT] INTERNAL ERROR: Assertion failed: cublasStatus == CUBLAS_STATUS_SUCCESS”

TRT 7.2.1 switches to use cuBLASLt (previously it was cuBLAS). cuBLASLt is the defaulted choice for SM version >= 7.0. You may need CUDA-10.2 Patch 1 (Released Aug 26, 2020) to resolve some cuBLASLt issues. Another option is to use the new TacticSource API and disable cuBLASLt tactics if you dont want to upgrade.

46.2 Libtorch

- Error: libtorch/share/cmake/Caffe2/Caffe2Config.cmake:96 (message):Your installed Caffe2 version uses cuDNN but I cannot find the cuDNN libraries. Please set the proper cuDNN prefixes and / or install cuDNN.

May export CUDNN_ROOT=/root/path/to/cudnn to resolve the build error.

46.3 Windows

- Error: similar like this `OSError: [WinError 1455] The paging file is too small for this operation to complete. Error loading "C:\Users\cx\miniconda3\lib\site-packages\torch\lib\cudnn_cnn_infer64_8.dll" or one of its dependencies`

Solution: according to this [post](#), the issue may be caused by NVidia and will fix in *CUDA release 11.7*. For now one could use the `fixNvPe.py` script to modify the nvidia dlls in the pytorch lib dir.

```
python fixNvPe.py --input=C:\Users\user\AppData\Local\Programs\Python\Python38\lib\
site-packages\torch\lib\*.dll
```

You can find your pytorch installation path with:

```
import torch
print(torch.__file__)
```

- `enable_language(CUDA)` error

```
-- Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.19044.
-- Found CUDA: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.1 (found
↪version "11.1")
CMake Error at C:/Software/cmake/cmake-3.23.1-windows-x86_64/share/cmake-3.23/
↪Modules/CMakeDetermineCompilerId.cmake:491 (message):
  No CUDA toolset found.
Call Stack (most recent call first):
  C:/Software/cmake/cmake-3.23.1-windows-x86_64/share/cmake-3.23/Modules/
↪CMakeDetermineCompilerId.cmake:6 (CMAKE_DETERMINE_COMPILER_ID_BUILD)
  C:/Software/cmake/cmake-3.23.1-windows-x86_64/share/cmake-3.23/Modules/
↪CMakeDetermineCompilerId.cmake:59 (__determine_compiler_id_test)
  C:/Software/cmake/cmake-3.23.1-windows-x86_64/share/cmake-3.23/Modules/
↪CMakeDetermineCUDACompiler.cmake:339 (CMAKE_DETERMINE_COMPILER_ID)
  C:/workspace/mmdploy-0.6.0-windows-amd64-cuda11.1-tensorrt8.2.3.0/sdk/lib/cmake/
↪MMDeploy/MMDeployConfig.cmake:27 (enable_language)
  CMakeLists.txt:5 (find_package)
```

Cause CUDA Toolkit 11.1 was installed before Visual Studio, so the VS plugin was not installed. Or the version of VS is too new, so that the installation of the VS plugin is skipped during the installation of the CUDA Toolkit

Solution This problem can be solved by manually copying the four files in `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.1\extras\visual_studio_integration\MSBuildExtensions` to `C:\Software\Microsoft Visual Studio\2022\Community\MSbuild\Microsoft\VC\v170\BuildCustomizations`. The specific path should be changed according to the actual situation.

46.4 ONNX Runtime

- Under Windows system, when visualizing model inference result failed with the following error:

```
onnxruntime.capi.onnxruntime_pybind11_state.Fail: [ONNXRuntimeError] : 1 : FAIL :
↪Failed to load library, error code: 193
```

Cause In latest Windows systems, there are two `onnxruntime.dll` under the system path, and they will be loaded first, causing conflicts.

```
C:\Windows\SysWOW64\onnxruntime.dll  
C:\Windows\System32\onnxruntime.dll
```

Solution Choose one of the following two options

1. Copy the dll in the lib directory of the downloaded onnxruntime to the directory where mmdeploy_onnxruntime_ops.dll locates (It is recommended to use Everything to search the ops dll). For example, copy `onnxruntime lib/onnxruntime.dll` to `mmdeploy/lib`, then the `mmdeploy/lib` directory should like this

```
`-- mmdeploy_onnxruntime_ops.dll  
`-- mmdeploy_onnxruntime_ops.lib  
`-- onnxruntime.dll
```

2. Rename the two dlls in the system path so that they cannot be loaded.

46.5 Pip

- pip installed package but could not import them.

Make sure your are using conda pip.

```
$ which pip  
# /path/to/.local/bin/pip  
/path/to/miniconda3/lib/python3.9/site-packages/pip
```

CHAPTER
FORTYSEVEN

ENGLISH

CHAPTER
FORTYEIGHT

CHAPTER
FORTYNINE

APIS

APIS/TENSORRT

APIS/ONNXRUNTIME

CHAPTER
FIFTYTHREE

APIS/PPLNN

INDICES AND TABLES

- `genindex`
- `search`