mmdeploy Documentation

发布 *0.7.0*

MMDeploy Contributors

2022 年 08 月 12 日

快速上手

点击页面左下角切换中英文。

CHAPTER 1

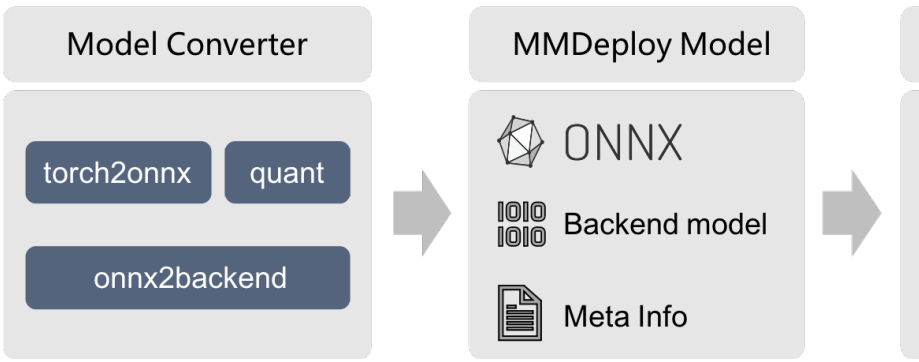
操作概述

MMDeploy 提供了一系列工具，帮助您更轻松的将 OpenMMLab 下的算法部署到各种设备与平台上。

您可以使用我们设计的流程一“部”到位，也可以定制您自己的转换流程。

1.1 流程简介

MMDeploy 定义的模型部署流程，如下图所示：



1.1.1 模型转换（Model Converter）

模型转换的主要功能是把输入的模型格式，转换为目标设备的推理引擎所要求的模型格式。

目前，MMDeploy 可以把 PyTorch 模型转换为 ONNX、TorchScript 等和设备无关的 IR 模型。也可以将 ONNX 模型转换为推理后端模型。两者相结合，可实现端到端的模型转换，也就是从训练端到生产端的一键式部署。

1.1.2 MMDeploy 模型（MMDeploy Model）

也称 SDK Model。它是模型转换结果的集合。不仅包括后端模型，还包括模型的元信息。这些信息将用于推理 SDK 中。

1.1.3 推理 SDK（Inference SDK）

封装了模型的前处理、网络推理和后处理过程。对外提供多语言的模型推理接口。

1.2 准备工作

对于端到端的模型转换和推理，MMDeploy 依赖 Python 3.6+ 以及 PyTorch 1.8+。

第一步：从官网下载并安装 Miniconda

第二步：创建并激活 conda 环境

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

第三步: 参考官方文档并安装 PyTorch

在 GPU 环境下：

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

在 CPU 环境下：

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

注解： 在 GPU 环境下，请务必保证 {cudatoolkit_version} 和主机的 CUDA Toolkit 版本一致，避免在使用 TensorRT 时，可能引起的版本冲突问题。

1.3 安装 MMDeploy

第一步：通过 MIM 安装 MMCV

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

第二步: 安装 MMDeploy 和推理引擎

我们推荐用户使用预编译包安装和体验 MMDeploy 功能。请根据目标软硬件平台，从这里 选择最新版本下载并安装。

目前，MMDeploy 的预编译包支持的平台和设备矩阵如下：

注：对于不在上述表格中的软硬件平台，请参考源码安装文档，正确安装和配置 MMDeploy。

以最新的预编译包为例，你可以参考以下命令安装：

```
# 安装 MMDeploy ONNX Runtime 自定义算子库和推理 SDK
wget https://github.com/open-mmlab/mmdeploy/releases/download/v0.7.0/mmdeploy-0.7.0-
↪linux-x86_64-onnxruntime1.8.1.tar.gz
tar -zxvf mmdeploy-0.7.0-linux-x86_64-onnxruntime1.8.1.tar.gz
cd mmdeploy-0.7.0-linux-x86_64-onnxruntime1.8.1
pip install dist/mmdeploy-0.7.0-py3-none-linux_x86_64.whl
pip install sdk/python/mmdeploy_python-0.7.0-cp38-none-linux_x86_64.whl
cd ..
# 安装推理引擎 ONNX Runtime
pip install onnxruntime==1.8.1
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
```

```
# 安装 MMDeploy TensorRT 自定义算子库和推理 SDK
wget https://github.com/open-mmlab/mmdeploy/releases/download/v0.7.0/mmdeploy-0.7.0-
↪linux-x86_64-cuda11.1-tensorrt8.2.3.0.tar.gz
tar -zxvf mmdeploy-v0.7.0-linux-x86_64-cuda11.1-tensorrt8.2.3.0.tar.gz
cd mmdeploy-0.7.0-linux-x86_64-cuda11.1-tensorrt8.2.3.0
pip install dist/mmdeploy-0.7.0-py3-none-linux_x86_64.whl
pip install sdk/python/mmdeploy_python-0.7.0-cp38-none-linux_x86_64.whl
cd ..
# 安装推理引擎 TensorRT
# !!! 从 NVIDIA 官网下载 TensorRT-8.2.3.0 CUDA 11.x 安装包并解压到当前目录
pip install TensorRT-8.2.3.0/python/tensorrt-8.2.3.0-cp38-none-linux_x86_64.whl
pip install pycuda
```

(下页继续)

(续上页)

```
export TENSORRT_DIR=$(pwd)/TensorRT-8.2.3.0
export LD_LIBRARY_PATH=${TENSORRT_DIR}/lib:$LD_LIBRARY_PATH
# !!! 从 NVIDIA 官网下载 cuDNN 8.2.1 CUDA 11.x 安装包并解压到当前目录
export CUDNN_DIR=$(pwd)/cuda
export LD_LIBRARY_PATH=$CUDNN_DIR/lib64:$LD_LIBRARY_PATH
```

请阅读 这里，了解 MMDeploy 预编译包在 Windows 平台下的使用方法。

1.4 模型转换

在准备工作就绪后，我们可以使用 MMDeploy 中的工具 `tools/deploy.py`，将 OpenMMLab 的 PyTorch 模型转换成推理后端支持的格式。对于 `tools/deploy.py` 的使用细节，请参考如何转换模型。

以 MMDetection 中的 `Faster R-CNN` 为例，我们可以使用如下命令，将 PyTorch 模型转换为 TenorRT 模型，从而部署到 NVIDIA GPU 上.

```
# 克隆 mmdeploy 仓库。转换时，需要使用 mmdeploy 仓库中的配置文件，建立转换流水线
git clone --recursive https://github.com/open-mmlab/mmdeploy.git

# 安装 mmdetection。转换时，需要使用 mmdetection 仓库中的模型配置文件，构建 PyTorch nn module
git clone https://github.com/open-mmlab/mmdetection.git
cd mmdetection
pip install -v -e .
cd ..

# 下载 Faster R-CNN 模型权重
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

# 执行转换命令，实现端到端的转换
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
```

转换结果被保存在 `--work-dir` 指向的文件夹中。**该文件夹中不仅包含推理后端模型，还包括推理元信息。这些内容的整体被定义为 SDK Model。推理 SDK 将用它进行模型推理。**

小技巧： 在安装了 MMDeploy-ONNXRuntime 预编译包后，把上述转换命令中的 detection_tensorrt_dynamic-

320x320-1344x1344.py 换成 detection_onnxruntime_dynamic.py，并修改 --device 为 cpu，即可以转出 onnx 模型，并用 ONNXRuntime 进行推理

1.5 模型推理

在转换完成后，你既可以使用 Model Converter 进行推理，也可以使用 Inference SDK。

1.5.1 使用 Model Converter 的推理 API

Model Converter 屏蔽了推理后端接口的差异，对其推理 API 进行了统一封装，接口名称为 `inference_model`。

以上文中 Faster R-CNN 的 TensorRT 模型为例，你可以使用如下方式进行模型推理工作：

```
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')
```

注解： 接口中的 model_path 指的是推理引擎文件的路径，比如例子当中 end2end.engine 文件的路径。路径必须放在 list 中，因为有的推理引擎模型结构和权重是分开存储的。

1.5.2 使用推理 SDK

你可以直接运行预编译包中的 demo 程序，输入 SDK Model 和图像，进行推理，并查看推理结果。

```
cd mmdeploy-0.7.0-linux-x86_64-cuda11.1-tensorrt8.2.3.0
# 运行 python demo
python sdk/example/python/object_detection.py cuda ../mmdeploy_model/faster-rcnn ../
↪mmdetection/demo/demo.jpg
# 运行 C/C++ demo
export LD_LIBRARY_PATH=$(pwd)/sdk/lib:$LD_LIBRARY_PATH
./sdk/bin/object_detection cuda ../mmdeploy_model/faster-rcnn ../mmdetection/demo/
↪demo.jpg
```

注解： 以上述命令中，输入模型是 SDK Model 的路径（也就是 Model Converter 中 --work-dir 参数），而不是推理引擎文件的路径。因为 SDK 不仅要获取推理引擎文件，还需要推理元信息（deploy.json, pipeline.json）。它们合在一起，构成 SDK Model，存储在 --work-dir 下

除了 demo 程序，预编译包还提供了 SDK 多语言接口。你可以根据自己的项目需求，选择合适的语言接口，把 MMDeploy SDK 集成到自己的项目中，进行二次开发。

Python API

对于检测功能，你也可以参考如下代码，集成 MMDeploy SDK Python API 到自己的项目中：

```
from mmdeploy_python import Detector
import cv2

# 读取图片
img = cv2.imread('mmdetection/demo/demo.jpg')

# 创建检测器
detector = Detector(model_path='mmdeploy_models/faster-rcnn', device_name='cuda',␣
↪device_id=0)
# 执行推理
bboxes, labels, _ = detector(img)
# 使用阈值过滤推理结果，并绘制到原图中
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)
```

更多示例，请查阅这里。

C++ API

使用 C++ API 进行模型推理的流程符合下面的模式：

以下是具体过程：

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

int main() {
  const char* device_name = "cuda";
  int device_id = 0;

  // mmdeploy SDK model，以上文中转出的 faster r-cnn 模型为例
  std::string model_path = "mmdeploy_model/faster-rcnn";
  std::string image_path = "mmdetection/demo/demo.jpg";

  // 1. 读取模型
  mmdeploy::Model model(model_path);
  // 2. 创建预测器
  mmdeploy::Detector detector(model, mmdeploy::Device{device_name, device_id});
  // 3. 读取图像
  cv::Mat img = cv::imread(image_path);
  // 4. 应用预测器推理
  auto dets = detector.Apply(img);
  // 5. 处理推理结果：此处我们选择可视化推理结果
  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,␣
↪score=%.4f\n",
            i, box.left, box.top, box.right, box.bottom, dets[i].label_id, dets[i].
↪score);
    if (bboxes[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;
}
```

在您的项目 CMakeLists 中，增加：

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

编译时，使用 -DMMDeploy_DIR，传入 MMDeloyConfig.cmake 所在的路径。它在预编译包中的 sdk/lib/cmake/MMDeloy 下。更多示例，请查阅此处。

对于 C API、C# API、Java API 的使用方法，请分别阅读代码C demos，C# demos 和 Java demos。我们将在后续版本中详细讲述它们的用法。

1.6 模型精度评估

为了测试部署模型的精度，推理效率，我们提供了 `tools/test.py` 来帮助完成相关工作。以上文中的部署模型为例：

```
python mmdeploy/tools/test.py \
    mmdeploy/configs/detection/detection_tensorrt_dynamic-320x320-1344x1344.py \
    mmdetection/configs/faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py \
    --model mmdeploy_model/faster-rcnn/end2end.engine \
    --metrics ${METRICS} \
    --device cuda:0
```

注解： 关于 --model 选项，当使用 Model Converter 进行推理时，它代表转换后的推理后端模型的文件路径。而当使用 SDK 测试模型精度时，该选项表示 MMDeploy Model 的路径.

请阅读如何进行模型评估 了解关于 `tools/test.py` 的使用细节。

CHAPTER 2

源码安装

如果环境允许（网络良好且宿主机强劲），我们建议使用*docker 方式* 。

2.1 下载

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

2.1.1 FAQ

- 如果由于网络等原因导致拉取仓库子模块失败，可以尝试通过如下指令手动再次安装子模块:

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

# 返回至 third_party 目录, 克隆 pybind11
cd ..
git clone git@github.com:pybind/pybind11.git pybind11
cd pybind11
git checkout 70a58c5
```

- 如果以 SSH 方式 `git clone` 代码失败，您可以尝试使用 HTTPS 协议下载代码:

```
git clone -b master https://github.com/open-mmlab/mmdeploy.git MMDeploy
cd MMDeploy
git submodule update --init --recursive
```

2.2 编译

根据您的目标平台，点击如下对应的链接，按照说明编译 MMDeploy

- Linux-x86_64
- Windows
- Android-aarch64
- NVIDIA Jetson
- Qcom SNPE

CHAPTER 3

使用 Docker 镜像

我们分别为 CPU 和 GPU 提供了两个 dockerfile。对于 CPU 用户，我们对接 ONNXRuntime、ncnn 和 OpenVINO 后端安装 MMDeploy。对于 GPU 用户，我们安装带有 TensorRT 后端的 MMDeploy。此外，用户可以在构建 docker 镜像时安装不同版本的 mmdeploy。

3.1 构建镜像

对于 CPU 用户，我们可以通过以下方式使用最新的 MMDeploy 构建 docker 镜像：

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

对于 GPU 用户，我们可以通过以下方式使用最新的 MMDeploy 构建 docker 镜像：

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

要安装具有特定版本的 MMDeploy，我们可以将 `--build-arg VERSION=${VERSION}` 附加到构建命令中。以 GPU 为例：

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

要切换成阿里源安装依赖，我们可以将 `--build-arg USE_SRC_INSIDE=${USE_SRC_INSIDE}` 附加到构建命令中。

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

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

3.2 运行 docker 容器

构建 docker 镜像成功后，我们可以使用 `docker run` 启动 docker 服务。GPU 镜像为例：

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

3.3 常见问答

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

   如 这里所说，更新 GPU 的驱动到您的 GPU 能使用的最新版本。

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
   ```

CHAPTER 4

如何转换模型

这篇教程介绍了如何使用 MMDeploy 的工具将一个 OpenMMlab 模型转换成某个后端的模型文件。

注意:

- 现在已支持的后端包括 ONNX Runtime， TensorRT， ncnn， PPLNN, OpenVINO。
- 现在已支持的代码库包括 MMClassification， MMDetection， MMSegmentation， MMOCR， MMEditing。

4.1 如何将模型从 pytorch 形式转换成其他后端形式

4.1.1 准备工作

1. 安装您的目标后端。您可以参考 ONNXRuntime-install ， TensorRT-install ， ncnn-install ， PPLNN-install, OpenVINO-install。
2. 安装您的目标代码库。您可以参考 MMClassification-install，MMDetection-install，MMSegmentation-install，MMOCR-install，MMEditing-install。

4.1.2 使用方法

```
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
```

4.1.3 参数描述

- `deploy_cfg` : MMDeploy 中用于部署的配置文件路径。
- `model_cfg` : OpenMMLab 系列代码库中使用的模型配置文件路径。
- `checkpoint` : OpenMMLab 系列代码库的模型文件路径。
- `img` : 用于模型转换时使用的图像文件路径。
- `--test-img` : 用于测试模型的图像文件路径。默认设置成 `None`。
- `--work-dir` : 工作目录，用来保存日志和模型文件。
- `--calib-dataset-cfg` : 此参数只有 int8 模式下生效，用于校准数据集配置文件。若在 int8 模式下未传入参数，则会自动使用模型配置文件中的'val' 数据集进行校准。
- `--device` : 用于模型转换的设备。默认是 `cpu`。
- `--log-level` : 设置日记的等级，选项包括`'CRITICAL','FATAL','ERROR','WARN','WARNING', 'INFO', 'DEBUG', 'NOTSET'`。默认是 `INFO`。

- `--show` : 是否显示检测的结果。
- `--dump-info` : 是否输出 SDK 信息。

4.1.4 如何查找 pytorch 模型对应的部署配置文件

1. 在 `configs/` 文件夹中找到模型对应的代码库文件夹。例如，转换一个 yolov3 模型您可以查找到 `configs/mmdet` 文件夹。
2. 根据模型的任务类型在 `configs/codebase_folder/` 下查找对应的文件夹。例如 yolov3 模型，您可以查找到 `configs/mmdet/detection` 文件夹。
3. 在 `configs/codebase_folder/task_folder/` 下找到模型的部署配置文件。例如部署 yolov3 您可以使用 `configs/mmdet/detection/detection_onnxruntime_dynamic.py`。

4.1.5 示例

```
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
```

4.2 如何评测模型

您可以尝试去评测转换出来的模型，参考*profile 模型*。

4.3 各后端已支持导出的模型列表

参考已支持的模型列表。

CHAPTER 5

如何写模型转换配置

这篇教程介绍了如何编写模型转换和部署的配置文件。部署配置文件由 ONNX 配置，代码库配置，推理框架配置组成。

- 7. 注意事项
- 8. 常见问题

5.1 1. 如何编写 ONNX 配置

ONNX 配置描述了如何将 PyTorch 模型转换为 ONNX 模型。

5.1.1 ONNX 配置参数说明

- `type`: 配置类型。默认为 `onnx`。
- `export_params`: 如果指定，将导出模型所有参数。如果您只想导出未训练模型将此项设置为 False。
- `keep_initializers_as_inputs`: 如果为 True，则所有初始化器（通常对应为参数）也将作为输入导出，添加到计算图中。如果为 False，则初始化器不会作为输入导出，不添加到计算图中，仅将非参数输入添加到计算图中。
- `opset_version`: ONNX 的算子集版本，默认为 11。
- `save_file`: 输出 ONNX 模型文件。
- `input_names`: 模型计算图中输入节点的名称。
- `output_names`: 模型计算图中输出节点的名称。
- `input_shape`: 模型输入张量的高度和宽度。

示例

```
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)
```

5.1.2 动态尺寸输入和输出配置

如果模型要求动态尺寸的输入和输出，您需要在 ONNX 配置中加入 dynamic_axes 配置。

- `dynamic_axes`: 描述输入和输出的维度信息。

示例

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

5.2 2. 如何编写代码库配置

代码库主要指 OpenMMLab 系列模型代码库，代码库配置由 OpenMMLab 系列模型代码库的简称和 OpenMMLab 系列模型任务类型组成。

5.2.1 代码库配置参数说明

- `type`: OpenMMLab 系列模型代码库的简称，包括 `mmcls`，`mmdet`，`mmseg`，`mmocr`，`mmedit`。
- `task`: OpenMMLab 系列模型任务类型，具体请参考 OpenMMLab 系列模型任务列表。

示例

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

5.3 3. 如何编写推理框架配置

推理框架配置主要用于指定模型运行在哪个推理框架，并提供模型在推理框架运行时所需的信息，具体参考 ONNX Runtime，TensorRT，ncnn，PPLNN。

- type: 模型推理框架, 包括 onnxruntime，ncnn，pplnn，tensorrt，openvino。

5.3.1 示例

```
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])))
    ])
```

5.4 4. 部署配置完整示例

这里我们提供了一个以 TensorRT 为推理框架的基于 mmcls 图像分类任务的完整部署配置示例。

```
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(
```

(下页继续)

(续上页)

```
            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])
```

5.5 5. 部署配置文件命名规则

我们遵循以下样式来命名配置文件。建议贡献者遵循相同的风格。

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

- `task name`: 模型任务类型。
- `backend name`: 推理框架名称。注意：如果您使用了量化，您需要指出量化类型。例如 `tensorrt-int8`。
- `dynamic or static`: 动态或者静态尺寸导出。注意：如果推理框架需要明确的形状信息，您需要添加输入大小的描述，格式为高度 x 宽度。例如 `dynamic-512x1024-2048x2048`, 这意味着最小输入形状是 `512x1024`，最大输入形状是 `2048x2048`。

5.5.1 示例

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

5.6 6. 如何编写模型配置文件

请根据模型具体任务的代码库，编写模型配置文件。模型配置文件用于初始化模型，详情请参考MMClassification，MMDetection，MMSegmentation，MMOCR，MMEditing。

CHAPTER 6

如何 Profile 模型

模型转换结束后，MMDeploy 提供了 `tools/test.py` 做为单测工具。

6.1 依赖

需要参照安装说明 完成依赖安装，按照转换说明 转出模型。

6.2 用法

```
python tools/test.py \
${DEPLOY_CFG} \
${MODEL_CFG} \
--model ${BACKEND_MODEL_FILES} \
[--speed-test] \
[--warmup ${WARM_UP}] \
[--log-interval ${LOG_INTERVERL}] \
[--log2file ${LOG_RESULT_TO_FILE}]
```

6.3 参数详解

6.4 使用样例

执行模型推理

```
python tools/test.py \
    configs/mmcls/classification_onnxruntime_static.py \
    {MMCLS_DIR}/configs/resnet/resnet50_b32x8_imagenet.py \
    --model model.onnx \
    --out out.pkl \
    --device cuda:0
```

profile 速度测试

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

CHAPTER 7

如何量化模型

7.1 为什么要量化

相对于 fp32 模型，定点模型有诸多优点：

- 体积更小，8-bit 模型可降低 75% 文件大小
- 由于模型变小，Cache 命中率提升，速度更快
- 芯片往往有对应的定点加速指令，这些指令更快、能耗更低（常见 CPU 上 int8 大约只需要 10% 能量）

安装包体积、发热都是移动端评价 APP 的关键指标；而在服务端，“加速”意味着可以维持相同 QPS、增大模型换取精度提升。

7.2 mmdeploy 离线量化方案

以 ncnn backend 为例，完整的工作流如下：

mmdeploy 基于静态图（onnx）生成推理框架所需的量化表，再用后端工具把浮点模型转为定点。

目前 mmdeploy 支持 ncnn PTQ。

7.3 模型怎么转定点

*mmdeploy 安装*完成后，加载 ppq 并安装

```
git clone https://github.com/openppl-public/ppq.git
cd ppq
git checkout edbecf4 # 需要一些特性和修复
pip install -r requirements.txt
python3 setup.py install
```

回到 mmdeploy, 使用 `tools/deploy.py --quant` 选项开启量化。

```
cd /path/to/mmdeploy
export MODEL_CONFIG=/path/to/mmclassification/configs/resnet/resnet18_8xb16_cifar10.py
export MODEL_PATH=https://download.openmmlab.com/mmclassification/v0/resnet/resnet18_
↪b16x8_cifar10_20210528-bd6371c8.pth

python3 tools/deploy.py  configs/mmcls/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/images
...
```

参数说明

7.4 自建校准数据集

校准集是用来计算量化层参数的，某些 DFQ（Data Free Quantization）方法甚至不需要校准集

- 新建文件夹，直接放入图片即可（不需要目录结构、不要负例、没有命名要求）
- 图片需为真实业务场景中的数据，相差过远会导致精度下降
- 不能直接拿测试集做量化，否则是过拟合

类型	训练集	验证集	测试集	校准集
用法	QAT	PTQ	测试精度	PTQ

强烈建议量化结束后，按此文档验证模型精度。这里是一些量化模型测试结果。

CHAPTER 8

更多工具介绍

除 deploy.py 以外，tools 目录下有很多实用工具

8.1 torch2onnx

把 OpenMMLab 模型转 onnx 格式。

8.1.1 用法

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

8.1.2 参数说明

- 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.

8.2 extract

有 Mark 节点的 onnx 模型会被分成多个子图，这个工具用来提取 onnx 模型中的子图。

8.2.1 用法

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

8.2.2 参数说明

- 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.

8.2.3 注意事项

要支持模型分块，必须在 onnx 模型中添加 mark 节点，用 `@mark` 修饰。下面这个例子里 mark 了 `multiclass_nms`，在 NMS 前设置 `end=multiclass_nms:input` 提取子图。

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

8.3 onnx2pplnn

这个工具可以把 onnx 模型转成 pplnn 格式。

8.3.1 用法

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

8.3.2 参数说明

- `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`.

8.4 onnx2tensorrt

这个工具把 onnx 转成 trt .engine 格式。

8.4.1 用法

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

8.4.2 参数说明

- `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`.

8.5 onnx2ncnn

onnx 转 ncnn

8.5.1 用法

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

8.5.2 参数说明

- 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.

8.6 profile

这个工具用来测试 torch 和 trt 等后端的速度，注意测试不包含前后处理。

8.6.1 用法

```
python tools/profile.py \
    ${DEPLOY_CFG} \
    ${MODEL_CFG} \
    ${IMAGE_DIR} \
    --model ${MODEL} \
    --device ${DEVICE} \
    --shape ${SHAPE} \
    --num-iter {NUM_ITER} \
    --warmup {WARMUP}
    --cfg-options ${CFG_OPTIONS}
```

8.6.2 参数说明

- 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.

8.6.3 使用举例

```
python tools/profile.py \
    configs/mmcls/classification_tensorrt_dynamic-224x224-224x224.py \
    ../mmclassification/configs/resnet/resnet18_8xb32_in1k.py \
    ../mmdetection/demo \
    --model work-dirs/mmcls/resnet/trt/end2end.engine \
    --device cuda \
    --shape 224x224 \
    --num-iter 100 \
    --warmup 10 \
```

输出：

```
----- 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 |
+--------+------------+---------+
```

CHAPTER 9

模型支持列表

自测完成的 model-backend 组合：

9.1 Note

- Tag:
  - static: This model only support static export. Please use `static` deploy config, just like $MMDEPLOY_DIR/configs/mmseg/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/detection_tensorrt_dynamic-300x300-512x512.py.
- YOLOX: YOLOX with ncnn only supports static shape.
- Swin Transformer: For TensorRT, only version 8.4+ is supported.

CHAPTER 10

精度速度测试结果

10.1 Backends

CPU: ncnn, ONNXRuntime, OpenVINO

GPU: ncnn, TensorRT, PPLNN

10.2 软硬件环境

- 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

10.2.1 配置

- 静态图导出
- batch 大小为 1
- 测试时，计算各个数据集中 100 张图片的平均耗时

用户可以直接通过*model profiling*获得想要的速度测试结果。下面是我们环境中的测试结果：

10.3 速度测试

10.4 精度测试

10.5 备注

- 由于某些数据集在代码库中包含各种分辨率的图像，例如 MMDet，速度基准是通过 MMDeploy 中的静态配置获得的，而性能基准是通过动态配置获得的
- TensorRT 的一些 int8 性能基准测试需要有 tensor core 的 Nvidia 卡，否则性能会大幅下降
- DBNet 在模型 `neck` 使用了 `nearest` 插值，TensorRT-7 用了与 Pytorch 完全不同的策略。为了使与 TensorRT-7 兼容，我们重写了 `neck` 以使用 `bilinear` 插值，这提高了检测性能。为了获得与 Pytorch 匹配的性能，推荐使用 TensorRT-8+，其插值方法与 Pytorch 相同。
- 对于 mmpose 模型，在模型配置文件中 `flip_test` 需设置为 `False`
- 部分模型在 fp16 模式下可能存在较大的精度损失，请根据具体情况对模型进行调整。

CHAPTER 11

边、端设备测试结果

这里给出我们边、端设备的测试结论，用户可以直接通过*model profiling* 获得自己环境的结果。

11.1 软硬件环境

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

11.2 mmcls 模型

tips:

1. ImageNet-1k 数据集较大，仅使用一部分测试（8000/50000）
2. 边、端设备发热会降频，因此耗时实际上会波动。这里给出运行一段时间后、稳定的数值。这个结果更贴近实际需求

11.3 mmocr 检测

11.4 mmpose 模型

tips:

- 测试 pose_hrnet 用的是 AnimalPose 的 test dataset，而非 val dataset

11.5 mmseg

tips:

- fcn 用 512x1024 尺寸运行正常。Cityscapes 数据集 1024x2048 分辨率会导致设备重启

11.6 其他模型

- mmdet 需要手动把模型拆成两部分。因为
  - snpe 源码中 `onnx_to_ir.py` 仅能解析输入，`ir_to_dlc.py` 还不支持 topk
  - UDO （用户自定义算子）无法和 `snpe-onnx-to-dlc` 配合使用
- mmedit 模型
  - srcnn 需要 cubic resize，snpe 不支持
  - esrgan 可正常转换，但加载模型会导致设备重启
- mmrotate 依赖 e2cnn ，需要手动安装 其 Python3.6 兼容分支

CHAPTER 12

量化测试结果

目前 mmdeploy 支持 ncnn 量化

12.1 ncnn 量化

12.1.1 分类任务

备注：

- 因为 imagenet-1k 数据量很大、ncnn 未正式发布 Vulkan int8 版本，考虑到 CPU 运行时间，仅用部分测试集（4000/50000）
- 量化后精度会有差异，分类模型涨点 1% 以内是正常情况

12.1.2 OCR 检测任务

备注：mmocr 使用 `shapely` 计算 IoU，实现方法会导致轻微的精度差异

CHAPTER 13

mmcls 模型支持列表

MMClassification 是基于 Python 的的图像分类工具，属于 OpenMMLab。

13.1 安装 mmcls

请参考 install.md 进行安装。

13.2 支持列表

CHAPTER 14

mmdet 模型支持列表

mmdet 是基于 pytorch 的检测工具箱，属于 OpenMMLab。

14.1 安装 mmdet

请参照 get_started.md 。

14.2 支持列表

CHAPTER 15

mmdet3d 模型支持列表

MMDetection3d 是用于通用 3D 物体检测平台。属于 OpenMMLab。

15.1 安装 mmdet3d

参照 getting_started.md。

15.2 示例

```
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 \
       checkpoints/point_pillars.pth \
       ${MMDET3D_DIR}/demo/data/kitti/kitti_000008.bin \
        --work-dir \
        work_dir \
        --show \
        --device \
        cuda:0
```

15.3 支持列表

15.4 注意事项

体素检测 onnx 模型不包含 model.voxelize 层和模型后处理，可用 python api 来调这些函数。

示例：

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

CHAPTER 16

mmedit 模型支持列表

mmedit 是基于 PyTorch 的开源图像和视频编辑工具箱，属于 OpenMMLab。

16.1 安装 mmedit

参照 official installation guide。

16.2 支持列表

CHAPTER 17

mmocr 模型支持列表

mmocr 是一个基于 PyTorch 和 mmdetection 的开源工具箱，用于文本检测，文本识别以及相应的下游任务，例如关键信息提取，是 OpenMMLab项目的一部分。

17.1 安装

参照 install.md。

17.2 支持列表

17.3 注意事项

请注意，ncnn、pplnn 和 OpenVINO 仅支持 DBNet 的 DBNet18 配置。

对于在 ICDAR 数据集上预训 checkpoint 的 PANet，如果要将模型转为具有 fp16 TensorRT，请尝试以下脚本。

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

import torch
import torch.nn.functional as F
```

(下页继续)

(续上页)

```
from mmdeploy.core import FUNCTION_REWRITER
from mmdeploy.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]:
    """Rewrite `forward` of FPEM_FFM for tensorrt backend.

    Rewrite this function avoid overflow for tensorrt-fp16 with the checkpoint
    `https://download.openmmlab.com/mmocr/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) * (
```

(下页继续)

(续上页)

```
            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,
        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:
```

(下页继续)

(续上页)

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

    Rewrite this function avoid overflow for tensorrt-fp16 with the checkpoint
    `https://download.openmmlab.com/mmocr/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:
        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
```

CHAPTER 18

mmpose 模型支持列表

mmpose 是一个基于 PyTorch 的姿态估计的开源工具箱，也是 OpenMMLab 项目的一部分。

18.1 安装 mmpose

参照 official installation guide。

18.2 支持列表

18.2.1 使用方法

```
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
```

注意事项

- mmpose 模型需要额外的输入，但我们无法直接获取它。在导出模型时，可以使用 `$MMDEPLOY_DIR/demo/resources/human-pose.jpg` 作为输入。

CHAPTER 19

mmrotate 模型支持列表

mmrotate 是一个基于 PyTorch 的旋转物体检测的开源工具箱，也是 OpenMMLab 项目的一部分。

19.1 安装 mmrotate

参照 official installation guide。

19.2 支持列表

19.2.1 使用举例

```
# convert ort
python tools/deploy.py \
configs/mmrotate/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/mmrotate/rotated_retinanet/ort \
--device cpu

# compute metric
```

(下页继续)

(续上页)

```
python tools/test.py \
    configs/mmrotate/rotated-detection_onnxruntime_dynamic.py \
    $MMROTATE_DIR/configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_
↪le135.py \
    --model work-dirs/mmrotate/rotated_retinanet/ort/end2end.onnx \
    --metrics mAP

# generate submit file
python tools/test.py \
    configs/mmrotate/rotated-detection_onnxruntime_dynamic.py \
    $MMROTATE_DIR/configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_
↪le135.py \
    --model work-dirs/mmrotate/rotated_retinanet/ort/end2end.onnx \
    --format-only \
    --metric-options submission_dir=work-dirs/mmrotate/rotated_retinanet/ort/Task1_
↪results
```

注意：

- mmrotate 模型需要额外输入，但我们无法直接获取它。在导出模型时，可以使用 `$MMROTATE_DIR/demo/demo.jpg` 作为输入。

CHAPTER 20

mmseg 模型支持列表

mmseg 是一个基于 PyTorch 的开源对象分割工具箱，也是 OpenMMLab 项目的一部分。

20.1 安装 mmseg

参照 get_started.md。

20.2 支持列表

20.3 注意事项

- 所有 mmseg 模型仅支持”whole” 推理模式。
- PSPNet，Fast-SCNN 仅支持静态输入，因为多数推理框架的 nn.AdaptiveAvgPool2d 不支持动态输入。
- 对于仅支持静态形状的模型，应使用静态形状的部署配置文件，例如 `configs/mmseg/segmentation_tensorrt_static-1024x2048.py`

CHAPTER 21

ncnn 支持情况

目前对 ncnn 特性使用情况如下：

以下特性还不能由 mmdeploy 自动开启，需要手动修改 ncnn 编译参数、或在 SDK 中调整运行参数

- bf16 inference
- nc4hw4 layout
- profiling per layer
- 关闭 NCNN_STRING 以减小 so 体积
- 设置线程数和 CPU 亲和力

CHAPTER 22

onnxruntime 支持情况

22.1 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.

22.2 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
```

22.3 Build custom ops

22.3.1 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
```

22.3.2 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)
```

22.4 How to convert a model

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

22.5 How to add a new custom op

22.6 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.

22.6.1 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:

22.7 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

CHAPTER 23

OpenVINO 支持情况

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

23.1 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 openvino-dev
```

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.

23.2 Usage

Example:

```
python tools/deploy.py \
    configs/mmdet/detection/detection_openvino_static-300x300.py \
    /mmdetection_dir/mmdetection/configs/ssd/ssd300_coco.py \
```

(下页继续)

(续上页)

```
    /tmp/snapshots/ssd300_coco_20210803_015428-d231a06e.pth \
    tests/data/tiger.jpeg \
    --work-dir ../deploy_result \
    --device cpu \
    --log-level INFO
```

23.3 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.

23.4 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.

23.5 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
```

CHAPTER 24

PPLNN 支持情况

MMDeploy supports ppl.nn v0.8.1 and later. This tutorial is based on Linux systems like Ubuntu-18.04.

24.1 Installation

1. Please install pyppl following install-guide.
2. Install MMDeploy following the *instructions*.

24.2 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
```

CHAPTER 25

SNPE 支持情况

目前 mmdeploy 集成了 onnx2dlc 模型转换的 SDK 推理，但以下特性还不支持：

- GPU_FP16 模式
- DSP/AIP 量化
- 算子内部 profiling
- UDO 算子

CHAPTER 26

TensorRT 支持情况

26.1 Installation

26.1.1 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
```

26.1.2 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)
```

26.2 Convert model

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

26.2.1 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']),
```

(下页继续)

(续上页)

```
            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.

26.3 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(
  ```

(下页继续)

(续上页)

```
            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.

CHAPTER 27

TorchScript 支持情况

27.1 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.

27.2 Build custom ops

27.2.1 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
```

(下页继续)

(续上页)

```
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
```

27.2.2 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)
```

27.3 How to convert a model

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

27.4 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.

CHAPTER 28

ncnn 自定义算子

28.1 Expand

28.1.1 Description

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

28.1.2 Parameters

Expand has no parameters.

28.1.3 Inputs

28.1.4 Outputs

28.1.5 Type Constraints

- ncnn.Mat: Mat(float32)

28.2 Gather

28.2.1 Description

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

28.2.2 Parameters

28.2.3 Inputs

28.2.4 Outputs

28.2.5 Type Constraints

- ncnn.Mat: Mat(float32)

28.3 Shape

28.3.1 Description

Get the shape of the ncnn blobs.

28.3.2 Parameters

Shape has no parameters.

28.3.3 Inputs

28.3.4 Outputs

28.3.5 Type Constraints

- ncnn.Mat: Mat(float32)

28.4 TopK

28.4.1 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`.

28.4.2 Parameters

28.4.3 Inputs

28.4.4 Outputs

28.4.5 Type Constraints

- ncnn.Mat: Mat(float32)

CHAPTER 29

onnxruntime 自定义算子

29.1 grid_sampler

29.1.1 Description

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

29.1.2 Parameters

29.1.3 Inputs

29.1.4 Outputs

29.1.5 Type Constraints

- T:tensor(float32, Linear)

29.2 MMCVModulatedDeformConv2d

29.2.1 Description

Perform Modulated Deformable Convolution on input feature, read Deformable ConvNets v2: More Deformable, Better Results for detail.

29.2.2 Parameters

29.2.3 Inputs

29.2.4 Outputs

29.2.5 Type Constraints

- T:tensor(float32, Linear)

29.3 NMSRotated

29.3.1 Description

Non Max Suppression for rotated bboxes.

29.3.2 Parameters

29.3.3 Inputs

29.3.4 Outputs

29.3.5 Type Constraints

- T:tensor(float32, Linear)

29.4 RoIAlignRotated

29.4.1 Description

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

29.4.2 Parameters

29.4.3 Inputs

29.4.4 Outputs

29.4.5 Type Constraints

- T:tensor(float32)

CHAPTER 30

TRT 自定义算子

30.1 TRTBatchedNMS

30.1.1 Description

Batched NMS with a fixed number of output bounding boxes.

30.1.2 Parameters

30.1.3 Inputs

30.1.4 Outputs

30.1.5 Type Constraints

- T:tensor(float32, Linear)

30.2 grid_sampler

30.2.1 Description

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

30.2.2 Parameters

30.2.3 Inputs

30.2.4 Outputs

30.2.5 Type Constraints

- T:tensor(float32, Linear)

30.3 MMCVInstanceNormalization

30.3.1 Description

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

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

30.3.2 Parameters

30.3.3 Inputs

30.3.4 Outputs

30.3.5 Type Constraints

- T:tensor(float32, Linear)

30.4 MMCVModulatedDeformConv2d

30.4.1 Description

Perform Modulated Deformable Convolution on input feature. Read Deformable ConvNets v2: More Deformable, Better Results for detail.

30.4.2 Parameters

30.4.3 Inputs

30.4.4 Outputs

30.4.5 Type Constraints

- T:tensor(float32, Linear)

30.5 MMCVMultiLevelRoiAlign

30.5.1 Description

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

30.5.2 Parameters

30.5.3 Inputs

30.5.4 Outputs

30.5.5 Type Constraints

- T:tensor(float32, Linear)

30.6 MMCVRoIAlign

30.6.1 Description

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

30.6.2 Parameters

30.6.3 Inputs

30.6.4 Outputs

30.6.5 Type Constraints

- T:tensor(float32, Linear)

30.7 ScatterND

30.7.1 Description

ScatterND takes three inputs `data` tensor of rank r >= 1, `indices` tensor of rank q >= 1, and `updates` tensor of rank q + r - 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]
```

30.7.2 Parameters

None

30.7.3 Inputs

30.7.4 Outputs

30.7.5 Type Constraints

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

30.8 TRTBatchedRotatedNMS

30.8.1 Description

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

30.8.2 Parameters

30.8.3 Inputs

30.8.4 Outputs

30.8.5 Type Constraints

- T:tensor(float32, Linear)

30.9 GridPriorsTRT

30.9.1 Description

Generate the anchors for object detection task.

30.9.2 Parameters

30.9.3 Inputs

30.9.4 Outputs

30.9.5 Type Constraints

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

CHAPTER 31

如何支持新的模型

我们提供了多种工具来支持模型转换

31.1 函数的重写器

PyTorch 神经网络是用 python 编写的，可以简化算法的开发。但与此同时 Python 的流程控制和第三方库会使得网络导出为中间语言的过程变得困难。为此我们提供了一个 “MonKey path” 工具将不支持的功能重写为另一个可支持中间语言导出的功能。下述是一个具体的使用例子：

```
from mmdeploy.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)
```

使用函数重写器是十分容易的，只需添加一个带参数的装饰器即可：

- `func_name` 是需要被重载的函数，它可以是其他 PyTorch 的函数或者是自定义的函数。模块中的方法也可以通过工具进行重载。
- `backend` 是推理引擎。当模型被导入到引擎的时候，函数会被重载。如果没有给出，重载默认的参数就是重载的参数。如果后端的重载的参数不存在，将会按照预设的默认模式进行重载。当参数与原始的

参数相同时，除了把上下文信息 `ctx` 作为第一的参数外，上下文也提供了一些有用的信息，例如: 部署的配置 `ctx.cfg` 和原始的函数（已经被重载）`ctx.origin_func`。

可参照这些样例代码。

31.2 模型重载器

如果您想用另一个模块替换整个模块，我们还有另一个重载器，如下所示:

```
@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)
```

就像函数重载器一样，可添加一个带参数的装饰器:

- `module_type` 要重载的模块类。
- `backend` 是推理引擎。当模型被导入到引擎的时候，函数会被重载。如果没有给出，重载默认的参数就是重载的参数。如果后端的重载的参数不存在，将会按照预设的默认模式进行重载。

网络中模块的所有实例都将替换为这个新类的实例。原始模块和部署配置将作为前两个参数进行传递。

31.3 符号函数重写

PyTorch 和 ONNX 之间的映射是通过 PyTorch 中的符号函数进行定义的。自定义符号函数可以帮助我们绕过一些推理引擎不支持的 ONNX 节点。

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

装饰器的参数

- `func_name` 要添加符号的函数名称。如果是自定义的，请使用完整路径 `torch.autograd.Function`。或者如果它是 PyTorch 内置函数，则只用写一个名称即可。
- `backend` 是推理引擎。当模型被导入到引擎的时候，函数会被重载。如果没有给出，重载默认的参数就是重载的参数。如果后端的重载的参数不存在，将会按照预设的默认模式进行重载。
- 如果函数是 PyTorch 内置函数，则为 True。
- `arg_descriptors` 符号函数参数的描述符，将被传递给 `torch.onnx.symbolic_helper._parse_arg`。

就像函数重载器的 `ctx` 一样，第一个参数会提供上下文信息。上下文中了一些有用的信息，例如部署配置 ctx.cfg 和原始功能（已被重载）`ctx.origin_func`。请注意，`ctx.origin_func` 只能在 `is_pytorch==False` 时使用。

这里有很多实现可参考。

CHAPTER 32

如何支持新的后端

MMDeploy 支持了许多后端推理引擎，但我们依然非常欢迎新后端的贡献。在本教程中，我们将介绍在 MMDeploy 中支持新后端的一般过程。

32.1 必要条件

在对 MMDeploy 添加新的后端引擎之前，需要先检查所要支持的新后端是否符合一些要求:

- 后端必须能够支持 ONNX 作为 IR。
- 如果后端需要“.onnx”文件以外的模型文件或权重文件，则需要添加将“.onnx”文件转换为模型文件或权重文件的转换工具，该工具可以是 Python API、脚本或可执行程序。
- 强烈建议新后端可提供 Python 接口来加载后端文件和推理以进行验证。

32.2 支持后端转换

MMDeploy 中的后端必须支持 ONNX，因此后端能直接加载“.onnx”文件，或者使用转换工具将“.onnx”转换成自己的格式。在本节中，我们将介绍支持后端转换的步骤。

1. 在 `mmdeploy/utils/constants.py` 文件中添加新推理后端变量，以表示支持的后端名称。

   示例:

```
# mmdeploy/utils/constants.py

class Backend(AdvancedEnum):
    # 以现有的 TensorRT 为例
    TENSORRT = 'tensorrt'
```

2. 在 mmdeploy/backend/ 目录下添加相应的库 (一个包括 __init__.py 的文件夹)，例如，mmdeploy/backend/tensorrt 。在 __init__.py 中，必须有一个名为 is_available 的函数检查用户是否安装了后端库。如果检查通过，则将加载库的剩余文件。

   例子:

```
# mmdeploy/backend/tensorrt/__init__.py

def is_available():
    return importlib.util.find_spec('tensorrt') is not None


if is_available():
    from .utils import from_onnx, load, save
    from .wrapper import TRTWrapper

    __all__ = [
        'from_onnx', 'save', 'load', 'TRTWrapper'
    ]
```

3. 在 configs/_base_/backends 目录中创建一个配置文件 (例如，configs/_base_/backends/tensorrt.py)。如果新后端引擎只是将 ".onnx" 文件作为输入，那么新的配置可以很简单, 对应配置只需包含一个表示后端名称的字段 (但也应该与 mmdeploy/utils/constants.py 中的名称相同)。

   例子

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

   但如果后端需要其他文件，则从 ".onnx" 文件转换为后端文件所需的参数也应包含在配置文件中。

   例子

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

   在拥有一个基本的后端配置文件后，您已经可以通过继承轻松构建一个完整的部署配置。有关详细信息，请参阅我们的配置教程。下面是一个例子:

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

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

4. 如果新后端需要模型文件或权重文件而不是“.onnx”文件，则需要在相应的文件夹中创建一个 onnx2backend.py 文件 (例如, 创建 mmdeploy/backend/tensorrt/onnx2tensorrt.py)。然后在文件中添加一个转换函数 onnx2backend。该函数应将给定的“.onnx”文件转换为给定工作目录中所需的后端文件。对函数的其他参数和实现细节没有要求，您可以使用任何工具进行转换。下面有些例子:

   使用 python 脚本

```
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)
```

   使用可执行文件

```
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])\
```

5. 在 mmdeploy/apis 中创建新后端库并声明对应 APIs

   例子

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

from mmdeploy.backend.ncnn import is_available

__all__ = ['is_available']
```

(下页继续)

(续上页)

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

然后根据需要使用这些 APIs 为 `tools/deploy.py` 添加相关转换代码

例子

```
# tools/deploy.py
# ...
    elif backend == Backend.NCNN:
        from mmdeploy.apis.ncnn import is_available as is_available_ncnn

        if not is_available_ncnn():
            logging.error('ncnn support is not available.')
            exit(-1)

        from mmdeploy.apis.ncnn import onnx2ncnn, get_output_model_file

        backend_files = []
        for onnx_path in onnx_files:
            create_process(
                f'mmdeploy_onnx2ncnn with {onnx_path}',
                target=onnx2ncnn,
                args=(onnx_path, args.work_dir),
                kwargs=dict(),
                ret_value=ret_value)
            backend_files += get_output_model_file(onnx_path, args.work_dir)
# ...
```

6. 将 OpenMMLab 的模型转换后 (如有必要) 并在后端引擎上进行推理。如果在测试时发现一些不兼容的算子，可以尝试按照重写器教程为后端重写原始模型或添加自定义算子。
7. 为新后端引擎代码添加相关注释和单元测试:).

32.3 支持后端推理

尽管后端引擎通常用 C/C++ 实现，但如果后端提供 Python 推理接口，则测试和调试非常方便。我们鼓励贡献者在 MMDeploy 的 Python 接口中支持新后端推理。在本节中，我们将介绍支持后端推理的步骤。

1. 添加一个名为 `wrapper.py` 的文件到 `mmdeploy/backend/{backend}` 中相应后端文件夹。例如，`mmdeploy/backend/tensorrt/wrapper` 。此模块应实现并注册一个封装类，该类继承 `mmdeploy/backend/base/base_wrapper.py` 中的基类 `BaseWrapper` 。

   例子

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

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

2. 封装类可以在函数 `__init__` 中初始化引擎以及在 `forward` 函数中进行推理。请注意，该 `__init__` 函数必须接受一个参数 `output_names` 并将其传递给基类以确定输出张量的顺序。其中 `forward` 输入和输出变量应表示 tensors 的名称和值的字典。

3. 为了方便性能测试，该类应该定义一个 `execute` 函数，只调用后端引擎的推理接口。该 `forward` 函数应在预处理数据后调用 `execute` 函数。

   例子

   ```
   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
           # ...
   ```

(下页继续)

(续上页)

```
        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. 为新封装装器添加默认初始化方法 mmdeploy/codebase/base/backend_model.py

   例子

```
    @staticmethod
    def _build_wrapper(backend: Backend,
                       backend_files: Sequence[str],
                       device: str,
                       output_names: Optional[Sequence[str]] = None):
        if backend == Backend.ONNXRUNTIME:
            from mmdeploy.backend.onnxruntime import ORTWrapper
            return ORTWrapper(
                onnx_file=backend_files[0],
                device=device,
                output_names=output_names)
```

5. 为新后端引擎代码添加相关注释和单元测试:).

32.4 将 MMDeploy 作为第三方库时添加新后端

前面的部分展示了如何在 MMDeploy 中添加新的后端，这需要更改其源代码。但是，如果我们将 MMDeploy 视为第三方，则上述方法不再有效。为此，添加一个新的后端需要我们预先安装另一个名为 aenum 的包。我们可以直接通过 `pip install aenum` 进行安装。

成功安装 aenum 后，我们可以通过以下方式使用它来添加新的后端：

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

try:
    Backend.get('backend_name')
```

(下页继续)

(续上页)

```
except Exception:
    extend_enum(Backend, 'BACKEND', 'backend_name')
```

我们可以在使用 MMDeploy 的重写逻辑之前运行上面的代码，这就完成了新后端的添加。

CHAPTER 33

为推理 ops 添加测试单元

本教程介绍如何为后端 ops 添加单元测试。在 backend_ops 目录下添加自定义 op 时，需要添加相应的测试单元。op 的单元测试在 `test/test_ops/test_ops.py` 中。

添加新的自定义 op 后，需要重新编译，引用*build.md* 。

33.1 ops 单元测试样例

```
@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()
```

(下页继续)

(续上页)

```
    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:
        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)
```

mmdeploy 支持的模型有两种格式：

- torch 模型：参考 roi_align 单元测试，必须要求 op 相关 Python 代码
- onnx 模型：参考 multi_level_roi_align 单元测试，需要调用 onnx api 进行构建

调用 `run_and_validate` 即可运行

```
    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):
```

33.1.1 Parameter Description

33.2 测试模型

用 pytest 调用 ops 测试

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

CHAPTER 34

测试模型重写

模型*rewriter* 完成后，还需完成对应测试用例，以验证重写是否生效。通常我们需要对比原始模型和重写后的输出。原始模型输出可以调用模型的 forward 函数直接获取，而生成重写模型输出的方法取决于重写的复杂性。

34.1 测试简单的重写

如果对模型的更改很小（例如，仅更改一个或两个变量且无副作用），则可为重写函数/模块构造输入，在 `RewriteContext` 中运行推理并检查结果。

```
# mmcls.models.classfiers.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, {})
```

在示例中，我们仅更改 forward 函数。我们可以通过编写以下函数来测试这个重写：

```
def test_baseclassfier_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):
            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'
```

在这个测试函数中，我们构造派生类 `BaseClassifier` 来测试重写能否工作。通过直接调用 `model(input)` 来获得原始输出，并通过在 `RewriteContext` 中调用 `model(input)` 来获取重写的输出。最后断检查输出。

34.2 测试复杂重写

有时我们可能会对原始模型函数进行重大更改（例如，消除分支语句以生成正确的计算图）。即使运行在 Python 中的重写模型的输出是正确的，我们也不能保证重写的模型可以在后端按预期工作。因此，我们需要在后端测试重写的模型。

```
# 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:
```

(下页继续)

(续上页)

```
        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)
```

此重写函数的行为很复杂，我们应该按如下方式测试它：

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

(下页继续)

(续上页)

```
        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
```

我们已经提供了一些使用函数做测试，例如可以先 build 模型，用 `get_model_outputs` 获取原始输出；然后用 `WrapModel` 包装重写函数，使用 `get_rewrite_outputs` 获取结果。这个例子里会返回输出内容和是否来自后端两个结果。

因为我们也不确定用户是否正确安装后端，所以得检查结果来自 Python 还是真实后端推理结果。单元测试必须涵盖这两种结果，最后用 `torch.allclose` 对比两种结果的差异。

API 文档中有测试用例完整用法。

CHAPTER 35

如何拆分 onnx 模型

MMDeploy 支持将 PyTorch 模型导出到 onnx 模型并进行拆分得到多个 onnx 模型文件，用户可以自由的对模型图节点进行标记并根据这些标记的节点定制任意的 onnx 模型拆分策略。在这个教程中，我们将通过具体例子来展示如何进行 onnx 模型拆分。在这个例子中，我们的目标是将 YOLOV3 模型拆分成两个部分，保留不带后处理的 onnx 模型，丢弃包含 Anchor 生成，NMS 的后处理部分。

35.1 步骤 1: 添加模型标记点

为了进行图拆分，我们定义了 `Mark` 类型 op，标记模型导出的边界。在实现方法上，采用 `mark` 装饰器对函数的输入、输出 `Tensor` 打标记。需要注意的是，我们的标记函数需要在某个重写函数中执行才能生效。

为了对 YOLOV3 进行拆分，首先我们需要标记模型的输入。这里为了通用性，我们标记检测器父类 `BaseDetector` 的 `forward` 方法中的 `img` `Tensor`，同时为了支持其他拆分方案，也对 `forward` 函数的输出进行了标记，分别是 `dets`，`labels` 和 `masks`。下面的代码是截图mmdeploy/codebase/mmdet/models/detectors/base.py中的一部分，可以看出我们使用 `mark` 装饰器标记了 `__forward_impl` 函数的输入输出，并在重写函数 `base_detector__forward` 进行了调用，从而完成了对检测器输入的标记。

```
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):
    ...
```

(下页继续)

(续上页)

```
@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(...)
```

接下来，我们只需要对 YOLOV3Head 中最后一层输出特征 Tensor 进行标记就可以将整个 YOLOV3 模型拆分成两部分。通过查看 mmdet 源码我们可以知道 YOLOV3Head 的 get_bboxes 方法中输入参数 pred_maps 就是我们想要的拆分点，因此可以在重写函数yolov3_head__get_bboxes中添加内部函数对 pred_mapes 进行标记，具体参考如下示例代码。值得注意的是，输入参数 pred_maps 是由三个 Tensor 组成的列表，所以我们在 onnx 模型中添加了三个 Mark 标记节点。

```
from mmdeploy.core import FUNCTION_REWRITER, mark

@FUNCTION_REWRITER.register_rewriter(
    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)
    ...
```

35.2 步骤 2: 添加部署配置文件

在完成模型中节点标记之后，我们需要创建部署配置文件，我们假设部署后端是 onnxruntime，并模型输入是固定尺寸 608x608，因此添加文件 configs/mmdet/detection/yolov3_partition_onnxruntime_static.py. 我们需要在配置文件中添加基本的配置信息如 onnx_config，如何你还不熟悉如何添加配置文件，可以参考*write_config.md*.

在这个部署配置文件中，我们需要添加一个特殊的模型分段配置字段 partition_config. 在模型分段配置中，我们可以可以给分段策略添加一个类型名称如 yolov3_partition，设定 apply_marks=True。

在分段方式 partition_cfg, 我们需要指定每段模型的分割起始点 start, 终止点 end 以及保存分段 onnx 的文件名。需要提醒的是，各段模型起始点 start 和终止点 end 是由多个标记节点 Mark 组成，例如'detector_forward:input' 代表 detector_forward 标记处输入所产生的标记节点。配置文件具体内容参考如下代码：

```
_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, ...]
    ])
```

35.3 步骤 3: 拆分 onnx 模型

添加好节点标记和部署配置文件，我们可以使用 tools/torch2onnx.py 工具导出带有 Mark 标记的完成 onnx 模型并根据分段策略提取分段的 onnx 模型文件。我们可以执行如下脚本，得到不带后处理的 YOLOV3onnx 模型文件 yolov3.onnx，同时输出文件中也包含了添加 Mark 标记的完整模型文件 end2end.onnx。此外，用户可以使用网页版模型可视化工具netron来查看和验证输出 onnx 模型的结构是否正确。

```
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
```

当得到分段 onnx 模型之后，我们可以使用 mmdeploy 提供的其他工具如 mmdeploy_onnx2ncnn, onnx2tensorrt 来进行后续的模型部署工作。

CHAPTER 36

如何进行回归测试

这篇教程介绍了如何进行回归测试。部署配置文件由每个 codebase 的回归配置文件，推理框架配置信息组成。

36.1 1. 环境搭建

36.1.1 MMDeploy 的安装及配置

本章节的内容，需要提前根据*build 文档*将 MMDeploy 安装配置好之后，才能进行。

36.1.2 Python 环境依赖

需要安装 test 的环境

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

如果在使用过程是 numpy 报错，则更新一下 numpy

```
pip install -U numpy
```

36.2 2. 用法

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

36.2.1 参数解析

- `--codebase`：需要测试的 codebase，eg.`mmdet`, 测试多个 `mmcls mmdet ...`
- `--backends`：筛选测试的后端, 默认测全部 `backend`, 也可传入若干个后端，例如 `onnxruntime tesnsorrt`。如果需要一同进行 SDK 的测试，需要在 `tests/regression/${codebase}.yml` 里面的 `sdk_config` 进行配置。

- `--models`：指定测试的模型，默认测试 `yml` 中所有模型，也可传入若干个模型名称，模型名称可参考相关 yml 配置文件。例如 `ResNet SE-ResNet "Mask R-CNN"`。注意的是，可传入只有字母和数字组成模型名称，例如 `resnet seresnet maskrcnn`。
- `--work-dir`：模型转换、报告生成的路径，默认是`../mmdeploy_regression_working_dir`，注意路径中不要含空格等特殊字符。
- `--checkpoint-dir`: PyTorch 模型文件下载保存路径，默认是`../mmdeploy_checkpoints`，注意路径中不要含空格等特殊字符。
- `--device`：使用的设备，默认 `cuda`。
- `--log-level`：设置日记的等级，选项包括`'CRITICAL','FATAL','ERROR','WARN','WARNING', 'INFO', 'DEBUG', 'NOTSET'`。默认是 `INFO`。
- `-p` 或 `--performance`：是否测试精度，加上则测试转换 + 精度，不加上则只测试转换

36.2.2 注意事项

对于 Windows 用户：

1. 要在 shell 命令中使用 `&&` 连接符，需要下载并使用 `PowerShell 7 Preview 5+`。
2. 如果您使用 conda env，可能需要在 regression_test.py 中将 `python3` 更改为 `python`，因为 `%USERPROFILE%\AppData\Local\Microsoft\WindowsApps` 目录中有 `python3.exe`。

36.3 例子

1. 测试 mmdet 和 mmpose 的所有 backend 的 **转换 + 精度**

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

2. 测试 mmdet 和 mmpose 的某几个 backend 的 **转换 + 精度**

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

3. 测试 mmdet 和 mmpose 的某几个 backend，**只测试转换**

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

4. 测试 mmdet 和 mmcls 的某几个 models，**只测试转换**

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

36.4 3. 回归测试配置文件

36.4.1 示例及参数解析

```
globals:
  codebase_dir: ../mmocr # 回归测试的 codebase 路径
  checkpoint_force_download: False # 回归测试是否重新下载模型即使其已经存在
  images: # 测试使用图片
    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 # 指标参数
    hmean-iou: # 命名根据 metafile.Results.Metrics
      eval_name: hmean-iou # 命名根据 test.py --metrics args 入参名称
      metric_key: 0_hmean-iou:hmean # 命名根据 eval 写入 log 的 key name
      tolerance: 0.1 # 容忍的阈值区间
      task_name: Text Detection # 命名根据模型 metafile.Results.Task
      dataset: ICDAR2015 # 命名根据模型 metafile.Results.Dataset
    word_acc: # 同上
      eval_name: acc
      metric_key: 0_word_acc_ignore_case
      tolerance: 0.2
      task_name: Text Recognition
```

(下页继续)

(续上页)

```
      dataset: IIIT5K
  convert_image_det: &convert_image_det # det 转换会使用到的图片
    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 # 是否对 backend 进行精度测试
  sdk: # SDK 配置文件
    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 # 转换过程中使用的图片
    backend_test: *default_backend_test # 是否进行后端测试，存在则判断，不存在则视为 False
    sdk_config: *sdk_recognition_dynamic # 是否进行 SDK 测试，存在则使用特定的 SDK config 进
行测试，不存在则视为不进行 SDK 测试
    deploy_config: configs/mmocr/text-recognition/text-recognition_onnxruntime_static.
↪py # 使用的 deploy cfg 路径，基于 mmdeploy 的路径

  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/mmocr/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/mmocr/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/mmocr/text-recognition/text-recognition_tensorrt-fp16_
↪dynamic-1x32x32-1x32x640.py

  pipeline_trt_detection_dynamic_fp16: &pipeline_trt_detection_dynamic_fp16
```

(下页继续)

(续上页)

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

openvino:
  # 此处省略，内容同上
ncnn:
  # 此处省略，内容同上
pplnn:
  # 此处省略，内容同上
torchscript:
  # 此处省略，内容同上


models:
  - name: crnn # 模型名称
    metafile: configs/textrecog/crnn/metafile.yml # 模型对应的 metafile 的路径，相对于␣
↪codebase 的路径
    codebase_model_config_dir: configs/textrecog/crnn # `model_configs` 的父文件夹路径，相
对于 codebase 的路径
    model_configs: # 需要测试的 config 名称
      - crnn_academic_dataset.py
    pipelines: # 使用的 pipeline
      - *pipeline_ort_recognition_dynamic_fp32

  - name: dbnet
    metafile: configs/textdet/dbnet/metafile.yml
    codebase_model_config_dir: configs/textdet/dbnet
    model_configs:
      - dbnet_r18_fpnc_1200e_icdar2015.py
    pipelines:
      - *pipeline_ort_detection_dynamic_fp32
      - *pipeline_trt_detection_dynamic_fp16

      # 特殊的 pipeline 可以这样加入
      - convert_image: xxx
        backend_test: xxx
        sdk_config: xxx
        deploy_config: configs/mmocr/text-detection/xxx
```

36.5 4. 生成的报告

36.5.1 模板

36.5.2 示例

这是 MMOCR 生成的报告

36.6 5. 支持的后端

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

36.7 6. 支持的 Codebase 及其 Metric

36.8 7. 注意事项

暂无

36.9 8. 常见问题

暂无

CHAPTER 37

ONNX export Optimizer

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

37.1 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 .
```

37.2 Usage

```
# import model_to_graph_custom_optimizer so we can hijack onnx.export
from mmdeploy.apis.onnx.optimizer import model_to_graph__custom_optimizer # noqa
from mmdeploy.core import RewriterContext
from mmdeploy.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):
    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
```

CHAPTER 38

第一章：模型部署简介

OpenMMLab 的算法如何部署？这是很多社区用户的困惑。而模型部署工具箱 MMDeploy 的开源，强势打通了从算法模型到应用程序这" 最后一公里"！今天我们将开启模型部署入门系列教程，在模型部署开源库 MMDeploy 的辅助下，介绍以下内容：

- 中间表示 ONNX 的定义标准。
- PyTorch 模型转换到 ONNX 模型的方法。
- 推理引擎 ONNX Runtime、TensorRT 的使用方法。
- 部署流水线 PyTorch - ONNX - ONNX Runtime/TensorRT 的示例及常见部署问题的解决方法。
- MMDeploy C/C++ 推理 SDK。

希望通过本系列教程，带领大家学会如何把自己的 PyTorch 模型部署到 ONNX Runtime/TensorRT 上，并学会如何把 OpenMMLab 开源体系中各个计算机视觉任务的模型用 MMDeploy 部署到各个推理引擎上。

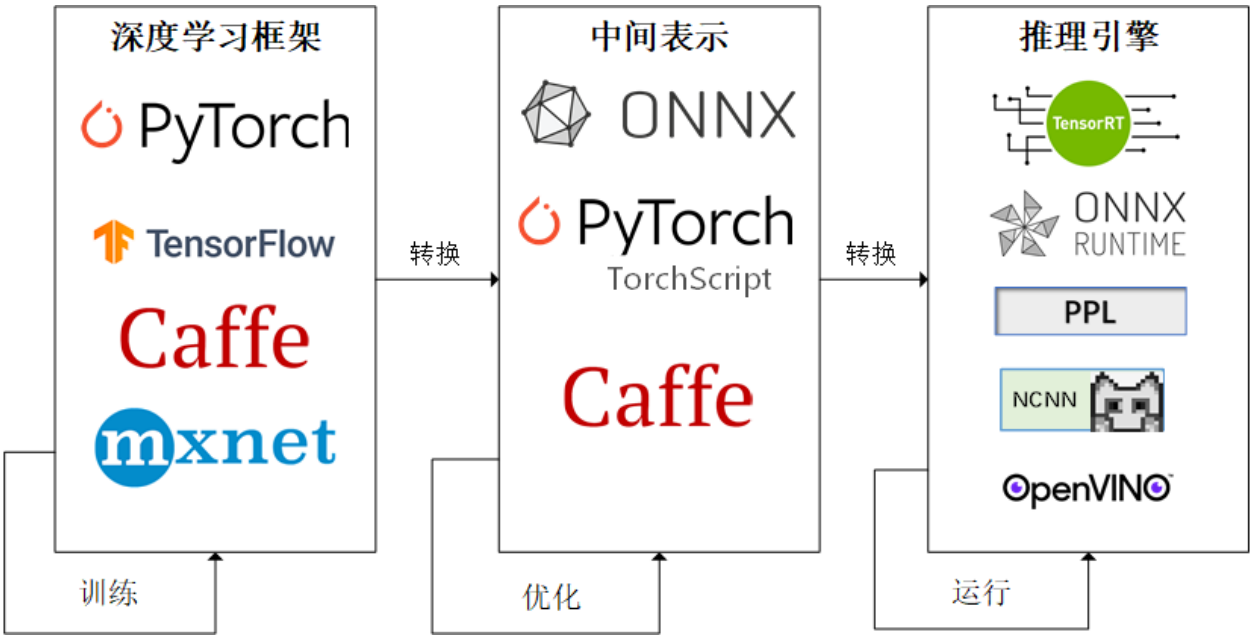
我们默认大家熟悉 Python 语言，并对 PyTorch 框架有基本的认识，除此之外不需要了解任何模型部署的知识。

在第一篇文章中，我们将部署一个简单的超分辨率模型，认识中间表示、推理引擎等模型部署中的概念。

38.1 初识模型部署

在软件工程中，部署指把开发完毕的软件投入使用的过程，包括环境配置、软件安装等步骤。类似地，对于深度学习模型来说，模型部署指让训练好的模型在特定环境中运行的过程。相比于软件部署，模型部署会面临更多的难题：

1. 运行模型所需的环境难以配置。深度学习模型通常是由一些框架编写，比如 PyTorch、TensorFlow。由于框架规模、依赖环境的限制，这些框架不适合在手机、开发板等生产环境中安装。
2. 深度学习模型的结构通常比较庞大，需要大量的算力才能满足实时运行的需求。模型的运行效率需要优化。因为这些难题的存在，模型部署不能靠简单的环境配置与安装完成。经过工业界和学术界数年的探索，模型部署有了一条流行的流水线：



为了让模型最终能够部署到某一环境上，开发者们可以使用任意一种深度学习框架来定义网络结构，并通过训练确定网络中的参数。之后，模型的结构和参数会被转换成一种只描述网络结构的中间表示，一些针对网络结构的优化会在中间表示上进行。最后，用面向硬件的高性能编程框架（如 CUDA，OpenCL）编写，能高效执行深度学习网络中算子的推理引擎会把中间表示转换成特定的文件格式，并在对应硬件平台上高效运行模型。

这一条流水线解决了模型部署中的两大问题：使用对接深度学习框架和推理引擎的中间表示，开发者不必担心如何在新环境中运行各个复杂的框架；通过中间表示的网络结构优化和推理引擎对运算的底层优化，模型的运算效率大幅提升。

现在，让我们从一个模型部署的“Hello World”项目入手，见识一下模型部署各方面的知识吧！

38.2 部署第一个模型

38.2.1 创建 PyTorch 模型

仿照 PyTorch 的官方部署教程，让我们用 PyTorch 实现一个超分辨率模型，并把模型部署到 ONNX Runtime 这个推理引擎上。首先，我们需要创建一个有 PyTorch 库的 Python 编程环境。如果你的 PyTorch 环境还没有装好，可以参考官方的入门教程。我们强烈推荐使用 conda 来管理 Python 库。使用 conda 可以靠如下的命令初始化一个 PyTorch 环境：

```
# 创建预安装 Python 3.7 的名叫 deploy 虚拟环境
conda create -n deploy python=3.7 -y
# 进入虚拟环境
conda activate deploy
# 安装 cpu 版本的 PyTorch
conda install pytorch torchvision cpuonly -c pytorch
```

如果你的设备支持 cuda 编程，我们建议你在配置 cuda 环境后使用 gpu 上的 PyTorch。比如将上面安装 PyTorch 的命令改成：

```
# 安装 cuda 11.3 的 PyTorch
# 如果你用的是其他版本的 cuda，请参考上面 PyTorch 的官方安装教程选择安装命令
conda install pytorch torchvision cudatoolkit=11.3 -c pytorch
```

本教程会用到其他一些第三方库。你可以用以下命令来安装这些库：

```
# 安装 ONNX Runtime, ONNX, OpenCV
pip install onnxruntime onnx opencv-python
```

在一切都配置完毕后，用下面的代码来创建一个超分辨率模型。

```
import os

import cv2
import numpy as np
import requests
import torch
import torch.onnx
from torch import nn

class SuperResolutionNet(nn.Module):
    def __init__(self, upscale_factor):
        super().__init__()
        self.upscale_factor = upscale_factor
        self.img_upsampler = nn.Upsample(
```

(下页继续)

(续上页)

```
            scale_factor=self.upscale_factor,
            mode='bicubic',
            align_corners=False)

        self.conv1 = nn.Conv2d(3,64,kernel_size=9,padding=4)
        self.conv2 = nn.Conv2d(64,32,kernel_size=1,padding=0)
        self.conv3 = nn.Conv2d(32,3,kernel_size=5,padding=2)

        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.img_upsampler(x)
        out = self.relu(self.conv1(x))
        out = self.relu(self.conv2(out))
        out = self.conv3(out)
        return out

# Download checkpoint and test image
urls = ['https://download.openmmlab.com/mmediting/restorers/srcnn/srcnn_x4k915_1x16_
↪1000k_div2k_20200608-4186f232.pth',
    'https://raw.githubusercontent.com/open-mmlab/mmediting/master/tests/data/face/
↪000001.png']
names = ['srcnn.pth', 'face.png']
for url, name in zip(urls, names):
    if not os.path.exists(name):
        open(name, 'wb').write(requests.get(url).content)

def init_torch_model():
    torch_model = SuperResolutionNet(upscale_factor=3)

    state_dict = torch.load('srcnn.pth')['state_dict']

    # Adapt the checkpoint
    for old_key in list(state_dict.keys()):
        new_key = '.'.join(old_key.split('.')[1:])
        state_dict[new_key] = state_dict.pop(old_key)

    torch_model.load_state_dict(state_dict)
    torch_model.eval()
    return torch_model

model = init_torch_model()
input_img = cv2.imread('face.png').astype(np.float32)
```

(下页继续)

(续上页)

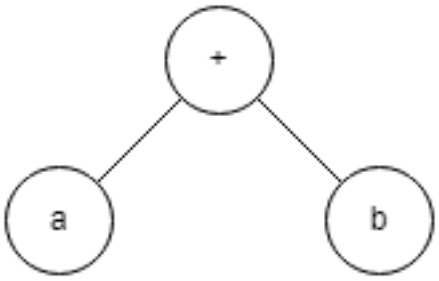
```
# HWC to NCHW
input_img = np.transpose(input_img, [2, 0, 1])
input_img = np.expand_dims(input_img, 0)

# Inference
torch_output = model(torch.from_numpy(input_img)).detach().numpy()

# NCHW to HWC
torch_output = np.squeeze(torch_output, 0)
torch_output = np.clip(torch_output, 0, 255)
torch_output = np.transpose(torch_output, [1, 2, 0]).astype(np.uint8)

# Show image
cv2.imwrite("face_torch.png", torch_output)
```

在这份代码中，我们创建了一个经典的超分辨率网络 SRCNN。SRCNN 先把图像上采样到对应分辨率，再用 3 个卷积层处理图像。为了方便起见，我们跳过训练网络的步骤，直接下载模型权重（由于 MMEditing 中 SRCNN 的权重结构和我们定义的模型不太一样，我们修改了权重字典的 key 来适配我们定义的模型），同时下载好输入图片。为了让模型输出成正确的图片格式，我们把模型的输出转换成 HWC 格式，并保证每一通道的颜色值都在 0~255 之间。如果脚本正常运行的话，一幅超分辨率的人脸照片会保存在 “face_torch.png” 中。

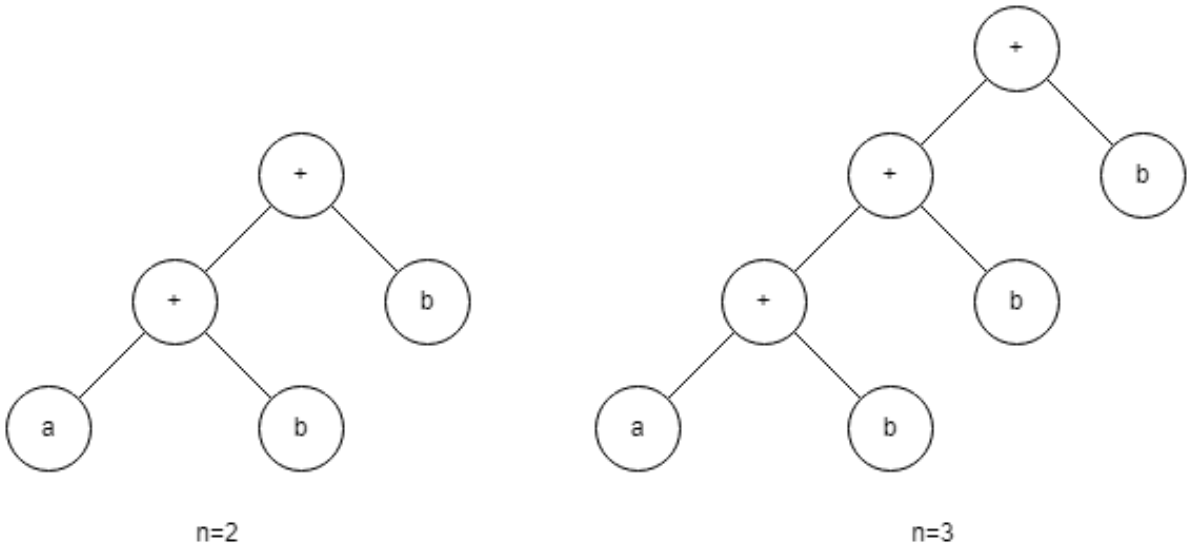
在 PyTorch 模型测试正确后，我们来正式开始部署这个模型。我们下一步的任务是把 PyTorch 模型转换成用中间表示 ONNX 描述的模型。

38.2.2 中间表示——ONNX

在介绍 ONNX 之前，我们先从本质上来认识一下神经网络的结构。神经网络实际上只是描述了数据计算的过程，其结构可以用计算图表示。比如 a+b 可以用下面的计算图来表示：



为了加速计算，一些框架会使用对神经网络“先编译，后执行”的静态图来描述网络。静态图的缺点是难以描述控制流（比如 if-else 分支语句和 for 循环语句），直接对其引入控制语句会导致产生不同的计算图。比如循环执行 n 次 a=a+b，对于不同的 n，会生成不同的计算图：



ONNX （Open Neural Network Exchange）是 Facebook 和微软在 2017 年共同发布的，用于标准描述计算图的一种格式。目前，在数家机构的共同维护下，ONNX 已经对接了多种深度学习框架和多种推理引擎。因此，ONNX 被当成了深度学习框架到推理引擎的桥梁，就像编译器的中间语言一样。由于各框架兼容性不一，我们通常只用 ONNX 表示更容易部署的静态图。

让我们用下面的代码来把 PyTorch 的模型转换成 ONNX 格式的模型：

```
x = torch.randn(1, 3, 256, 256)

with torch.no_grad():
```

(下页继续)

(续上页)

```
    torch.onnx.export(
        model,
        x,
        "srcnn.onnx",
        opset_version=11,
        input_names=['input'],
        output_names=['output'])
```

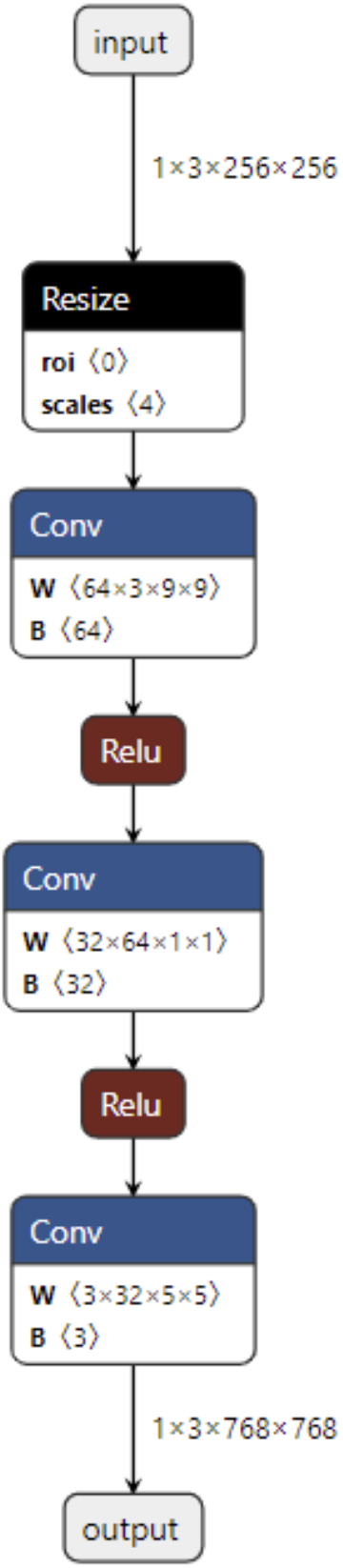
其中，**torch.onnx.export** 是 PyTorch 自带的把模型转换成 ONNX 格式的函数。让我们先看一下前三个必选参数：前三个参数分别是要转换的模型、模型的任意一组输入、导出的 ONNX 文件的文件名。转换模型时，需要原模型和输出文件名是很容易理解的，但为什么需要为模型提供一组输入呢？这就涉及到 ONNX 转换的原理了。从 PyTorch 的模型到 ONNX 的模型，本质上是一种语言上的翻译。直觉上的想法是像编译器一样彻底解析原模型的代码，记录所有控制流。但前面也讲到，我们通常只用 ONNX 记录不考虑控制流的静态图。因此，PyTorch 提供了一种叫做追踪（trace）的模型转换方法：给定一组输入，再实际执行一遍模型，即把这组输入对应的计算图记录下来，保存为 ONNX 格式。export 函数用的就是追踪导出方法，需要给任意一组输入，让模型跑起来。我们的测试图片是三通道，256x256 大小的，这里也构造一个同样形状的随机张量。

剩下的参数中，opset_version 表示 ONNX 算子集的版本。深度学习的发展会不断诞生新算子，为了支持这些新增的算子，ONNX 会经常发布新的算子集，目前已经更新 15 个版本。我们令 opset_version = 11，即使用第 11 个 ONNX 算子集，是因为 SRCNN 中的 bicubic （双三次插值）在 opset11 中才得到支持。剩下的两个参数 input_names, output_names 是输入、输出 tensor 的名称，我们稍后会用到这些名称。

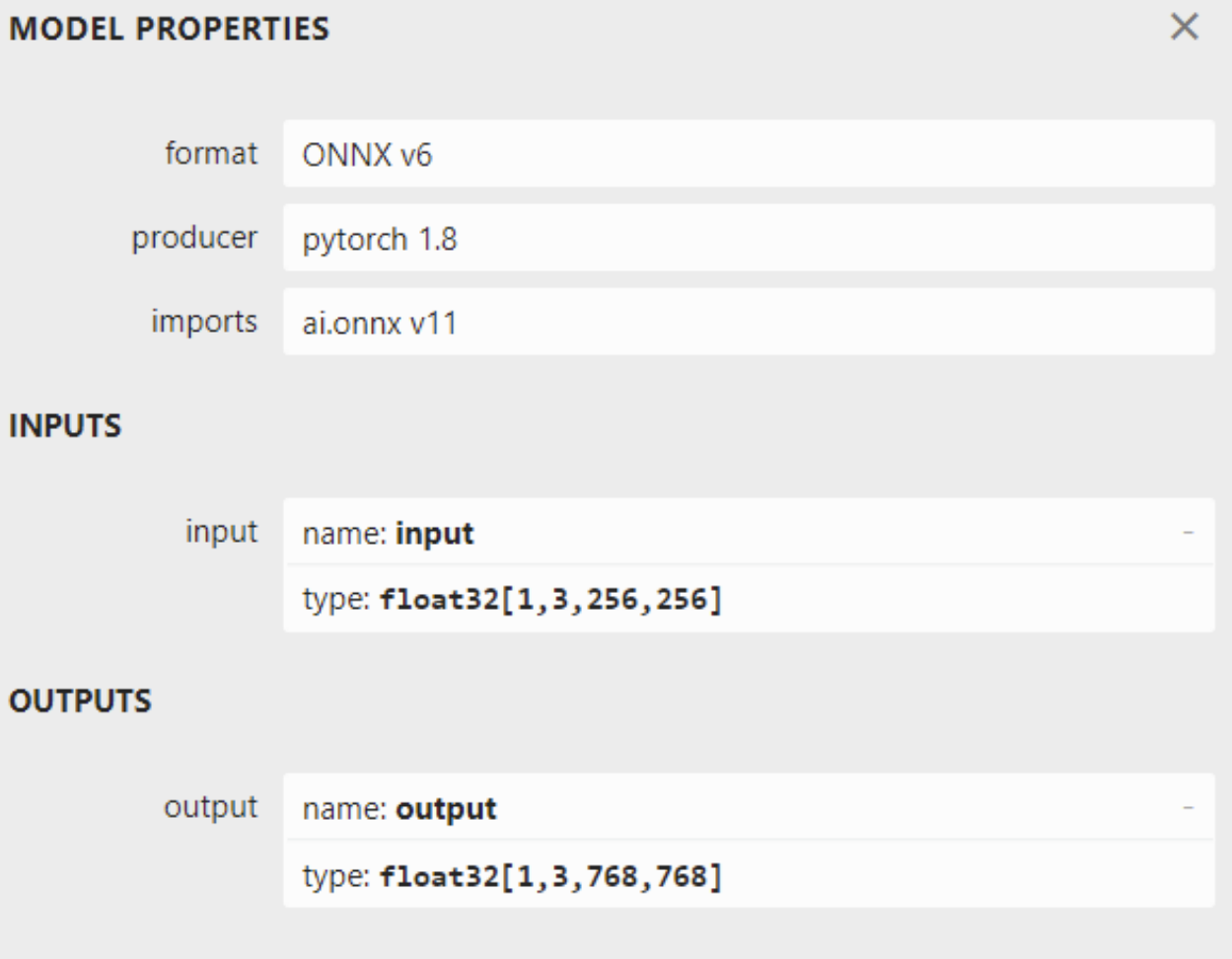
如果上述代码运行成功，目录下会新增一个”srcnn.onnx” 的 ONNX 模型文件。我们可以用下面的脚本来验证一下模型文件是否正确。

```
import onnx

onnx_model = onnx.load("srcnn.onnx")
try:
    onnx.checker.check_model(onnx_model)
except Exception:
    print("Model incorrect")
else:
    print("Model correct")
```

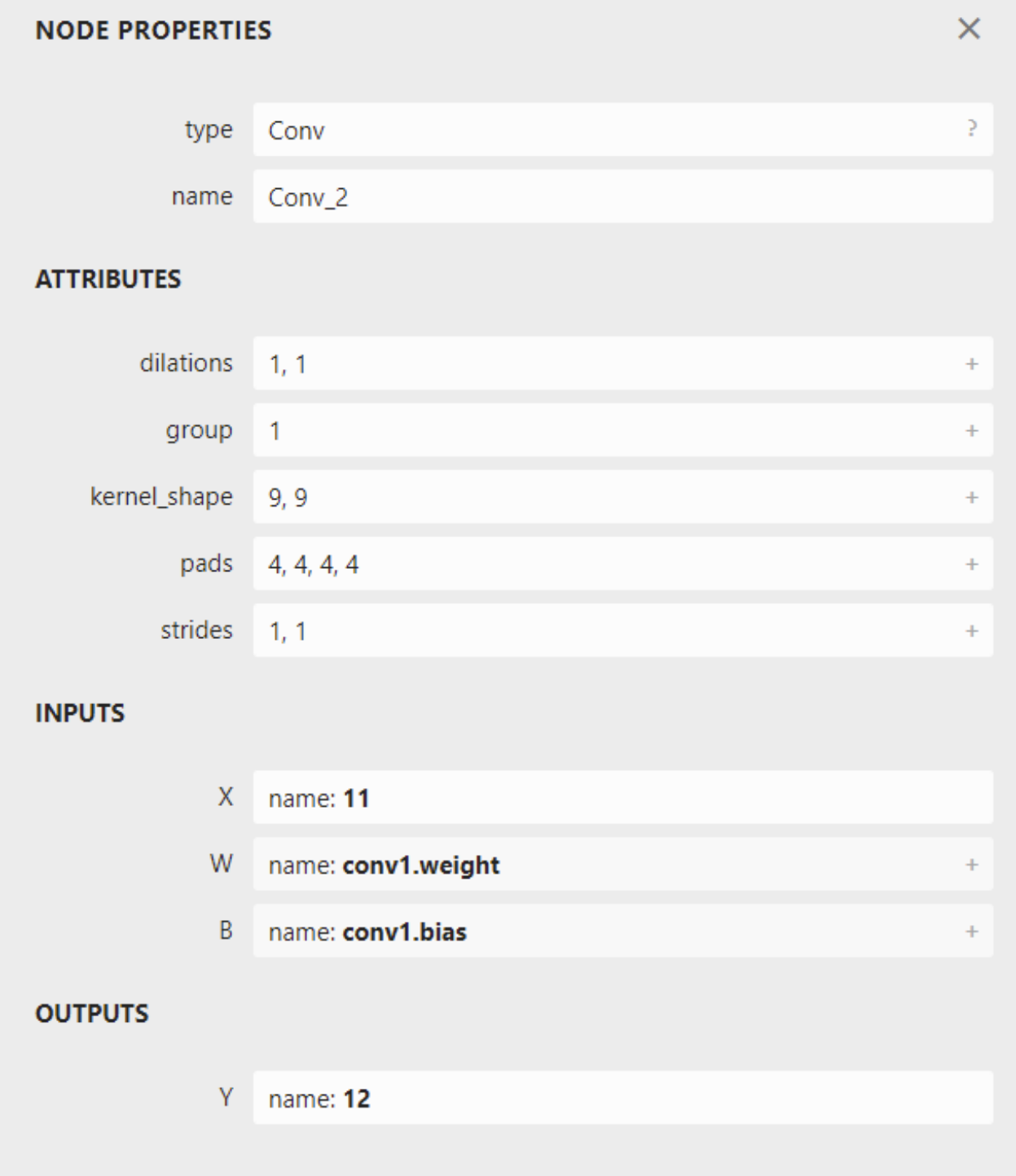
其中，onnx.load 函数用于读取一个 ONNX 模型。onnx.checker.check_model 用于检查模型格式是否正确，如果有错误的话该函数会直接报错。我们的模型是正确的，控制台中应该会打印出”Model correct”。接下来，让我们来看一看 ONNX 模型具体的结构是怎么样的。我们可以使用 Netron （开源的模型可视化工具）来可视化 ONNX 模型。把 srcnn.onnx 文件从本地的文件系统拖入网站，即可看到如下的可视化结果：


input
1×3×256×256
Resize
roi 〈0〉
scales 〈4〉
Conv
W 〈64×3×9×9〉
B 〈64〉
Relu
Conv
W 〈32×64×1×1〉
B 〈32〉
Relu
Conv
W 〈3×32×5×5〉
B 〈3〉
1×3×768×768
output

点击 input 或者 output，可以查看 ONNX 模型的基本信息，包括模型的版本信息，以及模型输入、输出的名称和数据类型。



点击某一个算子节点，可以看到算子的具体信息。比如点击第一个 Conv 可以看到:



每个算子记录了算子属性、图结构、权重三类信息。

- 算子属性信息即图中 attributes 里的信息，对于卷积来说，算子属性包括了卷积核大小 (kernel_shape)、卷积步长 (strides) 等内容。这些算子属性最终会用来生成一个具体的算子。
- 图结构信息指算子节点在计算图中的名称、邻边的信息。对于图中的卷积来说，该算子节点叫做 Conv_2，输入数据叫做 11，输出数据叫做 12。根据每个算子节点的图结构信息，就能完整地复原出网络的计算

图。

- 权重信息指的是网络经过训练后，算子存储的权重信息。对于卷积来说，权重信息包括卷积核的权重值和卷积后的偏差值。点击图中 conv1.weight, conv1.bias 后面的加号即可看到权重信息的具体内容。

现在，我们有了 SRCNN 的 ONNX 模型。让我们看看最后该如何把这个模型运行起来。

38.2.3 推理引擎——ONNX Runtime

ONNX Runtime 是由微软维护的一个跨平台机器学习推理加速器，也就是我们前面提到的”推理引擎“。ONNX Runtime 是直接对接 ONNX 的，即 ONNX Runtime 可以直接读取并运行.onnx 文件, 而不需要再把.onnx 格式的文件转换成其他格式的文件。也就是说，对于 PyTorch - ONNX - ONNX Runtime 这条部署流水线，只要在目标设备中得到.onnx 文件，并在 ONNX Runtime 上运行模型，模型部署就算大功告成了。

通过刚刚的操作，我们把 PyTorch 编写的模型转换成了 ONNX 模型，并通过可视化检查了模型的正确性。最后，让我们用 ONNX Runtime 运行一下模型，完成模型部署的最后一步。

ONNX Runtime 提供了 Python 接口。接着刚才的脚本，我们可以添加如下代码运行模型：

```
import onnxruntime

ort_session = onnxruntime.InferenceSession("srcnn.onnx")
ort_inputs = {'input': input_img}
ort_output = ort_session.run(['output'], ort_inputs)[0]

ort_output = np.squeeze(ort_output, 0)
ort_output = np.clip(ort_output, 0, 255)
ort_output = np.transpose(ort_output, [1, 2, 0]).astype(np.uint8)
cv2.imwrite("face_ort.png", ort_output)
```

这段代码中，除去后处理操作外，和 ONNX Runtime 相关的代码只有三行。让我们简单解析一下这三行代码。**onnxruntime.InferenceSession** 用于获取一个 ONNX Runtime 推理器，其参数是用于推理的 ONNX 模型文件。推理器的 run 方法用于模型推理，其第一个参数为输出张量名的列表，第二个参数为输入值的字典。其中输入值字典的 key 为张量名，value 为 numpy 类型的张量值。输入输出张量的名称需要和 **torch.onnx.export** 中设置的输入输出名对应。

如果代码正常运行的话，另一幅超分辨率照片会保存在”face_ort.png” 中。这幅图片和刚刚得到的”face_torch.png” 是一模一样的。这说明 ONNX Runtime 成功运行了 SRCNN 模型，模型部署完成了！以后有用户想实现超分辨率的操作，我们只需要提供一个”srcnn.onnx” 文件，并帮助用户配置好 ONNX Runtime 的 Python 环境，用几行代码就可以运行模型了。或者还有更简便的方法，我们可以利用 ONNX Runtime 编译出一个可以直接执行模型的应用程序。我们只需要给用户提供 ONNX 模型文件，并让用户在应用程序选择要执行的 ONNX 模型文件名就可以运行模型了。

38.3 总结

在这篇教程里，我们利用成熟的模型部署工具，轻松部署了一个初始版本的超分辨率模型 SRCNN。但在实际应用场景中，随着模型结构的复杂度不断加深，碰到的困难的也会越来越多。在下一篇教程里，我们将“升级”一下这个超分辨率模型，让它支持动态的输入。

看完这篇教程，是不是感觉知识太多一下消化不过来？没关系，模型部署本身有非常多的东西要学。为了举例的方便，这篇教程包含了许多未来才会讲到的知识点。事实上，读完这篇教程后，记下以下知识点就够了：

- 模型部署，指把训练好的模型在特定环境中运行的过程。模型部署要解决模型框架兼容性差和模型运行速度慢这两大问题。
- 模型部署的常见流水线是“深度学习框架-中间表示-推理引擎”。其中比较常用的一个中间表示是 ONNX。
- 深度学习模型实际上就是一个计算图。模型部署时通常把模型转换成静态的计算图，即没有控制流（分支语句、循环语句）的计算图。
- PyTorch 框架自带对 ONNX 的支持，只需要构造一组随机的输入，并对模型调用 torch.onnx.export 即可完成 PyTorch 到 ONNX 的转换。
- 推理引擎 ONNX Runtime 对 ONNX 模型有原生的支持。给定一个.onnx 文件，只需要简单使用 ONNX Runtime 的 Python API 就可以完成模型推理。

为了实现深度学习算法的落地，充满挑战的模型部署是一个逃不开的步骤。MMDeploy 实现了 OpenMMLab 中目标检测、图像分割、超分辨率等多个视觉任务模型的部署，支持 ONNX Runtime，TensorRT，ncnn，openppl，OpenVINO 等多个推理引擎。

在后续的模型部署教程中，我们将在介绍模型部署技术的同时，介绍这些技术是如何运用在 MMDeploy 中的。

CHAPTER 39

第二章：解决模型部署中的难题

在第一章中，我们部署了一个简单的超分辨率模型，一切都十分顺利。但是，上一个模型还有一些缺陷——图片的放大倍数固定是 4，我们无法让图片放大任意的倍数。现在，我们来尝试部署一个支持动态放大倍数的模型，体验一下在模型部署中可能会碰到的困难。

39.1 模型部署中常见的难题

在之前的学习中，我们在模型部署上顺风顺水，没有碰到任何问题。这是因为 SRCNN 模型只包含几个简单的算子，而这些卷积、插值算子已经在各个中间表示和推理引擎上得到了完美支持。如果模型的操作稍微复杂一点，我们可能就要为兼容模型而付出大量的功夫了。实际上，模型部署时一般会碰到以下几类困难：

- 模型的动态化。出于性能的考虑，各推理框架都默认模型的输入形状、输出形状、结构是静态的。而为了让模型的泛用性更强，部署时需要在尽可能不影响原有逻辑的前提下，让模型的输入输出或是结构动态化。
- 新算子的实现。深度学习技术日新月异，提出新算子的速度往往快于 ONNX 维护者支持的速度。为了部署最新的模型，部署工程师往往需要自己在 ONNX 和推理引擎中支持新算子。
- 中间表示与推理引擎的兼容问题。由于各推理引擎的实现不同，对 ONNX 难以形成统一的支持。为了确保模型在不同的推理引擎中有同样的运行效果，部署工程师往往得为某个推理引擎定制模型代码，这为模型部署引入了许多工作量。

我们会在后续教程详细讲述解决这些问题的方法。如果对前文中 ONNX、推理引擎、中间表示、算子等名词感觉陌生，不用担心，可以阅读第一章，了解有关概念。

现在，让我们对原来的 SRCNN 模型做一些小的修改，体验一下模型动态化对模型部署造成的困难，并学习解决该问题的一种方法。

39.2 问题：实现动态放大的超分辨率模型

在原来的 SRCNN 中，图片的放大比例是写死在模型里的：

```
class SuperResolutionNet(nn.Module):
    def __init__(self, upscale_factor):
        super().__init__()
        self.upscale_factor = upscale_factor
        self.img_upsampler = nn.Upsample(
            scale_factor=self.upscale_factor,
            mode='bicubic',
            align_corners=False)

...

def init_torch_model():
    torch_model = SuperResolutionNet(upscale_factor=3)
```

我们使用 upscale_factor 来控制模型的放大比例。初始化模型的时候，我们默认令 upscale_factor 为 3，生成了一个放大 3 倍的 PyTorch 模型。这个 PyTorch 模型最终被转换成了 ONNX 格式的模型。如果我们需要一个放大 4 倍的模型，需要重新生成一遍模型，再做一次到 ONNX 的转换。

现在，假设我们要做一个超分辨率的应用。我们的用户希望图片的放大倍数能够自由设置。而我们交给用户的，只有一个.onnx 文件和运行超分辨率模型的应用程序。我们在不修改.onnx 文件的前提下改变放大倍数。

因此，我们必须修改原来的模型，令模型的放大倍数变成推理时的输入。在第一章中的 Python 脚本的基础上，我们做一些修改，得到这样的脚本：

```
import torch
from torch import nn
from torch.nn.functional import interpolate
import torch.onnx
import cv2
import numpy as np


class SuperResolutionNet(nn.Module):

    def __init__(self):
        super().__init__()
```

(下页继续)

(续上页)

```
        self.conv1 = nn.Conv2d(3, 64, kernel_size=9, padding=4)
        self.conv2 = nn.Conv2d(64, 32, kernel_size=1, padding=0)
        self.conv3 = nn.Conv2d(32, 3, kernel_size=5, padding=2)

        self.relu = nn.ReLU()

    def forward(self, x, upscale_factor):
        x = interpolate(x,
                        scale_factor=upscale_factor,
                        mode='bicubic',
                        align_corners=False)
        out = self.relu(self.conv1(x))
        out = self.relu(self.conv2(out))
        out = self.conv3(out)
        return out


def init_torch_model():
    torch_model = SuperResolutionNet()

    # Please read the code about downloading 'srcnn.pth' and 'face.png' in
    # https://mmdeploy.readthedocs.io/zh_CN/latest/tutorials/chapter_01_introduction_
↪to_model_deployment.html#pytorch
    state_dict = torch.load('srcnn.pth')['state_dict']

    # Adapt the checkpoint
    for old_key in list(state_dict.keys()):
        new_key = '.'.join(old_key.split('.')[1:])
        state_dict[new_key] = state_dict.pop(old_key)

    torch_model.load_state_dict(state_dict)
    torch_model.eval()
    return torch_model


model = init_torch_model()

input_img = cv2.imread('face.png').astype(np.float32)

# HWC to NCHW
input_img = np.transpose(input_img, [2, 0, 1])
input_img = np.expand_dims(input_img, 0)
```

(下页继续)

(续上页)

```
# Inference
torch_output = model(torch.from_numpy(input_img), 3).detach().numpy()

# NCHW to HWC
torch_output = np.squeeze(torch_output, 0)
torch_output = np.clip(torch_output, 0, 255)
torch_output = np.transpose(torch_output, [1, 2, 0]).astype(np.uint8)

# Show image
cv2.imwrite("face_torch_2.png", torch_output)
```

SuperResolutionNet 未修改之前，nn.Upsample 在初始化阶段固化了放大倍数，而 PyTorch 的 interpolate 插值算子可以在运行阶段选择放大倍数。因此，我们在新脚本中使用 interpolate 代替 nn.Upsample，从而让模型支持动态放大倍数的超分。在第 55 行使用模型推理时，我们把放大倍数设置为 3。最后，图片保存在文件”face_torch_2.png” 中。一切正常的话，”face_torch_2.png” 和”face_torch.png” 的内容一模一样。

通过简单的修改，PyTorch 模型已经支持了动态分辨率。现在我们来一下尝试导出模型：

```
x = torch.randn(1, 3, 256, 256)

with torch.no_grad():
    torch.onnx.export(model, (x, 3),
                      "srcnn2.onnx",
                      opset_version=11,
                      input_names=['input', 'factor'],
                      output_names=['output'])
```

运行这些脚本时，会报一长串错误。没办法，我们碰到了模型部署中的兼容性问题。

39.3 解决方法：自定义算子

直接使用 PyTorch 模型的话，我们修改几行代码就能实现模型输入的动态化。但在模型部署中，我们要花数倍的时间来设法解决这一问题。现在，让我们顺着解决问题的思路，体验一下模型部署的困难，并学习使用自定义算子的方式，解决超分辨率模型的动态化问题。

刚刚的报错是因为 PyTorch 模型在导出到 ONNX 模型时，模型的输入参数的类型必须全部是 torch.Tensor。而实际上我们传入的第二个参数” 3 ” 是一个整形变量。这不符合 PyTorch 转 ONNX 的规定。我们必须要修改一下原来的模型的输入。为了保证输入的所有参数都是 torch.Tensor 类型的，我们做如下修改：

```
...

class SuperResolutionNet(nn.Module):

```

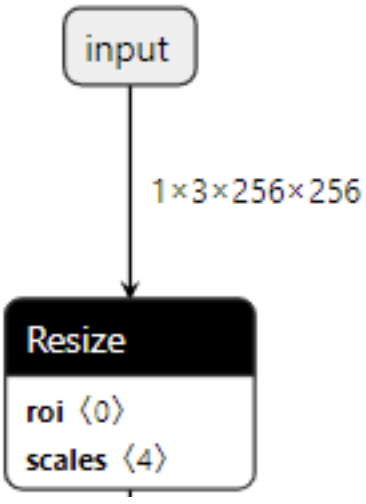
(下页继续)

(续上页)

```
    def forward(self, x, upscale_factor):
        x = interpolate(x,
                        scale_factor=upscale_factor.item(),
                        mode='bicubic',
                        align_corners=False)

...

# Inference
# Note that the second input is torch.tensor(3)
torch_output = model(torch.from_numpy(input_img), torch.tensor(3)).detach().numpy()

...

with torch.no_grad():
    torch.onnx.export(model, (x, torch.tensor(3)),
                      "srcnn2.onnx",
                      opset_version=11,
                      input_names=['input', 'factor'],
                      output_names=['output'])
```

由于 PyTorch 中 interpolate 的 scale_factor 参数必须是一个数值，我们使用 torch.Tensor.item() 来把只有一个元素的 torch.Tensor 转换成数值。之后，在模型推理时，我们使用 torch.tensor(3) 代替 3，以使得我们的所有输入都满足要求。现在运行脚本的话，无论是直接运行模型，还是导出 ONNX 模型，都不会报错了。

但是，导出 ONNX 时却报了一条 TraceWarning 的警告。这条警告说有一些量可能会追踪失败。这是怎么回事呢？让我们把生成的 srcnn2.onnx 用 Netron 可视化一下：



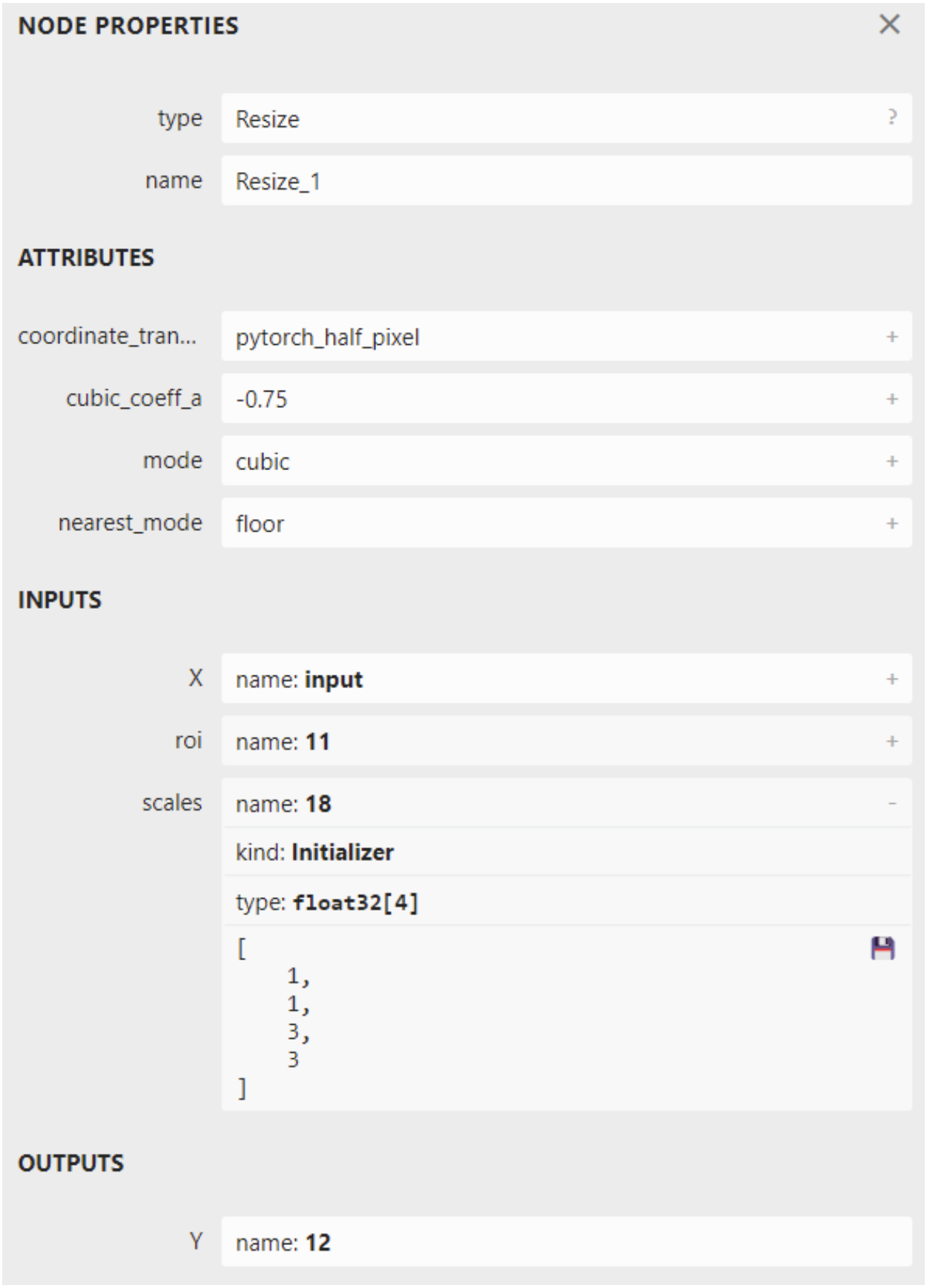
可以发现，虽然我们把模型推理的输入设置为了两个，但 ONNX 模型还是长得和原来一模一样，只有一个叫” input ” 的输入。这是由于我们使用了 torch.Tensor.item() 把数据从 Tensor 里取出来，而导出 ONNX 模型时

这个操作是无法被记录的，只好报了一条 TraceWarning。这导致 interpolate 插值函数的放大倍数还是被设置成了” 3 ” 这个固定值，我们导出的” srcnn2.onnx ” 和最开始的” srcnn.onnx ” 完全相同。

直接修改原来的模型似乎行不通，我们得从 PyTorch 转 ONNX 的原理入手，强行令 ONNX 模型明白我们的想法了。

仔细观察 Netron 上可视化出的 ONNX 模型，可以发现在 PyTorch 中无论是使用最早的 nn.Upsample，还是后来的 interpolate，PyTorch 里的插值操作最后都会转换成 ONNX 定义的 Resize 操作。也就是说，所谓 PyTorch 转 ONNX，实际上就是把每个 PyTorch 的操作映射成了 ONNX 定义的算子。

点击该算子，可以看到它的详细参数如下：


NODE PROPERTIES
type
Resize
name
Resize_1
ATTRIBUTES
coordinate_tran...
pytorch_half_pixel
cubic_coeff_a
-0.75
mode
cubic
nearest_mode
floor
INPUTS
X
name: input
roi
name: 11
scales
name: 18
kind: Initializer
type: float32[4]
[
1,
1,
3,
3
]
OUTPUTS
Y
name: 12

其中，展开 scales，可以看到 scales 是一个长度为 4 的一维张量，其内容为 [1, 1, 3, 3], 表示 Resize 操作每一个维度的缩放系数；其类型为 Initializer，表示这个值是根据常量直接初始化出来的。如果我们能够自己生成一个 ONNX 的 Resize 算子，让 scales 成为一个可变量而不是常量，就像它上面的 X 一样，那这个超分辨率模型就能动态缩放了。

现有实现插值的 PyTorch 算子有一套规定好的映射到 ONNX Resize 算子的方法，这些映射出的 Resize 算子的 scales 只能是常量，无法满足我们的需求。我们得自己定义一个实现插值的 PyTorch 算子，然后让它映射到一个我们期望的 ONNX Resize 算子上。

下面的脚本定义了一个 PyTorch 插值算子，并在模型里使用了它。我们先通过运行模型来验证该算子的正确性：

```
import torch
from torch import nn
from torch.nn.functional import interpolate
import torch.onnx
import cv2
import numpy as np


class NewInterpolate(torch.autograd.Function):

    @staticmethod
    def symbolic(g, input, scales):
        return g.op("Resize",
                    input,
                    g.op("Constant",
                         value_t=torch.tensor([], dtype=torch.float32)),
                    scales,
                    coordinate_transformation_mode_s="pytorch_half_pixel",
                    cubic_coeff_a_f=-0.75,
                    mode_s='cubic',
                    nearest_mode_s="floor")

    @staticmethod
    def forward(ctx, input, scales):
        scales = scales.tolist()[-2:]
        return interpolate(input,
                           scale_factor=scales,
                           mode='bicubic',
                           align_corners=False)


class StrangeSuperResolutionNet(nn.Module):
```

(下页继续)

(续上页)

```
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=9, padding=4)
        self.conv2 = nn.Conv2d(64, 32, kernel_size=1, padding=0)
        self.conv3 = nn.Conv2d(32, 3, kernel_size=5, padding=2)

        self.relu = nn.ReLU()

    def forward(self, x, upscale_factor):
        x = NewInterpolate.apply(x, upscale_factor)
        out = self.relu(self.conv1(x))
        out = self.relu(self.conv2(out))
        out = self.conv3(out)
        return out


def init_torch_model():
    torch_model = StrangeSuperResolutionNet()

    state_dict = torch.load('srcnn.pth')['state_dict']

    # Adapt the checkpoint
    for old_key in list(state_dict.keys()):
        new_key = '.'.join(old_key.split('.')[1:])
        state_dict[new_key] = state_dict.pop(old_key)

    torch_model.load_state_dict(state_dict)
    torch_model.eval()
    return torch_model


model = init_torch_model()
factor = torch.tensor([1, 1, 3, 3], dtype=torch.float)

input_img = cv2.imread('face.png').astype(np.float32)

# HWC to NCHW
input_img = np.transpose(input_img, [2, 0, 1])
input_img = np.expand_dims(input_img, 0)

# Inference
torch_output = model(torch.from_numpy(input_img), factor).detach().numpy()
```

(下页继续)

(续上页)

```
# NCHW to HWC
torch_output = np.squeeze(torch_output, 0)
torch_output = np.clip(torch_output, 0, 255)
torch_output = np.transpose(torch_output, [1, 2, 0]).astype(np.uint8)

# Show image
cv2.imwrite("face_torch_3.png", torch_output)
```

模型运行正常的话，一幅放大 3 倍的超分辨率图片会保存在”face_torch_3.png” 中，其内容和”face_torch.png” 完全相同。

在刚刚那个脚本中，我们定义 PyTorch 插值算子的代码如下：

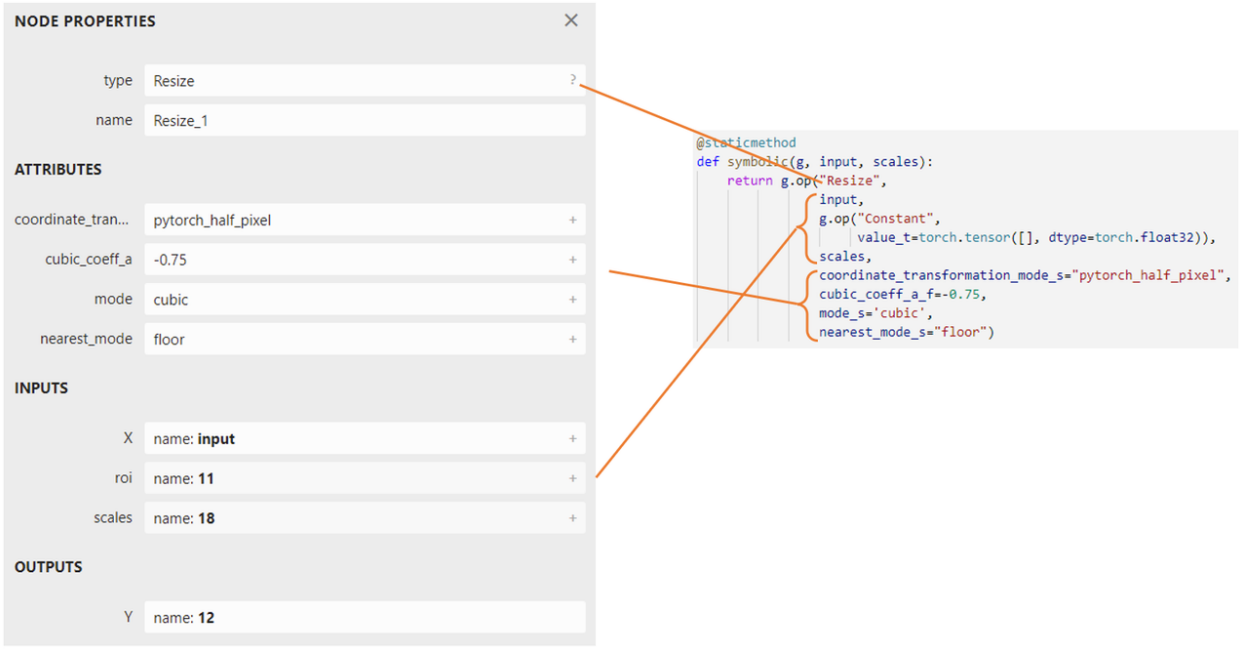
```
class NewInterpolate(torch.autograd.Function):

    @staticmethod
    def symbolic(g, input, scales):
        return g.op("Resize",
                    input,
                    g.op("Constant",
                         value_t=torch.tensor([], dtype=torch.float32)),
                    scales,
                    coordinate_transformation_mode_s="pytorch_half_pixel",
                    cubic_coeff_a_f=-0.75,
                    mode_s='cubic',
                    nearest_mode_s="floor")

    @staticmethod
    def forward(ctx, input, scales):
        scales = scales.tolist()[-2:]
        return interpolate(input,
                           scale_factor=scales,
                           mode='bicubic',
                           align_corners=False)
```

在具体介绍这个算子的实现前，让我们先理清一下思路。我们希望新的插值算子有两个输入，一个是被用于操作的图像，一个是图像的放缩比例。前面讲到，为了对接 ONNX 中 Resize 算子的 scales 参数，这个放缩比例是一个 [1, 1, x, x] 的张量，其中 x 为放大倍数。在之前放大 3 倍的模型中，这个参数被固定成了 [1, 1, 3, 3]。因此，在插值算子中，我们希望模型的第二个输入是一个 [1, 1, w, h] 的张量，其中 w 和 h 分别是图片宽和高的放大倍数。

搞清楚了插值算子的输入，再看一看算子的具体实现。算子的推理行为由算子的 forward 方法决定。该方法的第一个参数必须为 ctx，后面的参数为算子的自定义输入，我们设置两个输入，分别为被操作的图像和放缩比例。为保证推理正确，需要把 [1, 1, w, h] 格式的输入对接到原来的 interpolate 函数上。我们的做法是截取

输入张量的后两个元素，把这两个元素以 list 的格式传入 interpolate 的 scale_factor 参数。

接下来，我们要决定新算子映射到 ONNX 算子的方法。映射到 ONNX 的方法由一个算子的 symbolic 方法决定。symbolic 方法第一个参数必须是 g，之后的参数是算子的自定义输入，和 forward 函数一样。ONNX 算子的具体定义由 g.op 实现。g.op 的每个参数都可以映射到 ONNX 中的算子属性：

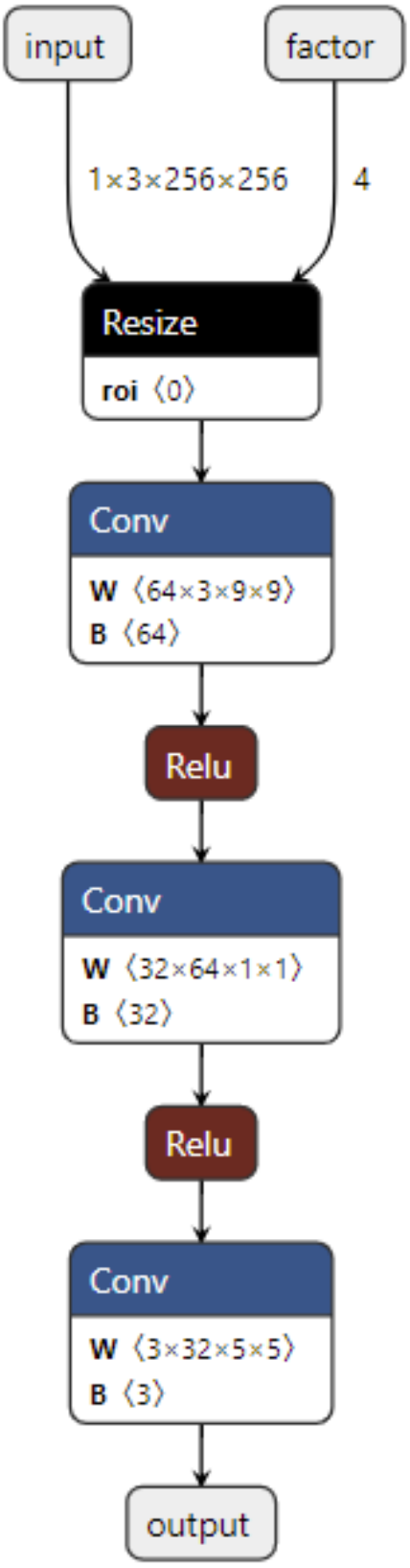


对于其他参数，我们可以照着现在的 Resize 算子填。而要注意的是，我们现在希望 scales 参数是由输入动态决定的。因此，在填入 ONNX 的 scales 时，我们要把 symbolic 方法的输入参数中的 scales 填入。

接着，让我们把新模型导出成 ONNX 模型：

```
x = torch.randn(1, 3, 256, 256)

with torch.no_grad():
    torch.onnx.export(model, (x, factor),
                      "srcnn3.onnx",
                      opset_version=11,
                      input_names=['input', 'factor'],
                      output_names=['output'])
```

把导出的” srcnn3.onnx ” 进行可视化：


input
factor
1×3×256×256
4
Resize
roi 〈0〉
Conv
W 〈64×3×9×9〉
B 〈64〉
Relu
Conv
W 〈32×64×1×1〉
B 〈32〉
Relu
Conv
W 〈3×32×5×5〉
B 〈3〉
output

可以看到，正如我们所期望的，导出的 ONNX 模型有了两个输入！第二个输入表示图像的放缩比例。

之前在验证 PyTorch 模型和导出 ONNX 模型时，我们宽高的缩放比例设置成了 3x3。现在，在用 ONNX Runtime 推理时，我们尝试使用 4x4 的缩放比例：

```
import onnxruntime

input_factor = np.array([1, 1, 4, 4], dtype=np.float32)
ort_session = onnxruntime.InferenceSession("srcnn3.onnx")
ort_inputs = {'input': input_img, 'factor': input_factor}
ort_output = ort_session.run(None, ort_inputs)[0]

ort_output = np.squeeze(ort_output, 0)
ort_output = np.clip(ort_output, 0, 255)
ort_output = np.transpose(ort_output, [1, 2, 0]).astype(np.uint8)
cv2.imwrite("face_ort_3.png", ort_output)
```

运行上面的代码，可以得到一个边长放大 4 倍的超分辨率图片”face_ort_3.png”。动态的超分辨率模型生成成功了！只要修改 input_factor，我们就可以自由地控制图片的缩放比例。

我们刚刚的工作，实际上是绕过 PyTorch 本身的限制，凭空“捏”出了一个 ONNX 算子。事实上，我们不仅可以创建现有的 ONNX 算子，还可以定义新的 ONNX 算子以拓展 ONNX 的表达能力。后续教程中我们将介绍自定义新 ONNX 算子的方法。

39.4 总结

通过学习前两篇教程，我们走完了整个部署流水线，成功部署了支持动态放大倍数的超分辨率模型。在这个过程中，我们既学会了如何简单地调用各框架的 API 实现模型部署，又学到了如何分析并尝试解决模型部署时碰到的难题。

同样，让我们总结一下本篇教程的知识点：

- 模型部署中常见的几类困难有：模型的动态化；新算子的实现；框架间的兼容。
- PyTorch 转 ONNX，实际上就是把每一个操作转化成 ONNX 定义的某一个算子。比如对于 PyTorch 中的 Upsample 和 interpolate，在转 ONNX 后最终都会成为 ONNX 的 Resize 算子。
- 通过修改继承自 torch.autograd.Function 的算子的 symbolic 方法，可以改变该算子映射到 ONNX 算子的行为。

至此，” 部署第一个模型 “的教程算是告一段落了。是不是觉得学到的知识还不够多？没关系，在接下来的几篇教程中，我们将结合 MMDeploy ，重点介绍 ONNX 中间表示和 ONNX Runtime/TensorRT 推理引擎的知识，让大家学会如何部署更复杂的模型。

CHAPTER 40

第三章：PyTorch 转 ONNX 详解

ONNX 是目前模型部署中最重要的中间表示之一。学懂了 ONNX 的技术细节，就能规避大量的模型部署问题。从这篇文章开始，在接下来的三篇文章里，我们将由浅入深地介绍 ONNX 相关的知识。在第一篇文章里，我们会介绍更多 PyTorch 转 ONNX 的细节，让大家完全掌握把简单的 PyTorch 模型转成 ONNX 模型的方法；在第二篇文章里，我们将介绍如何在 PyTorch 中支持更多的 ONNX 算子，让大家能彻底走通 PyTorch 到 ONNX 这条部署路线；第三篇文章里，我们讲介绍 ONNX 本身的知识，以及修改、调试 ONNX 模型的常用方法，使大家能自行解决大部分和 ONNX 有关的部署问题。

在把 PyTorch 模型转换成 ONNX 模型时，我们往往只需要轻松地调用一句 `torch.onnx.export` 就行了。这个函数的接口看上去简单，但它在使用上还有着诸多的“潜规则”。在这篇教程中，我们会详细介绍 PyTorch 模型转 ONNX 模型的原理及注意事项。除此之外，我们还会介绍 PyTorch 与 ONNX 的算子对应关系，以教会大家如何处理 PyTorch 模型转换时可能会遇到的算子支持问题。

40.1 torch.onnx.export 细解

在这一节里，我们将详细介绍 PyTorch 到 ONNX 的转换函数——torch.onnx.export。我们希望大家能够更加灵活地使用这个模型转换接口，并通过了解它的实现原理来更好地应对该函数的报错（由于模型部署的兼容性问题，部署复杂模型时该函数时常会报错）。

40.1.1 计算图导出方法

TorchScript 是一种序列化和优化 PyTorch 模型的格式，在优化过程中，一个 `torch.nn.Module` 模型会被转换成 TorchScript 的 `torch.jit.ScriptModule` 模型。现在，TorchScript 也被常当成一种中间表示使用。我们在其他文章中对 TorchScript 有详细的介绍，这里介绍 TorchScript 仅用于说明 PyTorch 模型转 ONNX 的原理。`torch.onnx.export` 中需要的模型实际上是一个 `torch.jit.ScriptModule`。而要把普通 PyTorch 模型转一个这样的 TorchScript 模型，有跟踪（trace）和脚本化（script）两种导出计算图的方法。如果给 `torch.onnx.export` 传入了一个普通 PyTorch 模型（`torch.nn.Module`），那么这个模型会默认使用跟踪的方法导出。这一过程如下图所示：

回忆一下我们第一篇教程 知识：跟踪法只能通过实际运行一遍模型的方法导出模型的静态图，即无法识别出模型中的控制流（如循环）；脚本化则能通过解析模型来正确记录所有的控制流。我们以下面这段代码为例来看一看这两种转换方法的区别：

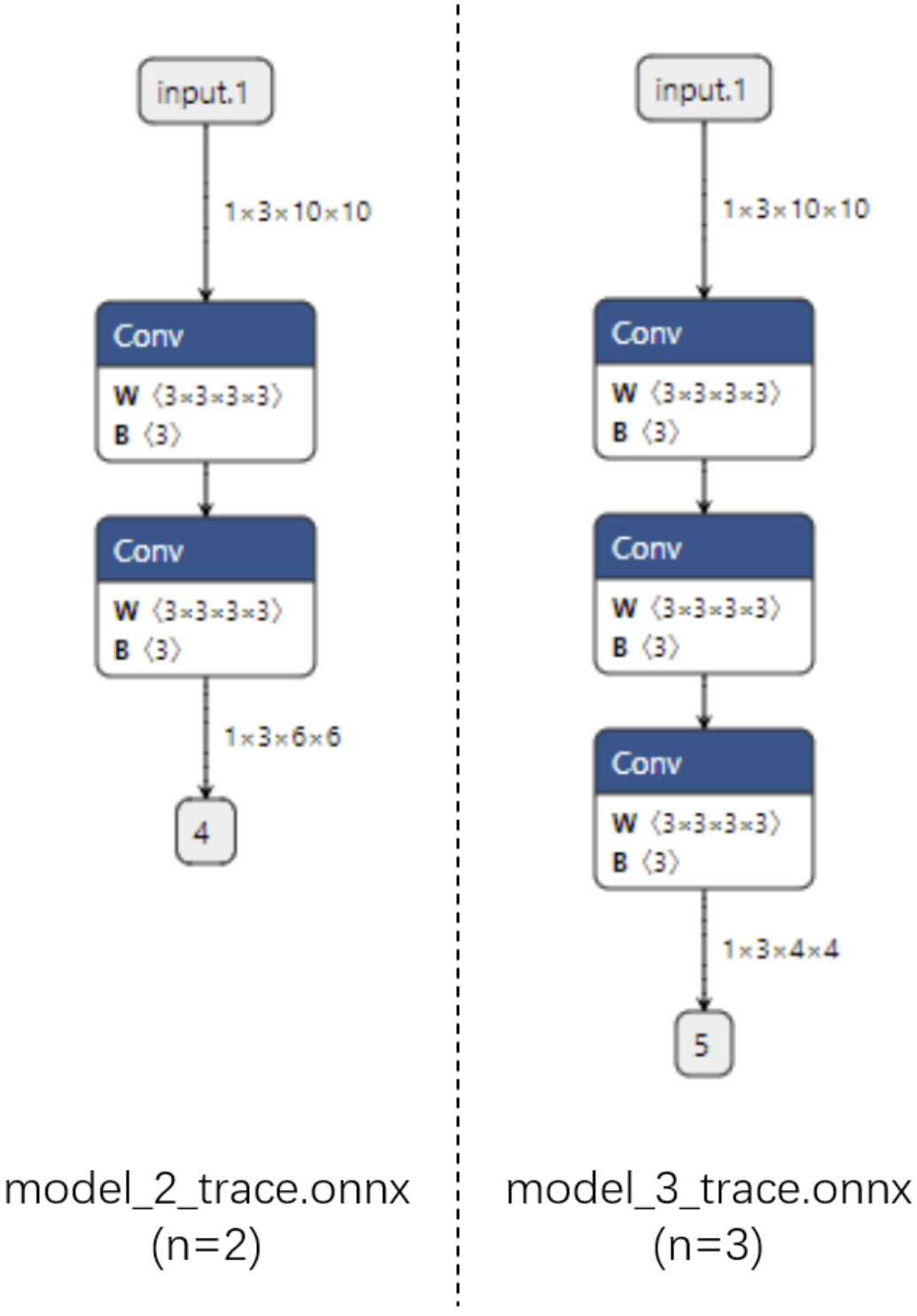
```
import torch

class Model(torch.nn.Module):
    def __init__(self, n):
        super().__init__()
        self.n = n
        self.conv = torch.nn.Conv2d(3, 3, 3)

    def forward(self, x):
        for i in range(self.n):
            x = self.conv(x)
        return x
```

(下页继续)

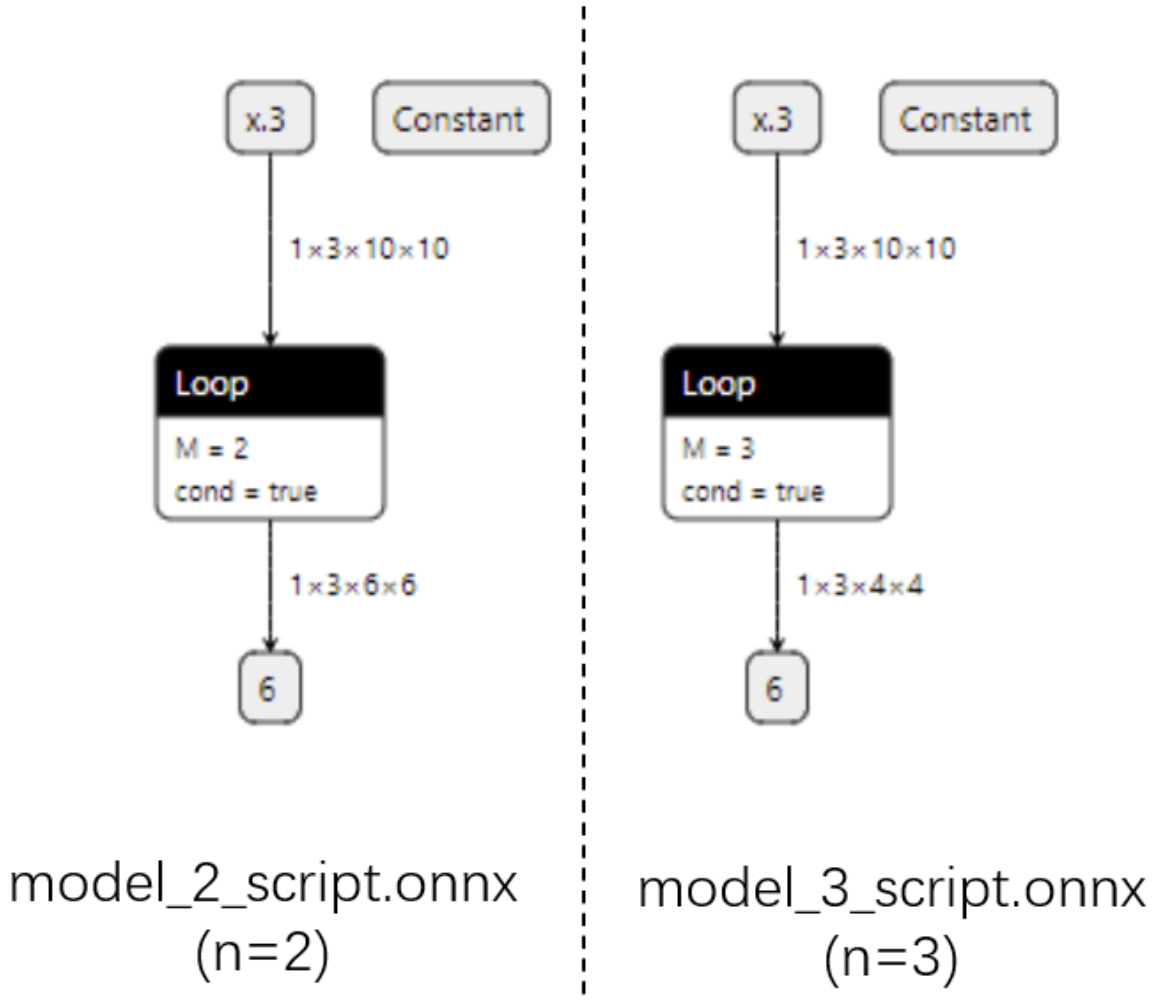
(续上页)

```
models = [Model(2), Model(3)]
model_names = ['model_2', 'model_3']

for model, model_name in zip(models, model_names):
    dummy_input = torch.rand(1, 3, 10, 10)
    dummy_output = model(dummy_input)
    model_trace = torch.jit.trace(model, dummy_input)
    model_script = torch.jit.script(model)

    # 跟踪法与直接 torch.onnx.export(model, ...) 等价
    torch.onnx.export(model_trace, dummy_input, f'{model_name}_trace.onnx', example_
↪outputs=dummy_output)
    # 脚本化必须先调用 torch.jit.sciprt
    torch.onnx.export(model_script, dummy_input, f'{model_name}_script.onnx', example_
↪outputs=dummy_output)
```

在这段代码里，我们定义了一个带循环的模型，模型通过参数 `n` 来控制输入张量被卷积的次数。之后，我们各创建了一个 `n=2` 和 `n=3` 的模型。我们把这两个模型分别用跟踪和脚本化的方法进行导出。值得一提的是，由于这里的两个模型（`model_trace, model_script`）是 TorchScript 模型，`export` 函数已经不需要再运行一遍模型了。（如果模型是用跟踪法得到的，那么在执行 `torch.jit.trace` 的时候就运行过一遍了；而用脚本化导出时，模型不需要实际运行）参数中的 `dummy_input` 和 `dummy_output` 仅仅是为了获取输入和输出张量的类型和形状。运行上面的代码，我们把得到的 4 个 onnx 文件用 Netron 可视化：



首先看跟踪法得到的 ONNX 模型结构。可以看出来，对于不同的 n,ONNX 模型的结构是不一样的。



而用脚本化的话，最终的 ONNX 模型用 `Loop` 节点来表示循环。这样哪怕对于不同的 `n`，ONNX 模型也有同样的结构。由于推理引擎对静态图的支持更好，通常我们在模型部署时不需要显式地把 PyTorch 模型转成 TorchScript 模型，直接把 PyTorch 模型用 `torch.onnx.export` 跟踪导出即可。了解这部分的知识主要是为了在模型转换报错时能够更好地定位问题是否发生在 PyTorch 转 TorchScript 阶段。

40.1.2 参数讲解

了解完转换函数的原理后，我们来详细介绍一下该函数的主要参数的作用。我们主要会从应用的角度来介绍每个参数在不同的模型部署场景中应该如何设置，而不会去列出每个参数的所有设置方法。该函数详细的 API 文档可参考 torch.onnx – PyTorch 1.11.0 documentation

`torch.onnx.export` 在 `torch.onnx.__init__.py` 文件中的定义如下：

```
def export(model, args, f, export_params=True, verbose=False, training=TrainingMode.
↪EVAL,
```

(下页继续)

(续上页)

```
            input_names=None, output_names=None, aten=False, export_raw_ir=False,
            operator_export_type=None, opset_version=None, _retain_param_name=True,
            do_constant_folding=True, example_outputs=None, strip_doc_string=True,
            dynamic_axes=None, keep_initializers_as_inputs=None, custom_opsets=None,
            enable_onnx_checker=True, use_external_data_format=False):
```

前三个必选参数为模型、模型输入、导出的 onnx 文件名，我们对这几个参数已经很熟悉了。我们来着重看一下后面的一些常用可选参数。

export_params

模型中是否存储模型权重。一般中间表示包含两大类信息：模型结构和模型权重，这两类信息可以在同一个文件里存储，也可以分文件存储。ONNX 是用同一个文件表示记录模型的结构和权重的。我们部署时一般都默认这个参数为 True。如果 onnx 文件是用来在不同框架间传递模型（比如 PyTorch 到 Tensorflow）而不是用于部署，则可以令这个参数为 False。

input_names, output_names

设置输入和输出张量的名称。如果不设置的话，会自动分配一些简单的名字（如数字）。ONNX 模型的每个输入和输出张量都有一个名字。很多推理引擎在运行 ONNX 文件时，都需要以“名称-张量值”的数据对来输入数据，并根据输出张量的名称来获取输出数据。在进行跟张量有关的设置（比如添加动态维度）时，也需要知道张量的名字。在实际的部署流水线中，我们都需要设置输入和输出张量的名称，并保证 ONNX 和推理引擎中使用同一套名称。

opset_version

转换时参考哪个 ONNX 算子集版本，默认为 9。后文会详细介绍 PyTorch 与 ONNX 的算子对应关系。

dynamic_axes

指定输入输出张量的哪些维度是动态的。为了追求效率，ONNX 默认所有参与运算的张量都是静态的（张量的形状不发生改变）。但在实际应用中，我们又希望模型的输入张量是动态的，尤其是本来就没有形状限制的全卷积模型。因此，我们需要显式地指明输入输出张量的哪几个维度的大小是可变的。我们来看一个 `dynamic_axes` 的设置例子：

```
import torch

class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = torch.nn.Conv2d(3, 3, 3)
```

(下页继续)

(续上页)

```
    def forward(self, x):
        x = self.conv(x)
        return x


model = Model()
dummy_input = torch.rand(1, 3, 10, 10)
model_names = ['model_static.onnx',
'model_dynamic_0.onnx',
'model_dynamic_23.onnx']

dynamic_axes_0 = {
    'in' : [0],
    'out' : [0]
}
dynamic_axes_23 = {
    'in' : [2, 3],
    'out' : [2, 3]
}

torch.onnx.export(model, dummy_input, model_names[0],
input_names=['in'], output_names=['out'])
torch.onnx.export(model, dummy_input, model_names[1],
input_names=['in'], output_names=['out'], dynamic_axes=dynamic_axes_0)
torch.onnx.export(model, dummy_input, model_names[2],
input_names=['in'], output_names=['out'], dynamic_axes=dynamic_axes_23)
```

首先，我们导出 3 个 ONNX 模型，分别为没有动态维度、第 0 维动态、第 2 第 3 维动态的模型。在这份代码里，我们是用列表的方式表示动态维度，例如：

```
dynamic_axes_0 = {
    'in' : [0],
    'out' : [0]
}
```

由于 ONNX 要求每个动态维度都有一个名字，这样写的话会引出一条 UserWarning，警告我们通过列表的方式设置动态维度的话系统会自动为它们分配名字。一种显式添加动态维度名字的方法如下：

```python
dynamic_axes_0 = {
    'in' : {0: 'batch'},
    'out' : {0: 'batch'}
}
```

由于在这份代码里我们没有更多的对动态维度的操作，因此简单地用列表指定动态维度即可。之后，我们用下面的代码来看一看动态维度的作用：

```
import onnxruntime
import numpy as np

origin_tensor = np.random.rand(1, 3, 10, 10).astype(np.float32)
mult_batch_tensor = np.random.rand(2, 3, 10, 10).astype(np.float32)
big_tensor = np.random.rand(1, 3, 20, 20).astype(np.float32)

inputs = [origin_tensor, mult_batch_tensor, big_tensor]
exceptions = dict()

for model_name in model_names:
    for i, input in enumerate(inputs):
        try:
            ort_session = onnxruntime.InferenceSession(model_name)
            ort_inputs = {'in': input}
            ort_session.run(['out'], ort_inputs)
        except Exception as e:
            exceptions[(i, model_name)] = e
            print(f'Input[{i}] on model {model_name} error.')
        else:
            print(f'Input[{i}] on model {model_name} succeed.')
```

我们在模型导出计算图时用的是一个形状为 `(1, 3, 10, 10)` 的张量。现在，我们来尝试以形状分别是 `(1, 3, 10, 10)`, `(2, 3, 10, 10)`, `(1, 3, 20, 20)` 为输入，用 ONNX Runtime 运行一下这几个模型，看看哪些情况下会报错，并保存对应的报错信息。得到的输出信息应该如下：

```
Input[0] on model model_static.onnx succeed.
Input[1] on model model_static.onnx error.
Input[2] on model model_static.onnx error.
Input[0] on model model_dynamic_0.onnx succeed.
Input[1] on model model_dynamic_0.onnx succeed.
Input[2] on model model_dynamic_0.onnx error.
Input[0] on model model_dynamic_23.onnx succeed.
Input[1] on model model_dynamic_23.onnx error.
Input[2] on model model_dynamic_23.onnx succeed.
```

可以看出，形状相同的 `(1, 3, 10, 10)` 的输入在所有模型上都没有出错。而对于 batch（第 0 维）或者长宽（第 2、3 维）不同的输入，只有在设置了对应的动态维度后才不会出错。我们可以错误信息中找出是哪些维度出了问题。比如我们可以用以下代码查看 `input[1]` 在 `model_static.onnx` 中的报错信息：

```
print(exceptions[(1, 'model_static.onnx')])
```

(下页继续)

(续上页)

```
# output
# [ONNXRuntimeError] : 2 : INVALID_ARGUMENT : Got invalid dimensions for input: in␣
↪for the following indices index: 0 Got: 2 Expected: 1 Please fix either the inputs␣
↪or the model.
```

这段报错告诉我们名字叫 `in` 的输入的第 0 维不匹配。本来该维的长度应该为 1，但我们的输入是 2。实际部署中，如果我们碰到了类似的报错，就可以通过设置动态维度来解决问题。

40.1.3 使用技巧

通过学习之前的知识，我们基本掌握了 `torch.onnx.export` 函数的部分实现原理和参数设置方法，足以完成简单模型的转换了。但在实际应用中，使用该函数还会踩很多坑。这里我们模型部署团队把在实战中积累的一些经验分享给大家。

使模型在 ONNX 转换时有不同的行为

有些时候，我们希望模型在直接用 PyTorch 推理时有一套逻辑，而在导出的 ONNX 模型中有另一套逻辑。比如，我们可以把一些后处理的逻辑放在模型里，以简化除运行模型之外的其他代码。`torch.onnx.is_in_onnx_export()` 可以实现这一任务，该函数仅在执行 `torch.onnx.export()` 时为真。以下是一个例子：

```
import torch

class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = torch.nn.Conv2d(3, 3, 3)

    def forward(self, x):
        x = self.conv(x)
        if torch.onnx.is_in_onnx_export():
            x = torch.clip(x, 0, 1)
        return x
```

这里，我们仅在模型导出时把输出张量的数值限制在 [0, 1] 之间。使用 `is_in_onnx_export` 确实能让我们方便地在代码中添加和模型部署相关的逻辑。但是，这些代码对只关心模型训练的开发者和用户来说很不友好，突兀的部署逻辑会降低代码整体的可读性。同时，`is_in_onnx_export` 只能在每个需要添加部署逻辑的地方都“打补丁”，难以进行统一的管理。我们之后会介绍如何使用 MMDeploy 的重写机制来规避这些问题。

利用中断张量跟踪的操作

PyTorch 转 ONNX 的跟踪导出法是不是万能的。如果我们在模型中做了一些很“出格”的操作，跟踪法会把某些取决于输入的中间结果变成常量，从而使导出的 ONNX 模型和原来的模型有出入。以下是一个会造成这种“跟踪中断”的例子：

```
class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = x * x[0].item()
        return x, torch.Tensor([i for i in x])

model = Model()
dummy_input = torch.rand(10)
torch.onnx.export(model, dummy_input, 'a.onnx')
```

如果你尝试去导出这个模型，会得到一大堆 warning，告诉你转换出来的模型可能不正确。这也难怪，我们在这个模型里使用了`.item()`把 torch 中的张量转换成了普通的 Python 变量，还尝试遍历 torch 张量，并用一个列表新建一个 torch 张量。这些涉及张量与普通变量转换的逻辑都会导致最终的 ONNX 模型不太正确。另一方面，我们也可以利用这个性质，在保证正确性的前提下令模型的中间结果变成常量。这个技巧常常用于模型的静态化上，即令模型中所有的张量形状都变成常量。在未来的教程中，我们会在部署实例中详细介绍这些“高级”操作。

使用张量为输入（PyTorch 版本 < 1.9.0）

正如我们第一篇教程所展示的，在较旧 (< 1.9.0) 的 PyTorch 中把 Python 数值作为`torch.onnx.export()`的模型输入时会报错。出于兼容性的考虑，我们还是推荐以张量为模型转换时的模型输入。

40.2 PyTorch 对 ONNX 的算子支持

在确保`torch.onnx.export()`的调用方法无误后，PyTorch 转 ONNX 时最容易出现的问题就是算子不兼容了。这里我们会介绍如何判断某个 PyTorch 算子在 ONNX 中是否兼容，以助大家在碰到报错时能更好地把错误归类。而具体添加算子的方法我们会在之后的文章里介绍。在转换普通的`torch.nn.Module`模型时，PyTorch 一方面会用跟踪法执行前向推理，把遇到的算子整合成计算图；另一方面，PyTorch 还会把遇到的每个算子翻译成 ONNX 中定义的算子。在这个翻译过程中，可能会碰到以下情况：

- 该算子可以一对一地翻译成一个 ONNX 算子。
- 该算子在 ONNX 中没有直接对应的算子，会翻译成一至多个 ONNX 算子。
- 该算子没有定义翻译成 ONNX 的规则，报错。

那么，该如何查看 PyTorch 算子与 ONNX 算子的对应情况呢？由于 PyTorch 算子是向 ONNX 对齐的，这里我们先看一下 ONNX 算子的定义情况，再看一下 PyTorch 定义的算子映射关系。

40.2.1 ONNX 算子文档

ONNX 算子的定义情况，都可以在官方的算子文档中查看。这份文档十分重要，我们碰到任何和 ONNX 算子有关的问题都得来”请教“这份文档。

ai.onnx (default)

Operator	Since version
Abs	13, 6, 1
Acos	7
Acosh	9
Add	14, 13, 7, 6, 1
And	7, 1
ArgMax	13, 12, 11, 1
ArgMin	13, 12, 11, 1
Asin	7

这份文档中最重要的开头的这个算子变更表格。表格的第一列是算子名，第二列是该算子发生变动的算子集版本号，也就是我们之前在 `torch.onnx.export` 中提到的 `opset_version` 表示的算子集版本号。通过查看算子第一次发生变动的版本号，我们可以知道某个算子是从哪个版本开始支持的；通过查看某算子小于等于 `opset_version` 的第一个改动记录，我们可以知道当前算子集版本中该算子的定义规则。

Relu

Relu takes one input data (Tensor) and produces one output data (Tensor) where the rectified linear function, y = max(0, x), is applied to the tensor elementwise.

Version

This version of the operator has been available since version 14 of the default ONNX operator set.

Other versions of this operator: 1, 6, 13

Inputs

`X` ***(differentiable) : T***
Input tensor

Outputs

`Y` ***(differentiable) : T***
Output tensor

Type Constraints

`T` ***: tensor(float), tensor(int32), tensor(int8), tensor(int16), tensor(int64), tensor(float16), tensor(double), tensor(bfloat16)***
Constrain input and output types to signed numeric tensors.

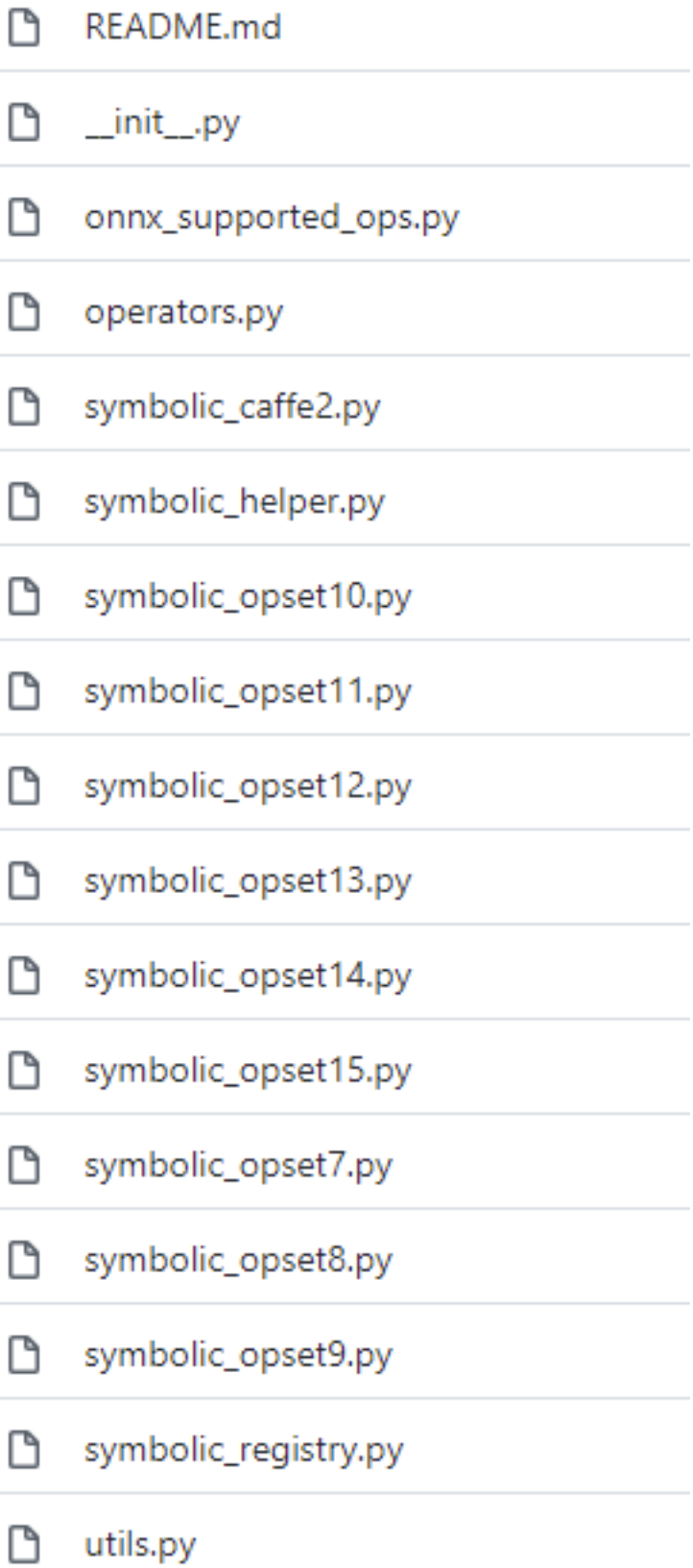
Examples

▶ relu

通过点击表格中的链接，我们可以查看某个算子的输入、输出参数规定及使用示例。比如上图是 Relu 在 ONNX 中的定义规则，这份定义表明 Relu 应该有一个输入和一个输入，输入输出的类型相同，均为 tensor。

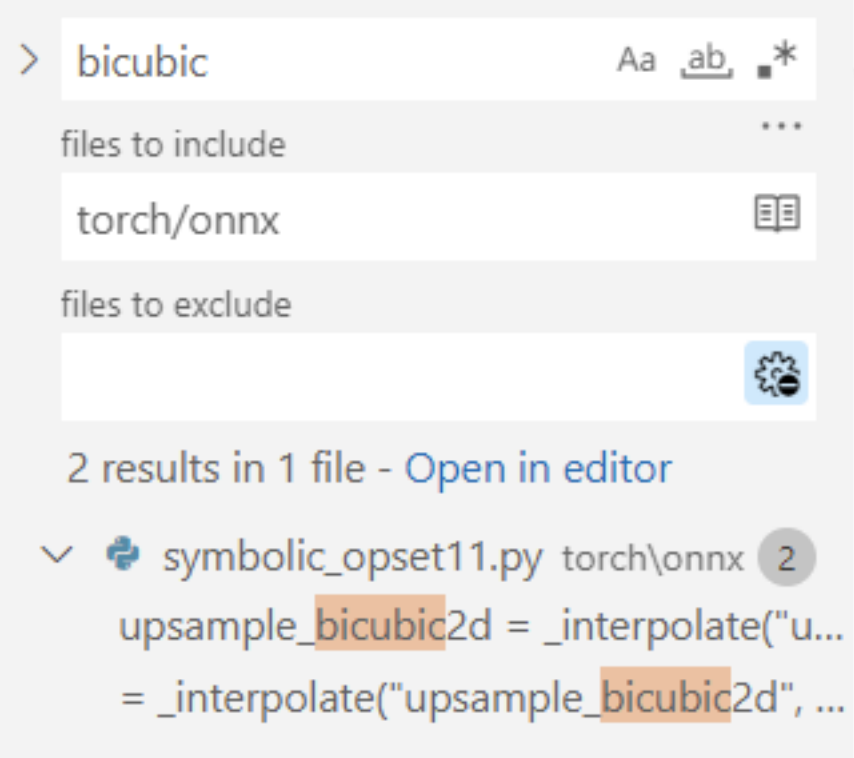
40.2.2 PyTorch 对 ONNX 算子的映射

在 PyTorch 中，和 ONNX 有关的定义全部放在 torch.onnx 目录中，如下图所示：



其中，`symbloic_opset{n}.py`（符号表文件）即表示 PyTorch 在支持第 n 版 ONNX 算子集时新加入的内

容。我们之前讲过，bicubic 插值是在第 11 个版本开始支持的。我们以它为例来看看如何查找算子的映射情况。首先，使用搜索功能，在 `torch/onnx` 文件夹搜索"bicubic"，可以发现这个这个插值在第 11 个版本的定义文件中：



之后，我们按照代码的调用逻辑，逐步跳转直到最底层的 ONNX 映射函数：

```
upsample_bicubic2d = _interpolate("upsample_bicubic2d", 4, "cubic")

->

def _interpolate(name, dim, interpolate_mode):
    return sym_help._interpolate_helper(name, dim, interpolate_mode)

->

def _interpolate_helper(name, dim, interpolate_mode):
    def symbolic_fn(g, input, output_size, *args):
        ...

    return symbolic_fn
```

最后，在 `symbolic_fn` 中，我们可以看到插值算子是怎么样被映射成多个 ONNX 算子的。其中，每一个 `g.op` 就是一个 ONNX 的定义。比如其中的 `Resize` 算子就是这样写的：

```
    return g.op("Resize",
                input,
                empty_roi,
                empty_scales,
                output_size,
                coordinate_transformation_mode_s=coordinate_transformation_mode,
                cubic_coeff_a_f=-0.75,  # only valid when mode="cubic"
                mode_s=interpolate_mode,  # nearest, linear, or cubic
                nearest_mode_s="floor")  # only valid when mode="nearest"
```

通过在前面提到的 ONNX 算子文档中查找 Resize 算子的定义，我们就可以知道这每一个参数的含义了。用类似的方法，我们可以去查询其他 ONNX 算子的参数含义，进而知道 PyTorch 中的参数是怎样一步一步传入到每个 ONNX 算子中的。掌握了如何查询 PyTorch 映射到 ONNX 的关系后，我们在实际应用时就可以在 `torch.onnx.export()` 的 `opset_version` 中先预设一个版本号，碰到了问题就去对应的 PyTorch 符号表文件里去查。如果某算子确实不存在，或者算子的映射关系不满足我们的要求，我们就可能得用其他的算子绕过去，或者自定义算子了。

40.3 总结

在这篇教程中，我们系统地介绍了 PyTorch 转 ONNX 的原理。我们先是着重讲解了使用最频繁的 `torch.onnx.export` 函数，又给出了查询 PyTorch 对 ONNX 算子支持情况的方法。通过本文，我们希望大家能够成功转换出大部分不需要添加新算子的 ONNX 模型，并在碰到算子问题时能够有效定位问题原因。具体而言，大家读完本文后应该了解以下的知识：

- 跟踪法和脚本化在导出带控制语句的计算图时有什么区别。
- `torch.onnx.export()` 中该如何设置 `input_names, output_names, dynamic_axes`。
- 使用 `torch.onnx.is_in_onnx_export()` 来使模型在转换到 ONNX 时有不同的行为。
- 如何查询 ONNX 算子文档。
- 如何查询 PyTorch 对某个 ONNX 版本的新特性支持情况。
- 如何判断 PyTorch 对某个 ONNX 算子是否支持，支持的方法是怎样的。

这期介绍的知识比较抽象，大家会不会觉得有点“水”？没关系，下一篇教程中，我们将以给出代码实例的形式，介绍多种为 PyTorch 转 ONNX 添加算子支持的方法，为大家在 PyTorch 转 ONNX 这条路上扫除更多的障碍。

40.4 练习

1. Asinh 算子出现于第 9 个 ONNX 算子集。PyTorch 在 9 号版本的符号表文件中是怎样支持这个算子的?
2. BitShift 算子出现于第 11 个 ONNX 算子集。PyTorch 在 11 号版本的符号表文件中是怎样支持这个算子的?
3. 在第一篇教程 中，我们讲过 PyTorch （截至第 11 号算子集）不支持在插值中设置动态的放缩系数。这个系数对应 `torch.onnx.symbolic_helper._interpolate_helper` 的 symbolic_fn 的 Resize 算子映射关系中的哪个参数? 我们是如何修改这一参数的?

练习的答案会在下期教程中揭晓。

CHAPTER 41

第四章：在 PyTorch 中支持更多 ONNX 算子

在上一篇教程中，我们系统地学习了 PyTorch 转 ONNX 的方法，可以发现 PyTorch 对 ONNX 的支持还不错。但在实际的部署过程中，难免碰到模型无法用原生 PyTorch 算子表示的情况。这个时候，我们就得考虑扩充 PyTorch，即在 PyTorch 中支持更多 ONNX 算子。

而要使 PyTorch 算子顺利转换到 ONNX ，我们需要保证以下三个环节都不出错：

- 算子在 PyTorch 中有实现
- 有把该 PyTorch 算子映射成一个或多个 ONNX 算子的方法
- ONNX 有相应的算子

可在实际部署中，这三部分的内容都可能有所缺失。其中最坏的情况是：我们定义了一个全新的算子，它不仅缺少 PyTorch 实现，还缺少 PyTorch 到 ONNX 的映射关系。但所谓车到山前必有路，对于这三个环节，我们也分别都有以下的添加支持的方法：

- PyTorch 算子
  - 组合现有算子
  - 添加 TorchScript 算子
  - 添加普通 C++ 拓展算子
- 映射方法
  - 为 ATen 算子添加符号函数
  - 为 TorchScript 算子添加符号函数
  - 封装成 torch.autograd.Function 并添加符号函数

- ONNX 算子
  - 使用现有 ONNX 算子
  - 定义新 ONNX 算子

那么，面对不同的情况时，就需要我们灵活地选用和组合这些方法。听起来是不是很复杂? 别担心，本篇文章中，我们将围绕着三种算子映射方法，学习三个添加算子支持的实例，来理清如何为 PyTorch 算子转 ONNX 算子的三个环节添加支持。

41.1 支持 ATen 算子

实际的部署过程中，我们都有可能会碰到一个最简单的算子缺失问题：算子在 ATen 中已经实现了，ONNX 中也有相关算子的定义，但是相关算子映射成 ONNX 的规则没有写。在这种情况下，我们只需要**为 ATen 算子补充描述映射规则的符号函数**就行了。

ATen 是 PyTorch 内置的 C++ 张量计算库，PyTorch 算子在底层绝大多数计算都是用 ATen 实现的。

上期习题中，我们曾经提到了 ONNX 的 `Asinh` 算子。这个算子在 ATen 中有实现，却缺少了映射到 ONNX 算子的符号函数。在这里，我们来尝试为它补充符号函数，并导出一个包含这个算子的 ONNX 模型。

41.1.1 获取 ATen 中算子接口定义

为了编写符号函数，我们需要获得 `asinh` 推理接口的输入参数定义。这时，我们要去 `torch/_C/_VariableFunctions.pyi` 和 `torch/nn/functional.pyi` 这两个文件中搜索我们刚刚得到的这个算子名。这两个文件是编译 PyTorch 时本地自动生成的文件，里面包含了 ATen 算子的 PyTorch 调用接口。通过搜索，我们可以知道 `asinh` 在文件 `torch/_C/_VariableFunctions.pyi` 中，其接口定义为:

```
def asinh(input: Tensor, *, out: Optional[Tensor]=None) -> Tensor: ...
```

经过这些步骤，我们确认了缺失的算子名为 `asinh`，它是一个有实现的 ATen 算子。我们还记下了 `asinh` 的调用接口。接下来，我们要为它补充符号函数，使它在转换成 ONNX 模型时不再报错。

41.1.2 添加符号函数

到目前为止，我们已经多次接触了定义 PyTorch 到 ONNX 映射规则的符号函数了。现在，我们向大家正式介绍一下符号函数。

符号函数，可以看成是 PyTorch 算子类的一个静态方法。在把 PyTorch 模型转换成 ONNX 模型时，各个 PyTorch 算子的符号函数会被依次调用，以完成 PyTorch 算子到 ONNX 算子的转换。符号函数的定义一般如下:

```
def symbolic(g: torch._C.Graph, input_0: torch._C.Value, input_1: torch._C.Value, ...
↪):
```

其中，torch._C.Graph 和 torch._C.Value 都对应 PyTorch 的 C++ 实现里的一些类。我们在这篇文章不深究它们的细节，只需要知道第一个参数就固定叫 g，它表示和计算图相关的内容；后面的每个参数都表示算子的输入，需要和算子的前向推理接口的输入相同。对于 ATen 算子来说，它们的前向推理接口就是上述两个 .pyi 文件里的函数接口。

g 有一个方法 op。在把 PyTorch 算子转换成 ONNX 算子时，需要在符号函数中调用此方法来为最终的计算图添加一个 ONNX 算子。其定义如下：

```
def op(name: str, input_0: torch._C.Value, input_1: torch._C.Value, ...)
```

其中，第一个参数是算子名称。如果该算子是普通的 ONNX 算子，只需要把它在 ONNX 官方文档里的名称填进去即可（我们稍后再讲其他情况）。

在最简单的情况下，我们只要把 PyTorch 算子的输入用 g.op() 一一对应到 ONNX 算子上即可，并把 g.op() 的返回值作为符号函数的返回值。在情况更复杂时，我们转换一个 PyTorch 算子可能要新建若干个 ONNX 算子。

补充完了背景知识，让我们回到 asinh 算子上，来为它编写符号函数。我们先去翻阅一下 ONNX 算子文档，学习一下我们在符号函数里的映射关系 g.op() 里应该怎么写。Asinh 的文档写道：该算子有一个输入 input，一个输出 output，二者的类型都为张量。

到这里，我们已经完成了信息收集环节。我们在上一小节得知了 asinh 的推理接口定义，在这一小节里收集了 ONNX 算子 Asinh 的定义。现在，我们可以用代码来补充这二者的映射关系了。在刚刚导出 asinh 算子的代码中，我们添加以下内容：

```
from torch.onnx.symbolic_registry import register_op

def asinh_symbolic(g, input, *, out=None):
    return g.op("Asinh", input)

register_op('asinh', asinh_symbolic, '', 9)
```

这里的 asinh_symbolic 就是 asinh 的符号函数。从除 g 以外的第二个输入参数开始，其输入参数应该严格对应它在 ATen 中的定义：

```
def asinh(input: Tensor, *, out: Optional[Tensor]=None) -> Tensor: ...
```

在符号函数的函数体中，g.op("Asinh", input) 则完成了 ONNX 算子的定义。其中，第一个参数"Asinh" 是算子在 ONNX 中的名称。至于第二个参数 input，如我们刚刚在文档里所见，这个算子只有一个输入，因此我们只要把符号函数的输入参数 input 对应过去就行。ONNX 的 Asinh 的输出和 ATen 的 asinh 的输出是一致的，因此我们直接把 g.op() 的结果返回即可。

定义完符号函数后，我们要把这个符号函数和原来的 ATen 算子“绑定”起来。这里，我们要用到 register_op 这个 PyTorch API 来完成绑定。如示例所示，只需要一行简单的代码即可把符号函数 asinh_symbolic 绑定到算子 asinh 上：

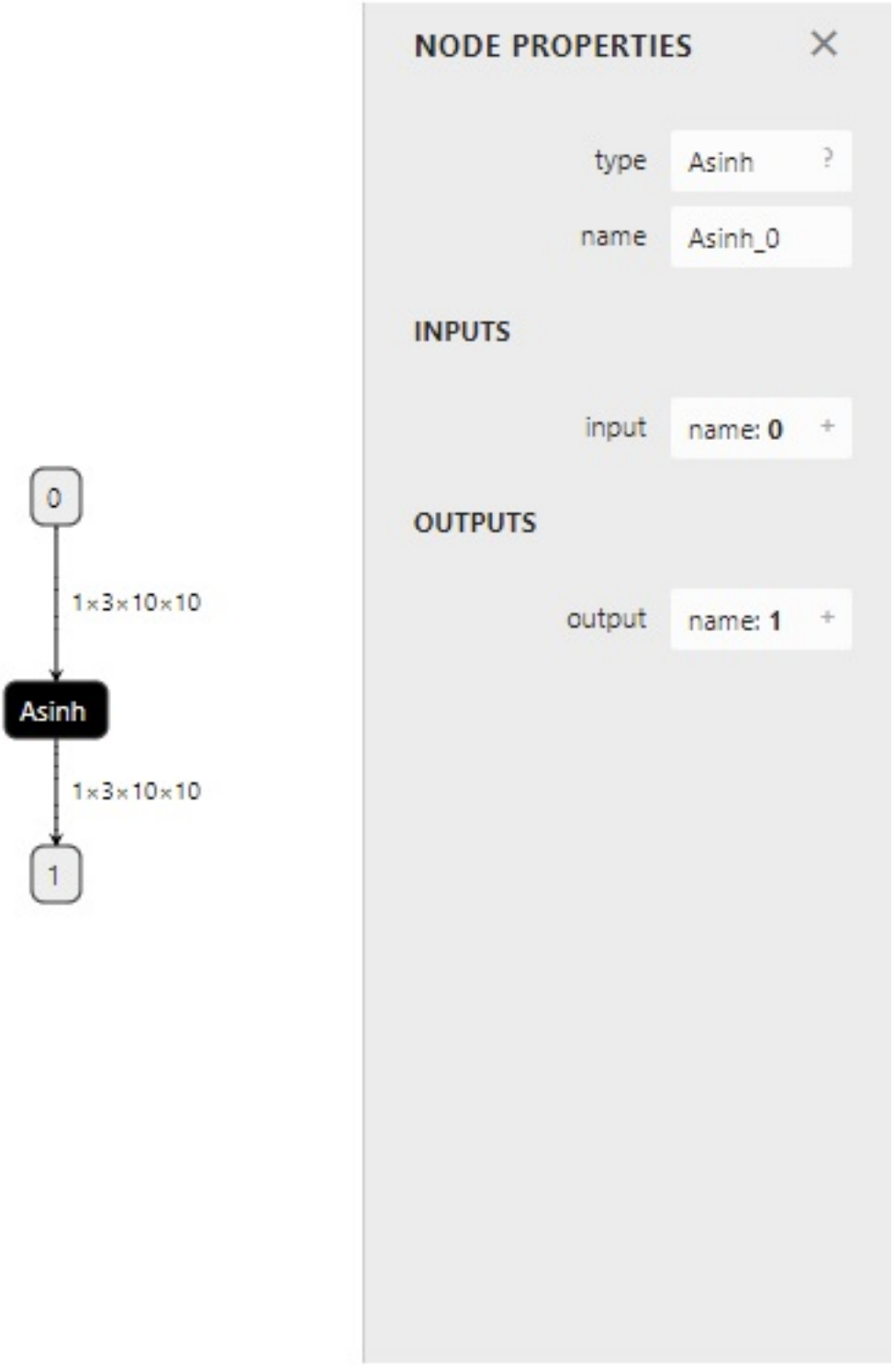
```
register_op('asinh', asinh_symbolic, '', 9)
```

register_op 的第一个参数是目标 ATen 算子名，第二个是要注册的符号函数，这两个参数很好理解。第三个参数是算子的“域”，对于普通 ONNX 算子，直接填空字符串即可。第四个参数表示向哪个算子集版本注册。我们遵照 ONNX 标准，向第 9 号算子集注册。值得注意的是，这里向第 9 号算子集注册，不代表较新的算子集（第 10 号、第 11 号……）都得到了注册。在示例中，我们先只向第 9 号算子集注册。

整理一下，我们最终的代码如下：

```
import torch

class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return torch.asinh(x)

from torch.onnx.symbolic_registry import register_op

def asinh_symbolic(g, input, *, out=None):
    return g.op("Asinh", input)

register_op('asinh', asinh_symbolic, '', 9)

model = Model()
input = torch.rand(1, 3, 10, 10)
torch.onnx.export(model, input, 'asinh.onnx')
```

成功导出的话，asinh.onnx 应该长这个样子：


NODE PROPERTIES
type
Asinh
name
Asinh_0
INPUTS
input
name: 0
OUTPUTS
output
name: 1
0
1×3×10×10
Asinh
1×3×10×10
1

41.1.3 测试算子

在完成了一份自定义算子后，我们一定要测试一下算子的正确性。一般我们要用 PyTorch 运行一遍原算子，再用推理引擎（比如 ONNX Runtime）运行一下 ONNX 算子，最后比对两次的运行结果。对于我们刚刚得到的 `asinh.onnx`，可以用如下代码来验证：

```
import onnxruntime
import torch
import numpy as np

class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return torch.asinh(x)

model = Model()
input = torch.rand(1, 3, 10, 10)
torch_output = model(input).detach().numpy()

sess = onnxruntime.InferenceSession('asinh.onnx')
ort_output = sess.run(None, {'0': input.numpy()})[0]

assert np.allclose(torch_output, ort_output)
```

在这份代码里，我们用 PyTorch 做了一遍推理，并把结果转成了 numpy 格式。之后，我们又用 ONNX Runtime 对 onnx 文件做了一次推理。最后，我们使用 `np.allclose` 来保证两个结果张量的误差在一个可以允许的范围内。一切正常的话，运行这段代码后，`assert` 所在行不会报错，程序应该没有任何输出。

41.2 支持 TorchScript 算子

对于一些比较复杂的运算，仅使用 PyTorch 原生算子是无法实现的。这个时候，就要考虑自定义一个 PyTorch 算子，再把它转换到 ONNX 中了。新增 PyTorch 算子的方法有很多，PyTorch 官方比较推荐的一种做法是添加 TorchScript 算子 。

由于添加算子的方法较繁琐，我们今天跳过新增 TorchScript 算子的内容，以可变形卷积（Deformable Convolution）算子为例，介绍为现有 TorchScript 算子添加 ONNX 支持的方法。

> 可变形卷积（Deformable Convolution）是在 Torchvision 中实现的 TorchScript 算子，虽然尚未得到广泛支持，但是出现在许多模型中。

有了支持 ATen 算子的经验之后，我们可以知道为算子添加符号函数一般要经过以下几步：

1. 获取原算子的前向推理接口。

2. 获取目标 ONNX 算子的定义。

3. 编写符号函数并绑定。

在为可变形卷积添加符号函数时，我们也可以尝试走一遍这个流程。

41.2.1 使用 TorchScript 算子

和之前一样，我们首先定义一个包含了算子的模型，为之后转换 ONNX 模型做准备。

```
import torch
import torchvision

class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(3, 18, 3)
        self.conv2 = torchvision.ops.DeformConv2d(3, 3, 3)

    def forward(self, x):
        return self.conv2(x, self.conv1(x))
```

其中，`torchvision.ops.DeformConv2d` 就是 Torchvision 中的可变形卷积层。相比于普通卷积，可变形卷积的其他参数都大致相同，唯一的区别就是在推理时需要多输入一个表示偏移量的张量。

然后，我们查询算子的前向推理接口。`DeformConv2d` 层最终会调用 `deform_conv2d` 这个算子。我们可以在 `torchvision/csrc/ops/deform_conv2d.cpp` 中查到该算子的调用接口：

```
m.def(TORCH_SELECTIVE_SCHEMA(
      "torchvision::deform_conv2d(Tensor input,
      Tensor weight,
      Tensor offset,
      ......
      bool use_mask) -> Tensor"));
```

那么接下来，根据之前的经验，我们就是要去 ONNX 官方文档中查找算子的定义了。

41.2.2 自定义 ONNX 算子

很遗憾的是，如果我们去 ONNX 的官方算子页面搜索”deform”，将搜不出任何内容。目前，ONNX 还没有提供可变形卷积的算子，我们要自己定义一个 ONNX 算子了。

我们在前面讲过，`g.op()` 是用来定义 ONNX 算子的函数。对于 ONNX 官方定义的算子，`g.op()` 的第一个参数就是该算子的名称。而对于一个自定义算子，`g.op()` 的第一个参数是一个带命名空间的算子名，比如：

```
g.op("custom::deform_conv2d, ...)
```

其中，”::” 前面的内容就是我们的命名空间。该概念和 C++ 的命名空间类似，是为了防止命名冲突而设定的。如果在 `g.op()` 里不加前面的命名空间，则算子会被默认成 ONNX 的官方算子。

PyTorch 在运行 `g.op()` 时会对官方的算子做检查，如果算子名有误，或者算子的输入类型不正确，`g.op()` 就会报错。为了让我们随心所欲地定义新 ONNX 算子，我们必须设定一个命名空间，给算子取个名，再定义自己的算子。

我们在第一篇教程学过：ONNX 是一套标准，本身并不包括实现。在这里，我们就简略地定义一个 ONNX 可变形卷积算子，而不去写它在某个推理引擎上的实现。在之后的教程中，我们再学习在各个推理引擎中添加新 ONNX 算子支持的方法。此处，我们只关心如何导出一个包含新 ONNX 算子节点的 onnx 文件。因此，我们可以为新算子编写如下简单的符号函数：

```
@parse_args("v", "v", "v", "v", "v", "i", "i", "i", "i", "i", "i", "i", "i", "none")
def symbolic(g,
        input,
        weight,
        offset,
        mask,
        bias,
        stride_h, stride_w,
        pad_h, pad_w,
        dil_h, dil_w,
        n_weight_grps,
        n_offset_grps,
        use_mask):
    return g.op("custom::deform_conv2d", input, offset)
```

在这个符号函数中，我们以刚刚搜索到的算子输入参数作为符号函数的输入参数，并只用 `input` 和 `offset` 来构造一个简单的 ONNX 算子。

这段代码中，最令人疑惑的就是装饰器 `@parse_args` 了。简单来说，TorchScript 算子的符号函数要求标注出每一个输入参数的类型。比如”v” 表示 Torch 库里的 `value` 类型，一般用于标注张量，而”i” 表示 int 类型，”f” 表示 float 类型，”none” 表示该参数为空。具体的类型含义可以在 torch.onnx.symbolic_helper.py中查看。这里输入参数中的 `input, weight, offset, mask, bias` 都是张量，所以用”v” 表示。后面的其他参数同理。我们不必纠结于 `@parse_args` 的原理，根据实际情况对符号函数的参数标注类型即可。

有了符号函数后，我们通过如下的方式注册符号函数：

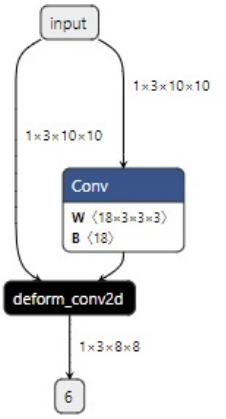
```
register_custom_op_symbolic("torchvision::deform_conv2d", symbolic, 9)
```

和前面的 `register_op` 类似，注册符号函数时，我们要输入算子名、符号函数、算子集版本。与前面不同的是，这里的算子集版本是最早生效版本，在这里设定版本 9，意味着之后的第 10 号、第 11 号……版本集都能使用这个新算子。

最后，我们完整的模型导出代码如下：

```
import torch
import torchvision

class Model(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(3, 18, 3)
        self.conv2 = torchvision.ops.DeformConv2d(3, 3, 3)

    def forward(self, x):
        return self.conv2(x, self.conv1(x))

from torch.onnx import register_custom_op_symbolic
from torch.onnx.symbolic_helper import parse_args

@parse_args("v", "v", "v", "v", "v", "i", "i", "i", "i", "i", "i", "i", "i", "none")
def symbolic(g,
        input,
        weight,
        offset,
        mask,
        bias,
        stride_h, stride_w,
        pad_h, pad_w,
        dil_h, dil_w,
        n_weight_grps,
        n_offset_grps,
        use_mask):
    return g.op("custom::deform_conv2d", input, offset)

register_custom_op_symbolic("torchvision::deform_conv2d", symbolic, 9)

model = Model()
input = torch.rand(1, 3, 10, 10)
torch.onnx.export(model, input, 'dcn.onnx')
```

代码成功运行的话，我们应该能得到如下的 ONNX 模型：



可以看到，我们自定义的 ONNX 算子 `deform_conv2d` 包含了两个输入，一个输出，和我们预想得一样。

41.3 使用 torch.autograd.Function

最后，我们来学习一种简单的为 PyTorch 添加 C++ 算子实现的方法，来代替较为复杂的新增 TorchScript 算子。同时，我们会用 torch.autograd.Function 封装这个新算子。torch.autograd.Function 能完成算子实现和算子调用的隔离。不管算子是怎么实现的，它封装后的使用体验以及 ONNX 导出方法会和原生的 PyTorch 算子一样。这是我们比较推荐的为算子添加 ONNX 支持的方法。

为了应对更复杂的情况，我们来自定义一个奇怪的 `my_add` 算子。这个算子的输入张量 a, b ，输出 `2a + b` 的值。我们会先把它在 PyTorch 中实现，再把它导出到 ONNX 中。

41.3.1 为 PyTorch 添加 C++ 拓展

为 PyTorch 添加简单的 C++ 拓展还是很方便的。对于我们定义的 my_add 算子，可以用以下的 C++ 源文件来实现。我们把该文件命名为”my_add.cpp”：

```
// my_add.cpp

#include <torch/torch.h>

torch::Tensor my_add(torch::Tensor a, torch::Tensor b)
{
    return 2 * a + b;
}

PYBIND11_MODULE(my_lib, m)
{
    m.def("my_add", my_add);
}
```

由于在 PyTorch 中添加 C++ 拓展和模型部署关系不大，这里我们仅给出这个简单的示例，并不对其原理做过多讲解。

在这段代码中，torch::Tensor 就是 C++ 中 torch 的张量类型，它的加法和乘法等运算符均已重载。因此，我们可以像对普通标量一样对张量做加法和乘法。

轻松地完成了算子的实现后，我们用 `PYBIND11_MODULE` 来为 C++ 函数提供 Python 调用接口。这里的 `my_lib` 是我们未来要在 Python 里导入的模块名。双引号中的 `my_add` 是 Python 调用接口的名称，这里我们对齐 C++ 函数的名称，依然用”my_add” 这个名字。

之后，我们可以编写如下的 Python 代码并命名为”setup.py”，来编译刚刚的 C++ 文件：

```
from setuptools import setup
from torch.utils import cpp_extension

setup(name='my_add',
      ext_modules=[cpp_extension.CppExtension('my_lib', ['my_add.cpp'])],
      cmdclass={'build_ext': cpp_extension.BuildExtension})
```

这段代码使用了 Python 的 setuptools 编译功能和 PyTorch 的 C++ 拓展工具函数，可以编译包含了 torch 库的 C++ 源文件。这里我们需要填写的只有模块名和模块中的源文件名。我们刚刚把模块命名为 `my_lib`，而源文件只有一个 `my_add.cpp`，因此拓展模块那一行要写成 `ext_modules=[cpp_extension.CppExtension('my_lib', ['my_add.cpp'])],`。

之后，像处理普通的 Python 包一样执行安装命令，我们的 C++ 代码就会自动编译了。

```
python setup.py develop
```

41.3.2 用 torch.autograd.Function 封装

直接用 Python 接口调用 C++ 函数不太“美观”，一种比较优雅的做法是把这个调用接口封装起来。这里我们用 torch.autograd.Function 来封装算子的底层调用：

```
import torch
import my_lib
class MyAddFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, a, b):
        return my_lib.my_add(a, b)

    @staticmethod
    def symbolic(g, a, b):
        two = g.op("Constant", value_t=torch.tensor([2]))
        a = g.op('Mul', a, two)
        return g.op('Add', a, b)
```

我们在前面的教程中已经见过 torch.autograd.Function，这里我们正式地对其做一个介绍。Function 类本身表示 PyTorch 的一个可导函数，只要为其定义了前向推理和反向传播的实现，我们就可以把它当成一个普通 PyTorch 函数来使用。

PyTorch 会自动调度该函数，合适地执行前向和反向计算。对模型部署来说，Function 类有一个很好的性质：如果它定义了 symbolic 静态方法，该 Function 在执行 torch.onnx.export() 时就可以根据 symbolic 中定义的规则转换成 ONNX 算子。这个 symbolic 就是前面提到的符号函数，只是它的名称必须是 symbolic 而已。

在 forward 函数中，我们用 my_lib.my_add(a, b) 就可以调用之前写的 C++ 函数了。这里 my_lib 是库名，my_add 是函数名，这两个名字是在前面 C++ 的 PYBIND11_MODULE 中定义的。

在 symbolic 函数中，我们用 g.op() 定义了三个算子：常量、乘法、加法。这里乘法和加法的用法和前面提到的 asinh 一样，只需要根据 ONNX 算子定义规则把输入参数填入即可。而在定义常量算子时，我们要把 PyTorch 张量的值传入 value_t 参数中。

在 ONNX 中，我们需要把新建常量当成一个算子来看待，尽管这个算子并不会以节点的形式出现在 ONNX 模型的可视化结果里。

把算子封装成 Function 后，我们可以把 my_add 算子用起来了。

```
my_add = MyAddFunction.apply

class MyAdd(torch.nn.Module):
    def __init__(self):
        super().__init__()
```

(下页继续)

(续上页)

```
    def forward(self, a, b):
        return my_add(a, b)
```

在这份代码里，我们先用 `my_add = MyAddFunction.apply` 获取了一个奇怪的变量。这个变量是用来做什么的呢？其实，`apply` 是 `torch.autograd.Function` 的一个方法，这个方法完成了 `Function` 在前向推理或者反向传播时的调度。我们在使用 `Function` 的派生类做推理时，不应该显式地调用 `forward`，而应该调用其 `apply` 方法。

这里我们使用 `my_add = MyAddFunction.apply` 把这个调用方法取了一个更简短的别名 `my_add`。以后在使用 `my_add` 算子时，我们应该忽略 `MyAddFunction` 的实现细节，而只通过 `my_add` 这个接口来访问算子。这里 `my_add` 的地位，和 PyTorch 的 `asinh, interpolate, conv2d` 等原生函数是类似的。

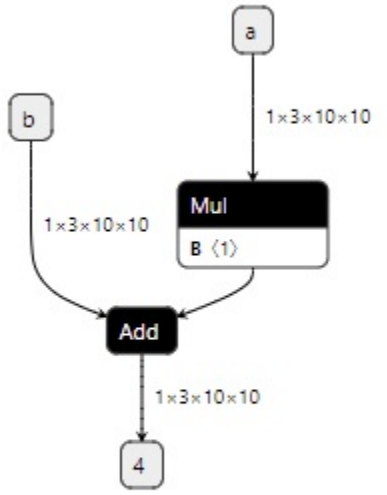
有了访问新算子的接口后，我们可以进一步把算子封装成一个神经网络中的计算层。我们定义一个叫做的 `MyAdd` 的 `torch.nn.Module`，它封装了 `my_add`，就和封装了 `conv2d` 的 `torch.nn.Conv2d` 一样。

41.3.3 测试算子

费了好大的功夫来“包装”我们的新算子后，我们终于可以来使用它了。和之前的测试流程一样，让我们用下面的代码来导出一个包含新算子的 ONNX 模型，并验证一下它是否正确。

```
model = MyAdd()
input = torch.rand(1, 3, 10, 10)
torch.onnx.export(model, (input, input), 'my_add.onnx')
torch_output = model(input, input).detach().numpy()

import onnxruntime
import numpy as np
sess = onnxruntime.InferenceSession('my_add.onnx')
ort_output = sess.run(None, {'a': input.numpy(), 'b': input.numpy()})[0]

assert np.allclose(torch_output, ort_output)
```

在这份代码中，我们直接把 `MyAdd` 作为要导出的模型。我们计算了一个 PyTorch 模型的运行结果，又导出 ONNX 模型，计算了 ONNX 模型在 ONNX Runtime 上的运算结果。如果一切正常的话，这两个结果是一样的，这份代码不会报任何错误，没有任何输出。



可视化一下 my_add.onnx，可以看出，和我们设计得一样，my_add 算子被翻译成了两个 ONNX 算子节点（其中常量算子被放入了 Mul 的参数中）。

整理一下，整个流程的 Python 代码如下：

```
import torch
import my_lib
class MyAddFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, a, b):
        return my_lib.my_add(a, b)

    @staticmethod
    def symbolic(g, a, b):
        two = g.op("Constant", value_t=torch.tensor([2]))
        a = g.op('Mul', a, two)
        return g.op('Add', a, b)

my_add = MyAddFunction.apply

class MyAdd(torch.nn.Module):
    def __init__(self):
        super().__init__()
```

(下页继续)

(续上页)

```
    def forward(self, a, b):
        return my_add(a, b)

model = MyAdd()
input = torch.rand(1, 3, 10, 10)
torch.onnx.export(model, (input, input), 'my_add.onnx')
torch_output = model(input, input).detach().numpy()

import onnxruntime
import numpy as np
sess = onnxruntime.InferenceSession('my_add.onnx')
ort_output = sess.run(None, {'a': input.numpy(), 'b': input.numpy()})[0]

assert np.allclose(torch_output, ort_output)
```

41.4 总结

在这篇教程中，我们围绕“为 ATen 算子添加符号函数”、“为 TorchScript 算子添加符号函数”、“封装成 `torch.autograd.Function` 并添加符号函数”这三种添加映射关系的方法，讲解了 3 个为 PyTorch 和 ONNX 添加支持的实例。在这个过程中，我们学到了很多零散的知识，来总结一下吧。

- ATen 是 PyTorch 的 C++ 张量运算库。通过查询 torch/_C/_VariableFunctions.pyi 和 torch/nn/functional.pyi，我们可以知道 ATen 算子的 Python 接口定义。
- 用 register_op 可以为 ATen 算子补充注册符号函数
- 用 register_custom_op_symbolic 可以为 TorchScript 算子补充注册符号函数
- 如何在 PyTorch 里添加 C++ 拓展
- 如何用 torch.autograd.Function 封装一个自定义 PyTorch 算子
- 如何编写符号函数 symbolic(g, ...)。
- 如何用 g.op() 把一个 PyTorch 算子映射成一个或多个 ONNX 算子，或者是自定义的 ONNX 算子。

这篇教程涉及的代码比较多。如果大家在阅读时碰到了问题，最好去跑一跑代码，改一改代码里的内容，实际感受一下每行代码的意义。

41.5 上期习题解答

1. PyTorch 目前没有支持 ONNX 的 `Asinh` 算子。我们在 `torch.onnx.symbolic_opset9.py` 中搜索不到 Asinh 的相关内容。

2. 通过在 `torch.onnx.symbolic_opset11.py` 搜索 `BitShift`，我们可以发现 PyTorch 在 `__lshift_` 和 `__rshift_` 里用到了 ONNX 的 `BitShift` 算子。当输入类型为 `Byte` 时，PyTorch 会把算子直接翻译翻译 `BitShift`，以代替乘除 2 的次幂的操作。

3. 对应 `Resize` 算子的第 3 个参数（`g.op()` 的第 4 个参数）`scales`。原来的 `scales` 传入 `g.op()` 前会经过 `_interpolate_get_scales_if_available()` 函数，一定会被转换成一个常量。为了让 `scales` 由输入决定，我们直接把输入参数中的 `scales` 传入 `g.op()`。

CHAPTER 42

第五章：ONNX 模型的修改与调试

在前两期教程中，我们学习了 PyTorch 模型转 ONNX 模型的方法，了解了如何在原生算子表达能力不足时，为 PyTorch 或 ONNX 自定义算子。一直以来，我们都是通过 PyTorch 来导出 ONNX 模型的，基本没有单独探究过 ONNX 模型的构造知识。

不知道大家会不会有这样一些疑问：ONNX 模型在底层是用什么格式存储的？如何不依赖深度学习框架，只用 ONNX 的 API 来构造一个 ONNX 模型？如果没有源代码，只有一个 ONNX 模型，该如何对这个模型进行调试？这篇教程可以解答大家的这些问题。

在这期教程里，我们将围绕 ONNX 这一套神经网络定义标准本身，探究 ONNX 模型的构造、读取、子模型提取、调试。首先，我们会学习 ONNX 的底层表示方式。之后，我们会用 ONNX API 构造和读取模型。最后，我们会利用 ONNX 提供的子模型提取功能，学习如何调试 ONNX 模型。

42.1 ONNX 的底层实现

42.1.1 ONNX 的存储格式

ONNX 在底层是用 **Protobuf** 定义的。Protobuf，全称 Protocol Buffer，是 Google 提出的一套表示和序列化数据的机制。使用 Protobuf 时，用户需要先写一份数据定义文件，再根据这份定义文件把数据存储进一份二进制文件。可以说，数据定义文件就是数据类，二进制文件就是数据类的实例。这里给出一个 Protobuf 数据定义文件的例子：

```
message Person {
  required string name = 1;
```

(下页继续)

(续上页)

```
  required int32 id = 2;
  optional string email = 3;
}
```

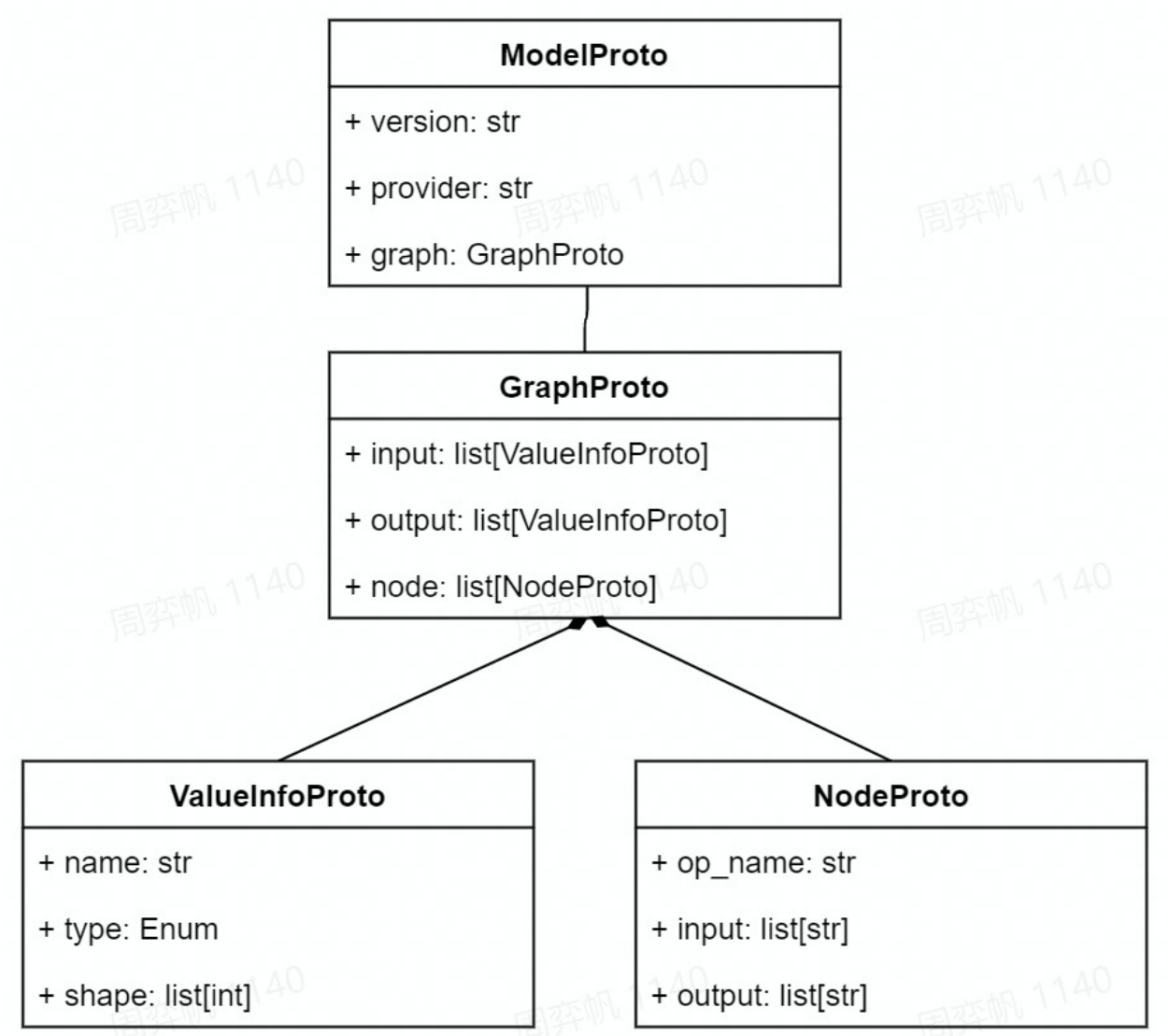
这段定义表示在 Person 这种数据类型中，必须包含 name、id 这两个字段，选择性包含 email 字段。根据这份定义文件，用户就可以选择一种编程语言，定义一个含有成员变量 name、id、email 的 Person 类，把这个类的某个实例用 Protobuf 存储成二进制文件；反之，用户也可以用二进制文件和对应的数据定义文件，读取出一个 Person 类的实例。

而对于 ONNX，它的 Protobuf 数据定义文件在其开源库中，这些文件定义了神经网络中模型、节点、张量的数据类型规范；而数据定义文件对应的二进制文件就是我们熟悉的 ".onnx" 文件，每一个".onnx" 文件按照数据定义规范，存储了一个神经网络的所有相关数据。直接用 Protobuf 生成 ONNX 模型还是比较麻烦的。幸运的是，ONNX 提供了很多实用 API，我们可以在完全不了解 Protobuf 的前提下，构造和读取 ONNX 模型。

42.1.2 ONNX 的结构定义

在用 API 对 ONNX 模型进行操作之前，我们还需要先了解一下 ONNX 的结构定义规则，学习一下 ONNX 在 Protobuf 定义文件里是怎样描述一个神经网络的。

回想一下，神经网络本质上是一个计算图。计算图的节点是算子，边是参与运算的张量。而通过可视化 ONNX 模型，我们知道 ONNX 记录了所有算子节点的属性信息，并把参与运算的张量信息存储在算子节点的输入输出信息中。事实上，ONNX 模型的结构可以用类图大致表示如下：



如图所示，一个 ONNX 模型可以用 `ModelProto` 类表示。`ModelProto` 包含了版本、创建者等日志信息，还包含了存储计算图结构的 `graph`。`GraphProto` 类则由输入张量信息、输出张量信息、节点信息组成。张量信息 `ValueInfoProto` 类包括张量名、基本数据类型、形状。节点信息 `NodeProto` 类包含了算子名、算子输入张量名、算子输出张量名。让我们来看一个具体的例子。假如我们有一个描述 `output=a*x+b` 的 ONNX 模型 `model`，用 `print(model)` 可以输出以下内容：

```
ir_version: 8
graph {
  node {
    input: "a"
    input: "x"
    output: "c"
    op_type: "Mul"
  }
```

(下页继续)

(续上页)

```
  node {
    input: "c"
    input: "b"
    output: "output"
    op_type: "Add"
  }
  name: "linear_func"
  input {
    name: "a"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {dim_value: 10}
          dim {dim_value: 10}
        }
      }
    }
  }
  input {
    name: "x"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {dim_value: 10}
          dim {dim_value: 10}
        }
      }
    }
  }
  input {
    name: "b"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {dim_value: 10}
          dim {dim_value: 10}
        }
      }
    }
  }
```

(下页继续)

(续上页)

```
  output {
    name: "output"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim { dim_value: 10}
          dim { dim_value: 10}
        }
      }
    }
  }
}
opset_import {version: 15}
```

对应上文中的类图，这个模型的信息由 `ir_version`，`opset_import` 等全局信息和 `graph` 图信息组成。而 `graph` 包含一个乘法节点、一个加法节点、三个输入张量 `a`，`x`，`b` 以及一个输出张量 `output`。在下一节里，我们会用 API 构造出这个模型，并输出这段结果。

42.2 读写 ONNX 模型

42.2.1 构造 ONNX 模型

在上一小节中，我们知道了 ONNX 模型是按以下的结构组织起来的：

- ModelProto
  - GraphProto
    - NodeProto
    - ValueInfoProto

现在，让我们抛开 PyTorch，尝试完全用 ONNX 的 Python API 构造一个描述线性函数 `output=a*x+b` 的 ONNX 模型。我们将根据上面的结构，自底向上地构造这个模型。

首先，我们可以用 `helper.make_tensor_value_info` 构造出一个描述张量信息的 `ValueInfoProto` 对象。如前面的类图所示，我们要传入张量名、张量的基本数据类型、张量形状这三个信息。在 ONNX 中，不管是输入张量还是输出张量，它们的表示方式都是一样的。因此，这里我们用类似的方式为三个输入 `a`，`x`，`b` 和一个输出 `output` 构造 `ValueInfoProto` 对象。如下面的代码所示：

```
import onnx
from onnx import helper
from onnx import TensorProto
```

(下页继续)

(续上页)

```
a = helper.make_tensor_value_info('a', TensorProto.FLOAT, [10, 10])
x = helper.make_tensor_value_info('x', TensorProto.FLOAT, [10, 10])
b = helper.make_tensor_value_info('b', TensorProto.FLOAT, [10, 10])
output = helper.make_tensor_value_info('output', TensorProto.FLOAT, [10, 10])
```

之后，我们要构造算子节点信息 NodeProto，这可以通过在 helper.make_node 中传入算子类型、输入张量名、输出张量名这三个信息来实现。我们这里先构造了描述 c=a*x 的乘法节点，再构造了 output=c+b 的加法节点。如下面的代码所示：

```
mul = helper.make_node('Mul', ['a', 'x'], ['c'])
add = helper.make_node('Add', ['c', 'b'], ['output'])
```

在计算机中，图一般是用一个节点集和一个边集表示的。而 ONNX 巧妙地把边的信息保存在了节点信息里，省去了保存边集的步骤。在 ONNX 中，如果某节点的输入名和之前某节点的输出名相同，就默认这两个节点是相连的。如上面的例子所示：Mul 节点定义了输出 c，Add 节点定义了输入 c，则 Mul 节点和 Add 节点是相连的。

正是因为有这种边的隐式定义规则，所以 ONNX 对节点的输入有一定的要求：一个节点的输入，要么是整个模型的输入，要么是之前某个节点的输出。如果我们把 a，x，b 中的某个输入节点从计算图中拿出（这个操作会在之后的代码中介绍），或者把 Mul 的输出从 c 改成 d，则最终的 ONNX 模型都是不满足标准的。

> 一个不满足标准的 ONNX 模型可能无法被推理引擎正确识别。ONNX 提供了 API onnx.checker.check_model 来判断一个 ONNX 模型是否满足标准。

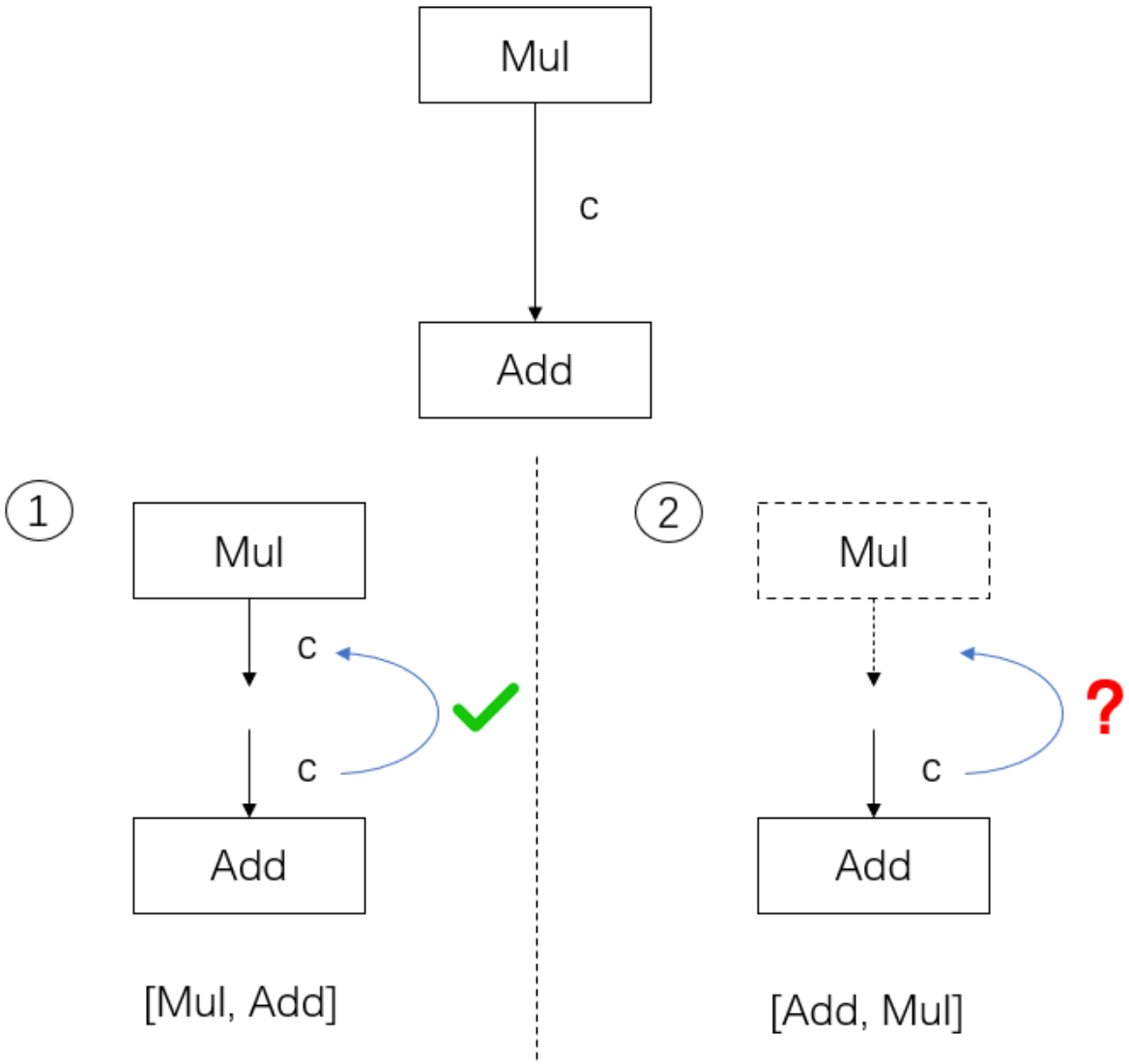
接下来，我们用 helper.make_graph 来构造计算图 GraphProto。helper.make_graph 函数需要传入节点、图名称、输入张量信息、输出张量信息这 4 个参数。如下面的代码所示，我们把之前构造出来的 NodeProto 对象和 ValueInfoProto 对象按照顺序传入即可。

```
graph = helper.make_graph([mul, add], 'linear_func', [a, x, b], [output])
```

这里 make_graph 的节点参数有一个要求：计算图的节点必须以拓扑序给出。

> 拓扑序是与有向图的相关的数学概念。如果按拓扑序遍历所有节点的话，能保证每个节点的输入都能在之前节点的输出里找到（对于 ONNX 模型，我们把计算图的输入张量也看成“之前的输出”）。

如果对这个概念不熟也没有关系，我们以刚刚构造出来的这个计算图为研究对象，通过下图展示的两个例子来直观理解拓扑序。



这里我们只关注 Mul 和 Add 节点以及它们之间的边 c。在情况 1 中：如果我们的节点以 [Mul, Add] 顺序给出，那么遍历到 Add 时，它的输入 c 可以在之前的 Mul 的输出中找到。但是，如情况 2 所示：如果我们的节点以 [Add, Mul] 的顺序给出，那么 Add 就找不到输入边，计算图也无法成功构造出来了。这里的 [Mul, Add] 就是符合有向图的拓扑序的，而 [Add, Mul] 则不满足。

最后，我们用 helper.make_model 把计算图 GraphProto 封装进模型 ModelProto 里，一个 ONNX 模型就构造完成了。make_model 函数中还可以添加模型制作者、版本等信息，为了简单起见，我们没有添加额外的信息。如下面的代码所示：

```
model = helper.make_model(graph)
```

构造完模型之后，我们用下面这三行代码来检查模型正确性、把模型以文本形式输出、存储到一个".onnx" 文件里。这里用 onnx.checker.check_model 来检查模型是否满足 ONNX 标准是必要的，因为无论模型是否满足标准，ONNX 都允许我们用 onnx.save 存储模型。我们肯定不希望生成一个不满足标准的模型。

```
onnx.checker.check_model(model)
print(model)
onnx.save(model, 'linear_func.onnx')
```

成功执行这些代码的话，程序会以文本格式输出模型的信息，其内容应该和我们在上一节展示的输出一样。

整理一下，用 ONNX Python API 构造模型的代码如下：

```
import onnx
from onnx import helper
from onnx import TensorProto

# input and output
a = helper.make_tensor_value_info('a', TensorProto.FLOAT, [10, 10])
x = helper.make_tensor_value_info('x', TensorProto.FLOAT, [10, 10])
b = helper.make_tensor_value_info('b', TensorProto.FLOAT, [10, 10])
output = helper.make_tensor_value_info('output', TensorProto.FLOAT, [10, 10])

# Mul
mul = helper.make_node('Mul', ['a', 'x'], ['c'])

# Add
add = helper.make_node('Add', ['c', 'b'], ['output'])

# graph and model
graph = helper.make_graph([mul, add], 'linear_func', [a, x, b], [output])
model = helper.make_model(graph)

# save model
onnx.checker.check_model(model)
print(model)
onnx.save(model, 'linear_func.onnx')
```

老规矩，我们可以用 ONNX Runtime 运行模型，来看看模型是否正确：

```
import onnxruntime
import numpy as np

sess = onnxruntime.InferenceSession('linear_func.onnx')
a = np.random.rand(10, 10).astype(np.float32)
b = np.random.rand(10, 10).astype(np.float32)
x = np.random.rand(10, 10).astype(np.float32)

output = sess.run(['output'], {'a': a, 'b': b, 'x': x})[0]
```

(下页继续)

(续上页)

```
assert np.allclose(output, a * x + b)
```

一切顺利的话，这段代码不会有任何报错信息。这说明我们的模型等价于执行 a * x + b 这个计算。

42.2.2 读取并修改 ONNX 模型

通过用 API 构造 ONNX 模型，我们已经彻底搞懂了 ONNX 由哪些模块组成。现在，让我们看看该如何读取现有的".onnx" 文件并从中提取模型信息。

首先，我们可以用下面的代码读取一个 ONNX 模型：

```
import onnx
model = onnx.load('linear_func.onnx')
print(model)
```

之前在输出模型时，我们传给 `onnx.save` 的是一个 `ModelProto` 的对象。同理，用上面的 `onnx.load` 读取 ONNX 模型时，我们收获的也是一个 `ModelProto` 的对象。输出这个对象后，我们应该得到和之前完全相同的输出。接下来，我们来看看怎么把图 `GraphProto`、节点 `NodeProto`、张量信息 `ValueInfoProto` 读取出来：

```
graph = model.graph
node = graph.node
input = graph.input
output = graph.output
print(node)
print(input)
print(output)
```

使用如上这些代码，我们可以分别访问模型的图、节点、张量信息。这里大家或许会有疑问：该怎样找出 `graph.node`,`graph.input` 中 `node, input` 这些属性名称呢？其实，属性的名称就写在每个对象的输出里。我们以 `print(node)` 的输出为例：

```
[input: "a"
input: "x"
output: "c"
op_type: "Mul"
, input: "c"
input: "b"
output: "output"
op_type: "Add"
]
```

在这段输出中，我们能看出 `node` 其实就是一个列表，列表中的对象有属性 `input, output, op_type`（这里 `input` 也是一个列表，它包含的两个元素都显示出来了）。我们可以用下面的代码来获取 `node` 里第一

个节点 Mul 的属性：

```
node_0 = node[0]
node_0_inputs = node_0.input
node_0_outputs = node_0.output
input_0 = node_0_inputs[0]
input_1 = node_0_inputs[1]
output = node_0_outputs[0]
op_type = node_0.op_type

print(input_0)
print(input_1)
print(output)
print(op_type)

# Output
"""
a
x
c
Mul
"""
```

当我们想知道 ONNX 模型某数据对象有哪些属性时，我们不必去翻 ONNX 文档，只需要先把数据对象输出一下，然后在输出结果找出属性名即可。

读取完 ONNX 模型的信息后，修改 ONNX 模型就是一件很轻松的事了。我们既可以按照上一小节的模型构造方法，新建节点和张量信息，与原有模型组合成一个新的模型，也可以在不违反 ONNX 规范的前提下直接修改某个数据对象的属性。

这里我们来看一个直接修改模型属性的例子：

```
import onnx
model = onnx.load('linear_func.onnx')

node = model.graph.node
node[1].op_type = 'Sub'

onnx.checker.check_model(model)
onnx.save(model, 'linear_func_2.onnx')
```

在读入之前的 linear_func.onnx 模型后，我们可以直接修改第二个节点的类型 node[1].op_type，把加法变成减法。这样，我们的模型描述的是 a * x - b 这个线性函数。大家感兴趣的话，可以用 ONNX Runtime 运行新模型 linear_func_2.onnx，来验证一下它和 a * x - b 是否等价。

42.3 调试 ONNX 模型

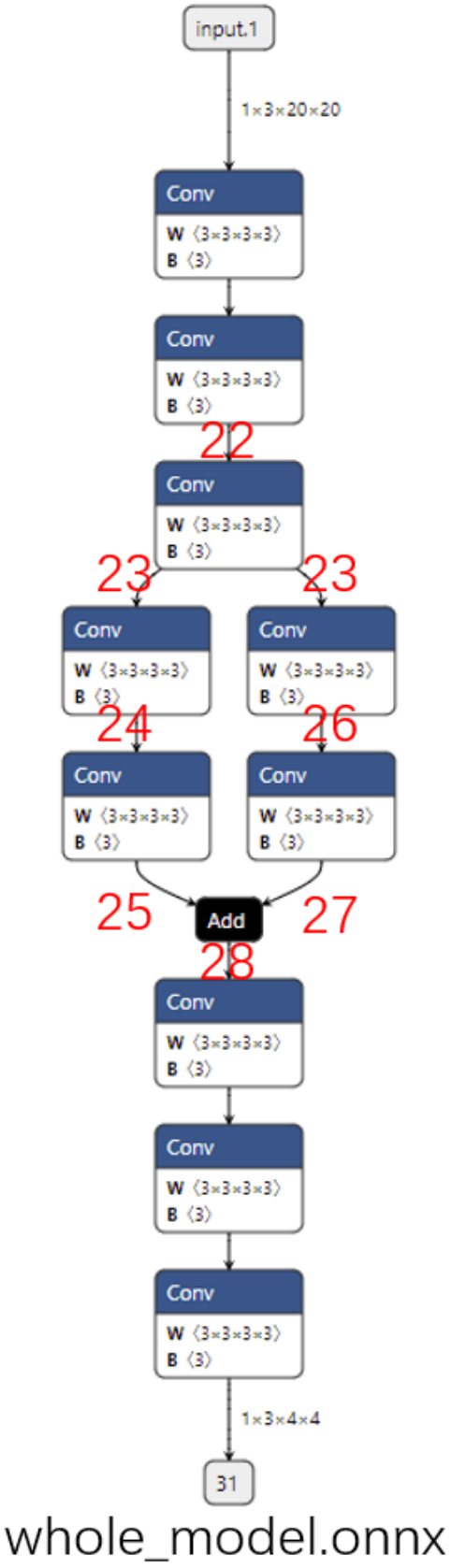
在实际部署中，如果用深度学习框架导出的 ONNX 模型出了问题，一般要通过修改框架的代码来解决，而不会从 ONNX 入手，我们把 ONNX 模型当成一个不可修改的黑盒看待。现在，我们已经深入学习了 ONNX 的原理，可以尝试对 ONNX 模型本身进行调试了。在这一节里，让我们看看该如何巧妙利用 ONNX 提供的子模型提取功能，对 ONNX 模型进行调试。

42.3.1 子模型提取

ONNX 官方为开发者提供了子模型提取（extract）的功能。子模型提取，顾名思义，就是从一个给定的 ONNX 模型中，拿出一个子模型。这个子模型的节点集、边集都是原模型中对应集合的子集。让我们来用 PyTorch 导出一个复杂一点的 ONNX 模型，并在它的基础上执行提取操作：

```
import torch

class Model(torch.nn.Module):

    def __init__(self):
        super().__init__()
        self.convs1 = torch.nn.Sequential(torch.nn.Conv2d(3, 3, 3),
                                          torch.nn.Conv2d(3, 3, 3),
                                          torch.nn.Conv2d(3, 3, 3))
        self.convs2 = torch.nn.Sequential(torch.nn.Conv2d(3, 3, 3),
                                          torch.nn.Conv2d(3, 3, 3))
        self.convs3 = torch.nn.Sequential(torch.nn.Conv2d(3, 3, 3),
                                          torch.nn.Conv2d(3, 3, 3))
        self.convs4 = torch.nn.Sequential(torch.nn.Conv2d(3, 3, 3),
                                          torch.nn.Conv2d(3, 3, 3),
                                          torch.nn.Conv2d(3, 3, 3))
    def forward(self, x):
        x = self.convs1(x)
        x1 = self.convs2(x)
        x2 = self.convs3(x)
        x = x1 + x2
        x = self.convs4(x)
        return x

model = Model()
input = torch.randn(1, 3, 20, 20)

torch.onnx.export(model, input, 'whole_model.onnx')
```

这个模型的可视化结果如下图所示（提取子模型需要输入边的序号，为了大家方面阅读，这幅图标出了之后要用到的边的序号）：

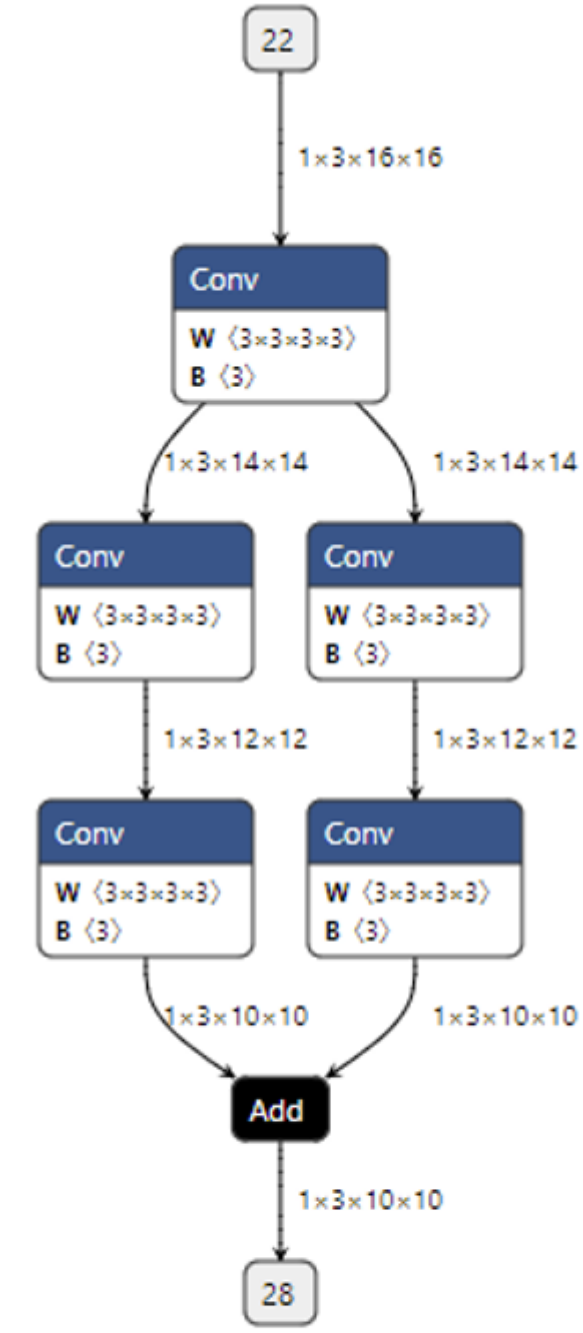

input.1
1×3×20×20
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
22
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
23
23
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
24
26
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
25
Add
27
28
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
1×3×4×4
31
whole_model.onnx

在前面的章节中，我们学过，ONNX 的边用同名张量表示的。也就是说，这里的边序号，实际上是前一个节点的输出张量序号和后一个节点的输入张量序号。由于这个模型是用 PyTorch 导出的，这些张量序号都是 PyTorch 自动生成的。

接着，我们可以下面的代码提取出一个子模型：

```
import onnx

onnx.utils.extract_model('whole_model.onnx', 'partial_model.onnx', ['22'], ['28'])
```

子模型的可视化结果如下图所示：



partial_model.onnx

通过观察代码和输出图，应该不难猜出这段代码的作用是把原计算图从边 22 到边 28 的子图提取出来，并组成一个子模型。`onnx.utils.extract_model` 就是完成子模型提取的函数，它的参数分别是原模型路径、

输出模型路径、子模型的输入边（输入张量）、子模型的输出边（输出张量）。

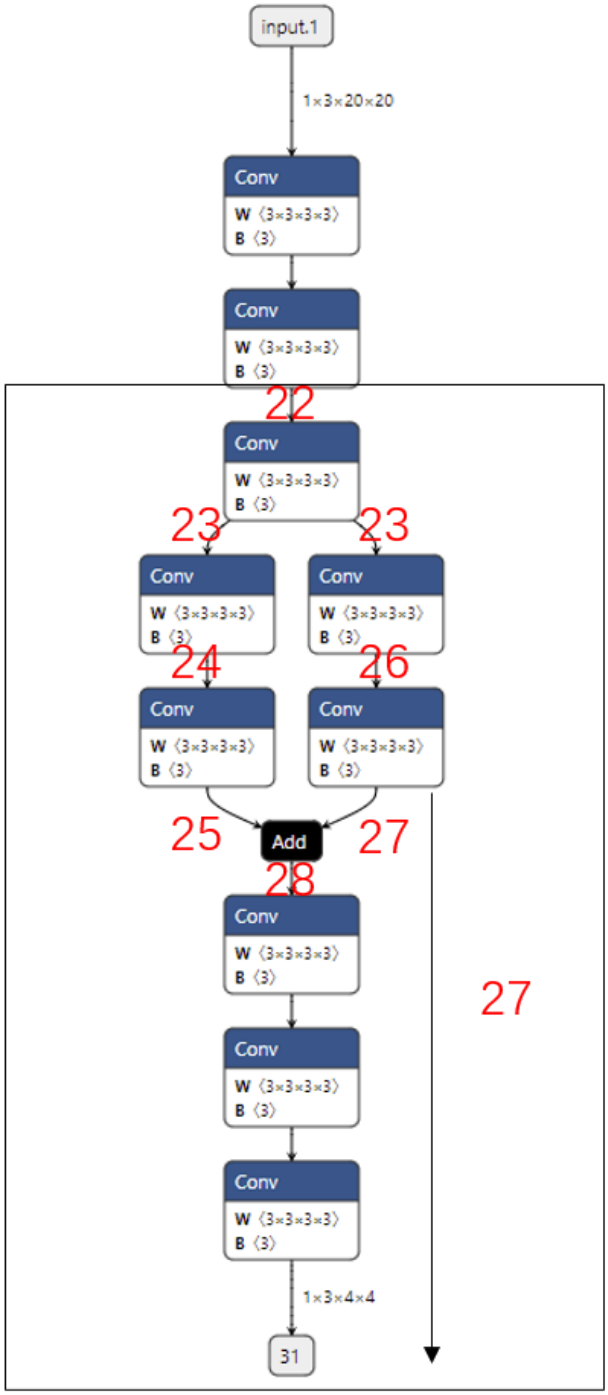
直观地来看，子模型提取就是把输入边到输出边之间的全部节点都取出来。那么，这个功能在使用上有什么限制呢？基于 `whole_model.onnx`, 我们来看一看三个子模型提取的示例。

添加额外输出

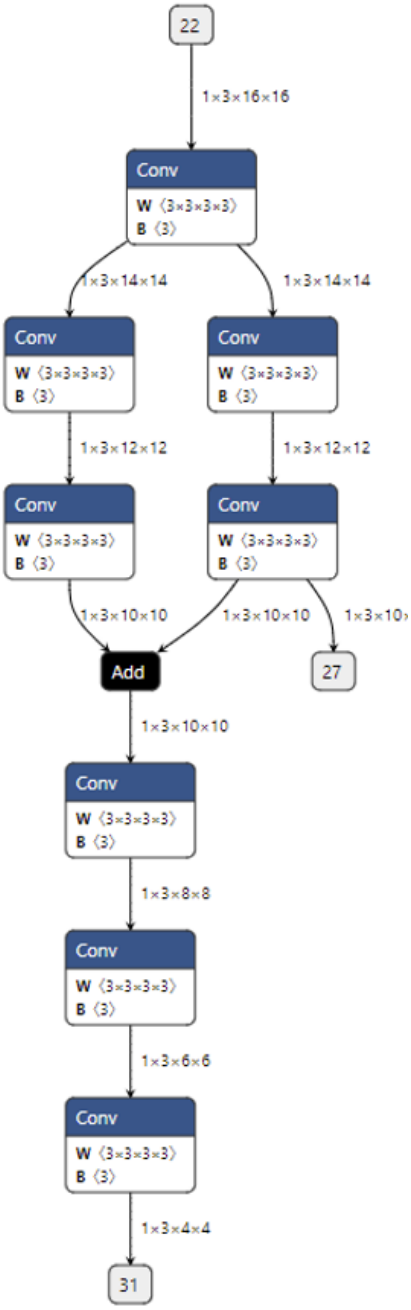
我们在提取时新设定了一个输出张量，如下面的代码所示：

```
onnx.utils.extract_model('whole_model.onnx', 'submodel_1.onnx', ['22'], ['27', '31'])
```

我们可以看到子模型会添加一条把张量输出的新边，如下图所示：



whole_model.onnx



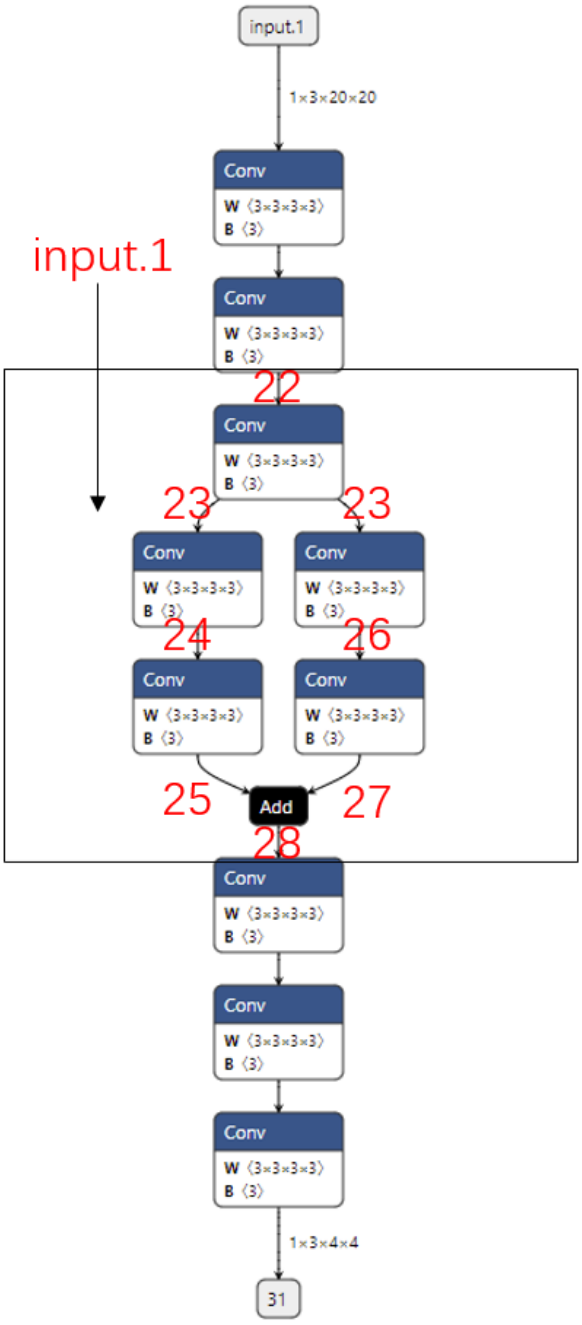
submodel_1.onnx

添加冗余输入

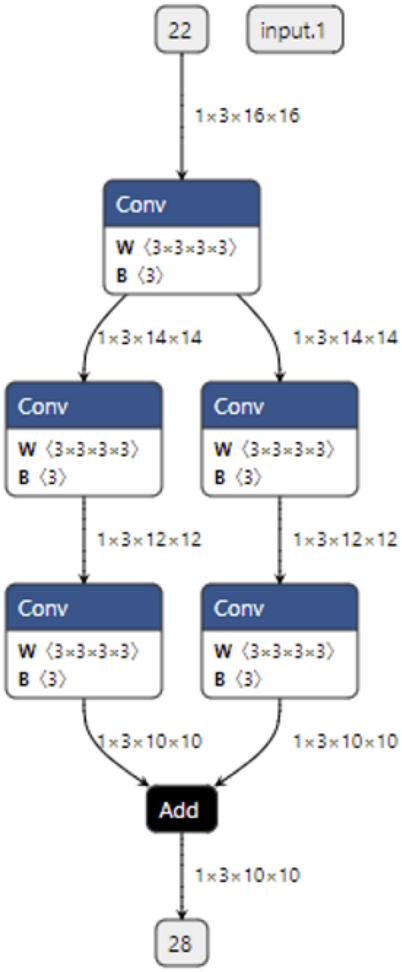
如果我们还是像开始一样提取边 22 到边 28 之间的子模型，但是多添加了一个输入 input.1，那么提取出的子模型会有一个冗余的输入 input.1，如下面的代码所示：

```
onnx.utils.extract_model('whole_model.onnx', 'submodel_2.onnx', ['22', 'input.1'], [
↪'28'])
```

从下图中可以看出：无论给这个输入传入什么值，都不会影响子模型的输出。可以认为如果只用子模型的部分输入就能得到输出，那么那些”较早“的多出来的输入就是冗余的。



whole_model.onnx

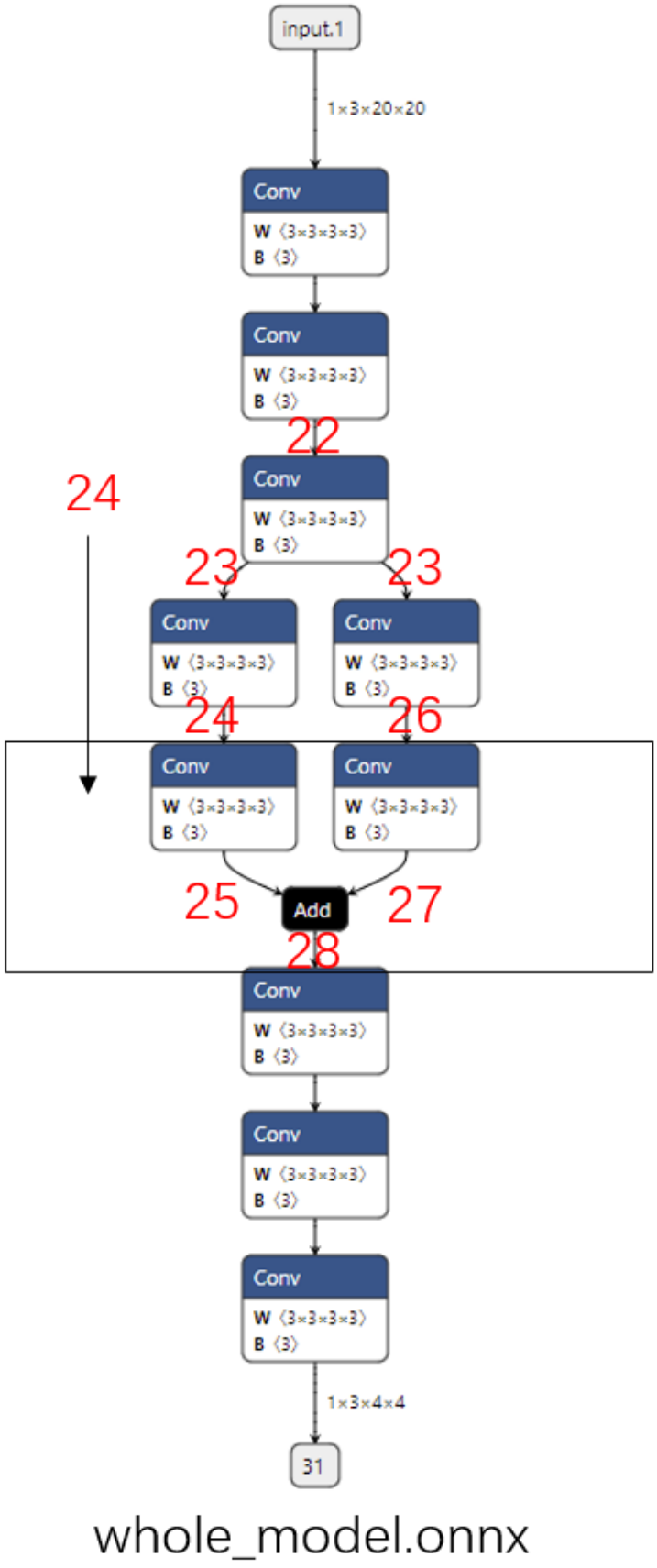


submodel_2.onnx

输入信息不足

这次，我们尝试提取的子模型输入是边 24，输出是边 28。如下面的代码和图所示：

```
# Error
onnx.utils.extract_model('whole_model.onnx', 'submodel_3.onnx', ['24'], ['28'])
```


input.1
1×3×20×20
Conv
W 〈3×3×3×3〉
B 〈3〉
22
24
23
23
24
26
Add
25
27
28
1×3×4×4
31
whole_model.onnx

从图中可以看出，想通过边 24 计算边 28 的结果，至少还需要输入边 26，或者更上面的边。仅凭借边 24 是无法计算出边 28 的结果的，因此这样提取子模型会报错。

通过上面几个使用示例，我们可以整理出子模型提取的实现原理：新建一个模型，把给定的输入和输出填入。之后把图的所有有向边反向，从输出边开始遍历节点，碰到输入边则停止，把这样遍历得到的节点做为子模型的节点。

如果还没有彻底弄懂这个提取原理，没关系，我们只要尽量保证在填写子模型的输入输出时，让输出恰好可以由输入决定即可。

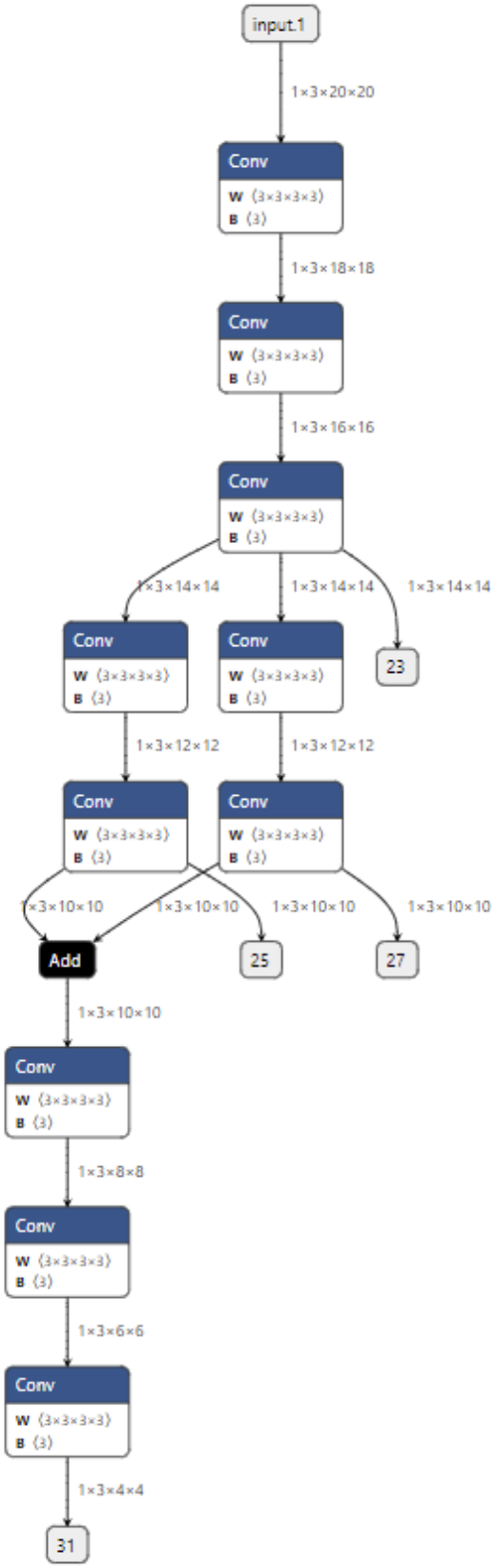
42.3.2 输出 ONNX 中间节点的值

在使用 ONNX 模型时，最常见的一个需求是能够用推理引擎输出中间节点的值。这多见于深度学习框架模型和 ONNX 模型的精度对齐中，因为只要能够输出中间节点的值，就能定位到精度出现偏差的算子。我们来看看如何用子模型提取实现这一任务。

在刚刚的第一个子模型提取示例中，我们添加了一条原来模型中不存在的输出边。用同样的原理，我们可以在保持原有输入输出不变的同时，新增加一些输出，提取出一个能输出中间节点的”子模型“。例如：

```
onnx.utils.extract_model('whole_model.onnx', 'more_output_model.onnx', ['input.1'], [
↪'31', '23', '25', '27'])
```

在这个子模型中，我们在保持原有的输入 `input.1`，输出 `31` 的同时，把其他几个边加入了输出中。如下图所示：

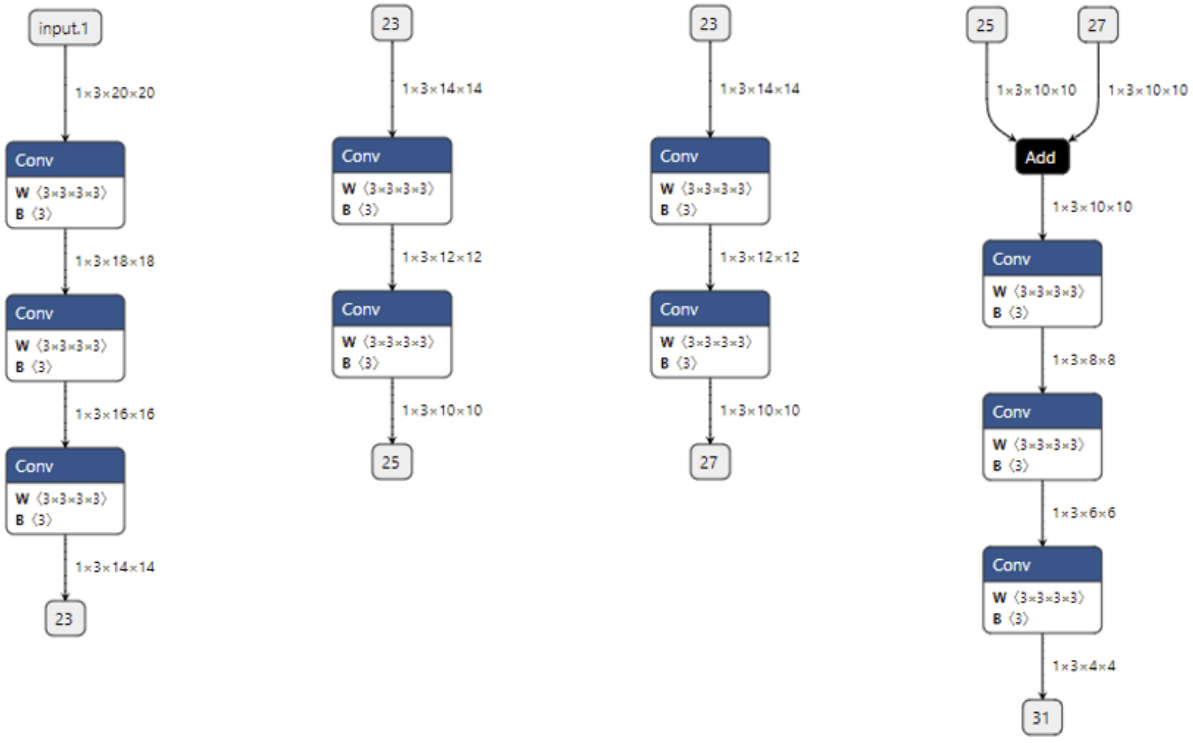

input.1
1×3×20×20
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
1×3×18×18
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
1×3×16×16
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
1×3×14×14
1×3×14×14
1×3×14×14
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
23
1×3×12×12
1×3×12×12
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
×3×10×10
1×3×10×10
1×3×10×10
1×3×10×10
Add
25
27
1×3×10×10
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
1×3×8×8
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
1×3×6×6
Conv
W ⟨3×3×3×3⟩
B ⟨3⟩
1×3×4×4
31

这样，用 ONNX Runtime 运行 `more_output_model.onnx` 这个模型时，我们就能得到更多的输出了。为了方便调试，我们还可以把原模型拆分成多个互不相交的子模型。这样，在每次调试时，可以只对原模型的部分子模块调试。比如：

```
onnx.utils.extract_model('whole_model.onnx', 'debug_model_1.onnx', ['input.1'], ['23
↪'])
onnx.utils.extract_model('whole_model.onnx', 'debug_model_2.onnx', ['23'], ['25'])
onnx.utils.extract_model('whole_model.onnx', 'debug_model_3.onnx', ['23'], ['27'])
onnx.utils.extract_model('whole_model.onnx', 'debug_model_4.onnx', ['25', '27'], ['31
↪'])
```

在这个例子中，我们把原来较为复杂的模型拆成了四个较为简单的子模型，如下图所示。在调试时，我们可以先调试顶层的子模型，确认顶层子模型无误后，把它的输出做为后面子模型的输入。

比如对于这些子模型，我们可以先调试第一个子模型，并存储输出 23。之后把张量 23 做为第二个和第三个子模型的输入，调试这两个模型。最后用同样方法调试第四个子模型。可以说，有了子模型提取功能，哪怕是面对一个庞大的模型，我们也能够从中提取出有问题的子模块，细致地只对这个子模块调试。



子模型提取固然是一个便利的 ONNX 调试工具。但是，在实际的情况中，我们一般是用 PyTorch 等框架导出 ONNX 模型。这里有两个问题：

1. 一旦 PyTorch 模型改变，ONNX 模型的边序号也会改变。这样每次提取同样的子模块时都要重新去 ONNX 模型里查序号，如此繁琐的调试方法是不会在实践中采用的。

2. 即使我们能保证 ONNX 的边序号不发生改变，我们也难以把 PyTorch 代码和 ONNX 节点对应起来——当模型结构变得十分复杂时，要识别 ONNX 中每个节点的含义是不可能的。

MMDeploy 为 PyTorch 模型添加了模型分块功能。使用这个功能，我们可以通过只修改 PyTorch 模型的实现代码来把原模型导出成多个互不相交的子 ONNX 模型。我们会在后续教程中对其介绍。

42.4 总结

在这篇教程中，我们抛开了 PyTorch，学习了 ONNX 模型本身的知识。老规矩，我们来总结一下这篇教程的知识点：

- ONNX 使用 Protobuf 定义规范和序列化模型。
- 一个 ONNX 模型主要由 `ModelProto`,`GraphProto`,`NodeProto`,`ValueInfoProto` 这几个数据类的对象组成。
- 使用 `onnx.helper.make_xxx`，我们可以构造 ONNX 模型的数据对象。
- `onnx.save()` 可以保存模型，`onnx.load()` 可以读取模型，`onnx.checker.check_model()` 可以检查模型是否符合规范。
- `onnx.utils.extract_model()` 可以从原模型中取出部分节点，和新定义的输入、输出边构成一个新的子模型。
- 利用子模型提取功能，我们可以输出原 ONNX 模型的中间结果，实现对 ONNX 模型的调试。

至此，我们对 ONNX 相关知识的学习就告一段落了。回顾一下，我们先学习了 PyTorch 转 ONNX 有关 API 的用法；接着，我们学习了如何用自定义算子解决 PyTorch 和 ONNX 表达能力不足的问题；最后我们单独学习了 ONNX 模型的调试方法。通过对 ONNX 由浅入深的学习，我们基本可以应对模型部署中和 ONNX 有关的绝大多数问题了。

如果大家想了解更多有关 ONNX API 的知识，可以去阅读 ONNX 的官方 Python API 文档。

CHAPTER 43

Ubuntu18.04 交叉编译 NDK snpe 推理服务

mmdeploy 已提供预编译包，如果你想自己编译、或需要对.proto 接口做修改，可参考此文档。

注意 gRPC 官方文档并没有对 NDK 的完整支持。

43.1 一、环境说明

43.2 二、NDK 交叉编译 gRPC

1. 拉取 gRPC repo, 在 host 上编译出 `protoc` 和 `grpc_cpp_plugin`

```
# 安装依赖
$ apt-get update && apt-get install -y libssl-dev
# 编译
$ 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 \
  ../..
```

(下页继续)

(续上页)

```
# 需要安装到 host 环境
$ make -j
$ sudo make install
```

2. 下载 NDK，交叉编译 android aarch64 所需静态库

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

# 设置环境变量
$ export ANDROID_NDK=/path/to/android-ndk-r17c

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

$ cmake ../.. \
 -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. 此时 /tmp/android_grpc_install 应有完整的安装文件

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

43.3 三、【可跳过】自测 NDK gRPC 是否正常

1. 编译 gRPC 自带的 helloworld

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

$ cmake ../.. \
 -DCMAKE_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. 打开手机调试模式，push 编译结果到 `/data/local/tmp` 目录

tips：对于国产手机，设置 - 版本号，点击 7 次可进入开发者模式，然后才能打开 USB 调试

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

3. `adb shell` 进手机，执行 client/server

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

43.4 四、交叉编译 snpe 推理服务

1. 打开 snpe tools 官网，下载 1.59 版本。解压并设置环境变量

注意 snpe >= 1.60 开始使用 `clang-8.0`，可能导致旧设备与 `libc++_shared.so` 不兼容。

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

2. 打开 mmdeploy snpe server 目录，使用交叉编译 gRPC 时的选项

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

$ mkdir -p build && cd build
$ export ANDROID_NDK=/path/to/android-ndk-r17c
$ cmake .. \
 -DCMAKE_TOOLCHAIN_FILE=${ANDROID_NDK_ROOT}/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
```

最终可得到 infernece_server，adb push 到设备上即可执行。

43.5 五、重新生成 proto 接口

如果改过 inference.proto，需要重新生成.cpp 和.py 通信接口

```
$ 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
```

43.6 参考文档

- 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

CHAPTER 44

FAQ

44.1 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.

44.2 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.

44.3 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__)
```

- 编译时 enable_language(CUDA) 报错

```
-- 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)
```

(下页继续)

(续上页)

```
  C:/Software/cmake/cmake-3.23.1-windows-x86_64/share/cmake-3.23/Modules/
↪CMakeDetermineCUDACompiler.cmake:339 (CMAKE_DETERMINE_COMPILER_ID)
  C:/workspace/mmdeploy-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)
```

原因： CUDA Toolkit 11.1 安装在 Visual Studio 之前，造成 VS 的插件没有安装。或者 VS 的版本过新，使得 CUDA Toolkit 的安装的时候跳过了 VS 插件的安装

解决方法： 可以通过手工拷贝插件的方式来解决这个问题。比如将 `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.1\extras\visual_studio_integration\MSBuildExtensions` 中的四个文件拷贝到 `C:\Software\Microsoft Visual Studio\2022\Community\Msbuild\Microsoft\VC\v170\BuildCustomizations` 目录下。具体路径根据实际情况进行更改。

44.4 ONNX Runtime

- Windows 系统下，转模型可视化时以及 SDK 推理时遇到

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

原因：在较新的 windows 系统中，系统路径下下有两个 `onnxruntime.dll`，且会优先加载，造成冲突。

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

解决方法：以下两个方案任选其一

1. 将下载的 onnxruntime 中的 lib 目录下的 dll 拷贝到 mmdeploy_onnxruntime_ops.dll 的同级目录（推荐使用 Everything 进行查找）
2. 将系统路径下的这两个 dll 改名，使其加载不到，可能涉及到修改文件权限的操作

44.5 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 45

English

CHAPTER 46

简体中文

CHAPTER 47

apis

CHAPTER 48

apis/tensorrt

mmdeploy.apis.tensorrt.**from_onnx**(*onnx_model: Union[str, onnx.onnx_ml_pb2.ModelProto]*, *output_file_prefix: str*, *input_shapes: Dict[str, Sequence[int]]*, *max_workspace_size: int = 0*, *fp16_mode: bool = False*, *int8_mode: bool = False*, *int8_param: Optional[dict] = None*, *device_id: int = 0*, *log_level: tensorrt.Logger.Severity = tensorrt.Logger.ERROR*, ***kwargs*) → tensorrt.ICudaEngine

Create a tensorrt engine from ONNX.

参数

- **onnx_model** (*str or onnx.ModelProto*) -- Input onnx model to convert from.
- **output_file_prefix** (*str*) -- The path to save the output ncnn file.
- **input_shapes** (*Dict[str, Sequence[int]]*) -- The min/opt/max shape of each input.
- **max_workspace_size** (*int*) -- To set max workspace size of TensorRT engine. some tactics and layers need large workspace. Defaults to *0*.
- **fp16_mode** (*bool*) -- Specifying whether to enable fp16 mode. Defaults to *False*.
- **int8_mode** (*bool*) -- Specifying whether to enable int8 mode. Defaults to *False*.
- **int8_param** (*dict*) -- A dict of parameter int8 mode. Defaults to *None*.
- **device_id** (*int*) -- Choice the device to create engine. Defaults to *0*.

- **log_level** (*trt.Logger.Severity*) -- The log level of TensorRT. Defaults to *trt.Logger.ERROR*.

返回 The TensorRT engine created from onnx_model.

返回类型 tensorrt.ICudaEngine

示例

```
>>> from mmdeploy.apis.tensorrt import from_onnx
>>> engine = from_onnx(
>>>             "onnx_model.onnx",
>>>             {'input': {"min_shape" : [1, 3, 160, 160],
>>>                        "opt_shape" : [1, 3, 320, 320],
>>>                        "max_shape" : [1, 3, 640, 640]}},
>>>             log_level=trt.Logger.WARNING,
>>>             fp16_mode=True,
>>>             max_workspace_size=1 << 30,
>>>             device_id=0)
>>>             })
```

mmdeploy.apis.tensorrt.**is_available**()

Check whether TensorRT package is installed and cuda is available.

返回 True if TensorRT package is installed and cuda is available.

返回类型 bool

mmdeploy.apis.tensorrt.**is_custom_ops_available**()

Check whether TensorRT custom ops are installed.

返回 True if TensorRT custom ops are compiled.

返回类型 bool

mmdeploy.apis.tensorrt.**load**(*path: str*) → tensorrt.ICudaEngine

Deserialize TensorRT engine from disk.

参数 **path** (*str*) -- The disk path to read the engine.

返回 The TensorRT engine loaded from disk.

返回类型 tensorrt.ICudaEngine

mmdeploy.apis.tensorrt.**onnx2tensorrt**(*work_dir: str*, *save_file: str*, *model_id: int*, *deploy_cfg: Union[str, mmcv.utils.config.Config]*, *onnx_model: Union[str, onnx.onnx_ml_pb2.ModelProto]*, *device: str = 'cuda:0'*, *partition_type: str = 'end2end'*, ***kwargs*)

Convert ONNX to TensorRT.

实际案例

```
>>> from mmdeploy.backend.tensorrt.onnx2tensorrt import onnx2tensorrt
>>> work_dir = 'work_dir'
>>> save_file = 'end2end.engine'
>>> model_id = 0
>>> deploy_cfg = ('configs/mmdet/detection/'
                  'detection_tensorrt_dynamic-320x320-1344x1344.py')
>>> onnx_model = 'work_dir/end2end.onnx'
>>> onnx2tensorrt(work_dir, save_file, model_id, deploy_cfg,
        onnx_model, 'cuda:0')
```

参数

- **work_dir** (*str*) -- A working directory.
- **save_file** (*str*) -- The base name of the file to save TensorRT engine. E.g. *end2end.engine*.
- **model_id** (*int*) -- Index of input model.
- **deploy_cfg** (*str | mmcv.Config*) -- Deployment config.
- **onnx_model** (*str | onnx.ModelProto*) -- input onnx model.
- **device** (*str*) -- A string specifying cuda device, defaults to 'cuda:0'.
- **partition_type** (*str*) -- Specifying partition type of a model, defaults to 'end2end'.

mmdeploy.apis.tensorrt.**save**(*engine: tensorrt.ICudaEngine*, *path: str*) → None

Serialize TensorRT engine to disk.

参数

- **engine** (*tensorrt.ICudaEngine*) -- TensorRT engine to be serialized.
- **path** (*str*) -- The absolute disk path to write the engine.

CHAPTER 49

apis/onnxruntime

mmdeploy.apis.onnxruntime.**is_available**()

Check whether ONNX Runtime package is installed.

返回 True if ONNX Runtime package is installed.

返回类型 bool

mmdeploy.apis.onnxruntime.**is_custom_ops_available**()

Check whether ONNX Runtime custom ops are installed.

返回 True if ONNX Runtime custom ops are compiled.

返回类型 bool

CHAPTER 50

apis/ncnn

mmdeploy.apis.ncnn.**from_onnx**(*onnx_model: Union[onnx.onnx_ml_pb2.ModelProto, str]*, *output_file_prefix: str*)

Convert ONNX to ncnn.

The inputs of ncnn include a model file and a weight file. We need to use a executable program to convert the *.onnx* file to a *.param* file and a *.bin* file. The output files will save to work_dir.

示例

```
>>> from mmdeploy.apis.ncnn import from_onnx
>>> onnx_path = 'work_dir/end2end.onnx'
>>> output_file_prefix = 'work_dir/end2end'
>>> from_onnx(onnx_path, output_file_prefix)
```

参数

- **onnx_path** (*ModelProto|str*) -- The path of the onnx model.
- **output_file_prefix** (*str*) -- The path to save the output ncnn file.

mmdeploy.apis.ncnn.**is_available**()

Check whether ncnn and onnx2ncnn tool are installed.

返回 True if ncnn and onnx2ncnn tool are installed.

返回类型 bool

mmdeploy.apis.ncnn.**is_custom_ops_available**()

Check whether ncnn extension and custom ops are installed.

返回 True if ncnn extension and custom ops are compiled.

返回类型 bool

CHAPTER 51

apis/pplnn

mmdeploy.apis.pplnn.**is_available**()

Check whether pplnn is installed.

返回 True if pplnn package is installed.

返回类型 bool

CHAPTER 52

Indices and tables

- genindex
- search

Python 模块索引

m

索引

F

I

L

M

O

S

模块