Pytorch YOLOv3 移植步骤

> 此demo以yolov3为例，介绍将模型移植到MLU上的步骤，详情参考用户手册和教学视频。

1.配置环境

声明环境变量及进入虚拟环境（该操作每次进入docker都需要进行）

```
  cd /workspace/volume/private/sdk/cambricon_pytorch
  source env_pytorch.sh
```

2. 准备模型

demo位置：/workspace/volume/private/00_Yolov3_example

准备yolov3的模型：**需要将原darknet框架生成的yolov3权重yolov3.weight转换为pytorch可读取的pth格式。**
本demo已经准备好yolov3.pth模型，并保存在/workspace/volume/private/00_Yolov3_example/model/online/路径下。
【注】 模型必须以pth的格式（**pth中只有权重，不保存模型结构**）保存。

3. 在线推理

在线推理的示例代码在/workspace/volume/private/00_Yolov3_example/online/yolov3目录下。

3.1 **模型量化**

```
cd /workspace/volume/private/00_Yolov3_example/online/yolov3
bash quantize.sh #进行量化
```

模型在MLU上运行需要先进行量化，cambricon-pytorch提供了相应的接口来量化模型的权重。

```
models.object_detection.yolov3(weight_path,pretrained=True, img_size,
conf_thres, nms_thres)
mean = [0.0, 0.0, 0.0]
std  = [1.0, 1.0, 1.0]
# 调用量化接口进行量化
qconfig = {'use_avg':False, 'data_scale':1.0, 'mean': mean, 'std': std,
'per_channel': per_channel, 'firstconv':True}
quantized_model = mlu_quantize.quantize_dynamic_mlu(model, qconfig, dtype=dtype,
gen_quant=True)
......
# 保存量化模型
checkpoint = quantized_model.state_dict()
torch.save(checkpoint,'{}/yolov3_int8.pth'.format(opt.quantized_model_path))
```

量化过程需要在CPU上完成，此demo量化后的模型yolov3_int8.pth保存在/workspace/volume/private/00_Yolov3_example/model/online/目录下。

```
#模型量化 quantize.sh
python test.py --mlu false --jit false --batch_size 1 --core_number 1 --image_number 1 --half_input 1 --quantized_mode 1 --quantization true --input_channel_order 0 --quantized_model_path ../../model/online
```

读取原始模型

```
#读取原始模型，请参考"/workspace/volume/private/sdk/venv/pytorch/lib/python3.7/site-packages/torchvision/models/object_detection/yolov3/models.py"文件
    model = models.object_detection.yolov3(weight_path,pretrained=True, img_size=opt.img_size, conf_thres=opt.conf_thres, nms_thres=opt.nms_thres)
```

这里有一部分模型，代码提供YOLO模型框架

3.2 在线推理

完成量化后即可加载量化模型进行推理。
【注】 可通过python test.py -h 查看参数定义

```
bash run_online_accuracy.sh      #在线推理精度测试
bash run_online_performance.sh   #在线推理性能测试
```

模型和数据需要通过 `.to(torch_mlu.core.mlu_model.mlu_device())` 来指定设备为MLU。

```
model = models.quantization.object_detection.yolov3(quantized_weight_path,pretrained=True,quantize=True,img_size,conf_thres,nms_thres)
model.to(torch_mlu.core.mlu_model.mlu_device())
```

```
......
imgs = Variable(imgs.type(torch.HalfTensor)) if opt.half_input and opt.mlu else Variable(imgs.type(Tensor))
imgs = imgs.to(torch_mlu.core.mlu_model.mlu_device())
outputs = model(imgs)
......
```

利用 JIT 模块可以实现融合模式。融合模式会对整个网络构建一个静态图，并对静态图进行优化，有效提高性能。

```
# trace network
example = torch.randn(opt.batch_size, 3, img_size, img_size).float()
trace_input = torch.randn(1, 3, img_size, img_size).float()
if opt.half_input:
    example = example.type(torch.HalfTensor)
    trace_input = trace_input.type(torch.HalfTensor)
model = torch.jit.trace(model, trace_input.to(torch_mlu.core.mlu_model.mlu_device()), check_trace = False)
```

```
#在线融合推理精度测试
python test.py --mlu true --jit true --batch_size 1 --core_number 1 --image_number 5000 --half_input 1 --quantized_mode 1 --quantization false --input_channel_order 0 --compute_map true --run_mode false
```

模型和数据需要通过`.to(torch_mlu.core.mlu_model.mlu_device())`来指定设备为MLU。

```
        imgs = Variable(imgs.type(torch.HalfTensor)) if opt.half_input and opt.mlu else Variable(
imgs.type(Tensor))
```

PyTorch的包autograd提供了自动求导的功能。当使用autograd时，定义的前向网络会生成一个计算图：每个节点是一个Tensor，边表示由输入Tensor到输出Tensor的函数。沿着计算图的反向传播可以很容易地计算出梯度。

在实现的时候，用到了Variable对象。Variable对Tensor对象进行封装，只需要Variable::data即可取出Tensor，并且Variable还封装了该Tensor的梯度Variable::grad(是个Variable对象)。现在用Variable作为计算图的节点，则通过反向传播自动求得的导数就保存在Variable对象中了。

Variable提供了和Tensor一样的API，即能在Tensor上执行的操作也可以在Variable上执行。

参数设置问题,更改后是否影响

batch_size:size of each image batch

core_numbe:Core number of mfus and offline model with simple compilation.

half_input:the input data type

quantized_mode:the data type, 0-float16 1-int8 2-int16 3-c_int8 4-c_int16, default 1.

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.293
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.553
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.285
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.131
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.327
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.428
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.262
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.415
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.448
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.247
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.500
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.603
Mean AP: 0.553
```

> - **AveragePrecision(AP): AveragePrecision(AP):**
>
> AP % AP at IoU=0.50:0.05:0.95 (主要挑战指标)
> AP^{50} % AP at IoU=0.50 (PASCAL VOC 指标)
> AP^{75} % AP at IoU=0.75 (严格指标)
>
> - **APAcrossScales: APAcrossScales:**
>
> AP^{S} % AP （对于小目标）：area < 32^2
> AP^{L} % AP （对于中等目标）： 32^2 < area < 96^2
> AP^{M} % AP （对于大目标）：area > 96^2
>
> - **AverageRecall(AR): AverageRecall(AR):**
>
> AR^{1} % AR given 1 detection per image
> AR^{10} % AR given 10 detections per image
> AR^{100} % AR given 100 detections per image
>
> - **ARAcrossScales: ARAcrossScales:**
>
> AR^{S} % AR （对于小目标）：area < 32^2
> AR^{M} % AR （对于中等目标）：32^2 < area < 96^2
> AR^{L} % AR （对于大目标）：area > 96^2

> 使用JIT模块不清楚

```
#在线融合推理性能测试
python test.py --mlu true --jit true --batch_size 16 --core_number 16 --
image_number 496 --half_input 1 --quantized_mode 1 --quantization false --
input_channel_order 0 --compute_map false --run_mode true
```

```
Throughput(fps): 40.66380713746556
Latency(ms): 113.8092258064516
```

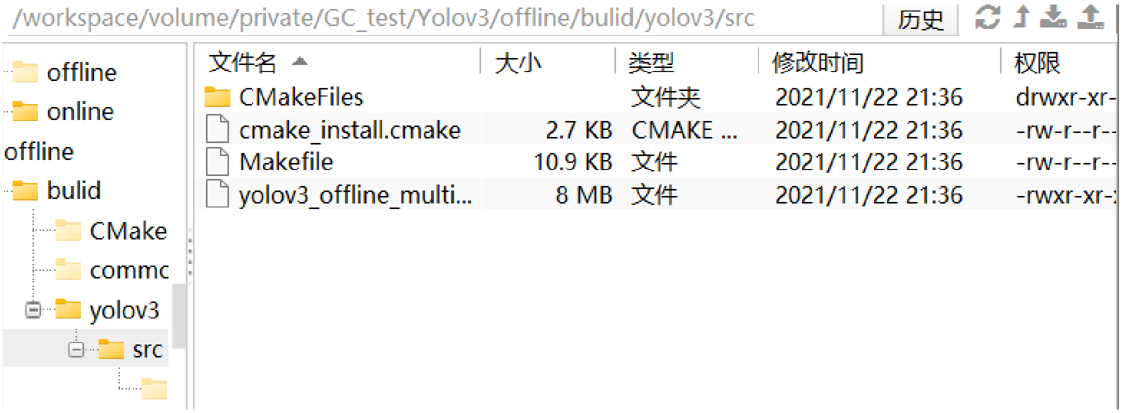
```
if opt.run_mode:
 print('Throughput(fps): ' + str(opt.image_number / total_e2e))
 print('Latency(ms): '+ str(opt.batch_size /
(opt.image_number/total_hardware) * 1000))
```

4.离线推理

离线推理的代码在/workspace/volume/private/00_Yolov3_example/offline/yolov3/src目录下。
4.1 **编译代码**

```
cd /workspace/volume/private/00_Yolov3_example/offline/
mkdir build
cd build
cmake ..
make
cd ..
```

完成编译后会在/workspace/volume/private/00_Yolov3_example/offline/build/yolov3/src目录下生成yolov3_offline_multicore可执行文件。

> 
>
>
> 这里的可执行文件有什么作用
>
> 作为脚本中可执行的文件，里面是BANG或者cnrt代码，生成离线模型运行代码，在机器上部署的方式
>
> cmake的作用

4.2 **生成离线模型及推理**

```
  cd /workspace/volume/private/00_Yolov3_example/offline/yolov3/

  #生成batch_size=1,core_number=1的离线模型yolov3.cambricon并保存在/workspace/volume/private/00_Yolov3_example/model/offline/目录下。
  bash run_get_accuracy_offlinemodel.sh

  #进行离线推理精度测试
  bash run_offline_accuracy.sh

  #生成batch_size=16,core_number=16的离线模型yolov3.cambricon保存在../../model/offline/目录下。
  bash run_get_performance_offlinemodel.sh

  #进行离线推理性能测试
  bash run_offline_performance.sh
```

通过调用`torch_mlu.core.mlu_model.save_as_cambricon(model_name)`接口，在进行jit.trace时会自动生成离线模型。生成的离线模型一般是以model_name.cambricon命名的离线模型文件，其中包含一个名为 model_name 的模型。

```
torch_mlu.core.mlu_model.save_as_cambricon('yolov3')
```

> ```
> #生成离线模型 run_get_accuracy_offlinemodel.sh
> cd ../../online/yolov3/
> python test.py --mlu true --jit true --batch_size 1 --core_number 1 --image_number 10 --half_input 1 --quantized_mode 1 --quantization false --input_channel_order 0 --compute_map false --save_offline_model true --run_mode false
> cd -
> ```

```python
if opt.jit:
    # trace network
    example = torch.randn(opt.batch_size, 3, opt.img_size, opt.img_size).float()
    trace_input = torch.randn(1, 3, opt.img_size, opt.img_size).float()
    if opt.half_input:
        example = example.type(torch.HalfTensor)
        trace_input = trace_input.type(torch.HalfTensor)
    model = torch.jit.trace(model, trace_input.to(ct.mlu_device()), check_trace = False)
    if opt.save_offline_model:
        ct.save_as_cambricon(offline_model_name)
    # warm up
    logger.info('Warming up...')
    for i in range(10):
        model(example.to(ct.mlu_device()))
```

运行run_offline_accuracy.sh时有问题

build		文件夹	2021/11/22 21:36	drwxr-xr-x	root/root
cmake		文件夹	2021/11/15 20:49	drwxr-xr-x	root/root
common		文件夹	2021/11/15 20:49	drwxr-xr-x	root/root
scripts		文件夹	2021/11/15 20:49	drwxr-xr-x	root/root
yolov3		文件夹	2021/11/23 19:56	drwxr-xr-x	root/root
CMakeLists.txt	1.8 KB	文本文档	2021/11/15 20:49	-rw-r--r--	root/root

路径build有问题

```
    # run offline.cambricon model command
    run_cmd="${EXAMPLE_DIR}/yolov3_offline_multicore$SUFFIX -offlinemodel $CURRENT_DIR/../../model/offline/${network}.cambricon -images $file_list -labels $CURRENT_DIR/label_map_coco.txt -outputdir $CURRENT_DIR/output -dump 1 -simple_compile 1 -dataset_path /workspace/dataset/public/zhumeng-dataset/coco_2014/  -input_format $channel_order &>> $CURRENT_DIR/$log_file"

    check_cmd="python $OFFLINE_DIR/scripts/meanAP_COCO.py --file_list $file_list --result_dir $CURRENT_DIR/output --ann_dir /workspace/dataset/public/zhumeng-dataset/coco_2014/ --data_type val2014 &>> $CURRENT_DIR/$log_file"

    echo "run_cmd: $run_cmd" &>> $CURRENT_DIR/$log_file
                                                                              39,1          42%
```

运行结果

```
==========================================================
running yolov3 offline multiple core ...
--------------------
running offline test...
HardwareLatency(ms): 26.1183
Inference count: 5000 times
CNRT: 4.10.1 a884a9a
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.290
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.552
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.280
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.125
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.326
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.428
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.261
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.412
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.445
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.242
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.498
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.602
Mean AP: 0.552
```

```
#生成离线模型
cd ../../online/yolov3/
python test.py --mlu true --jit true --batch_size 16 --core_number 16 --image_number 16 --half_input 1 --quantized_mode 1 --quantization false --input_channel_order 0 --compute_map false --save_offline_model true --run_mode false
cd -
```

```
    # run offline.cambricon model command
    run_cmd="${EXAMPLE_DIR}/yolov3_offline_multicore$SUFFIX -offlinemodel $CURRENT_DIR/../../model/offline/${network}.cambricon -images $file_list -labels $CURRENT_DIR/label_map_coco.txt -outputdir /home/tmp -dump 1 -simple_compile 1 -dataset_path /workspace/dataset/public/zhumeng-dataset/coco_2014/ -perf_mode 1 -perf_mode_img_num 496  -input_format $channel_order &>> $CURRENT_DIR/$log_file"

    check_cmd="python $OFFLINE_DIR/scripts/meanAP_COCO.py --file_list $file_list --result_dir /home/tmp --ann_dir /workspace/dataset/public/zhumeng-dataset/coco_2014/ --data_type val2014 &>> $CURRENT_DIR/$log_file"
```

> 离线模型的性能更高

离线模型运行代码的编写可以参考cnrt文档中的示例。

```
// when generating an offline model, u need cnml and cnrt both
 // when running an offline model, u need cnrt only
```

> CNRT（Cambricon Neuware Runtime Library，寒武纪运行时库）提供了一套面向MLU（Machine Learning Unit，寒武纪机器学习单元）设备的高级别的接口，用于主机与MLU设备之间的交互。CNRT作为寒武纪软件系统最底层支撑，所有其他的寒武纪软件运行都需要调用CNRT接口。

> 使用YOLOv5训练BDD数据集得到新的权重，按照以上部署方案进行，最终生成的离线模型设备更改为MLU220，YOLOv5的离线模型运行代码，参考/workspace/volume/private/zhumeng/offline