模型压缩优化

模型压缩优化可以更好地满足在边缘、移动端如NVIDIA Jetson，Raspberry Pi上部署的性能需求，有效降低模型的体积，以及计算量，加速预测性能。 本文档主要介绍和测试了模型裁剪和模型量化两种广泛使用的模型优化技术，评估优劣和适用性，旨在为后面项目的模型优化提供建议方案。

环境配置： 本次测试所用环境主要包括：

```
paddle-bfloat              0.1.7
paddlepaddle-gpu        2.3.2.post112
paddleslim                 2.2.1
paddlex                    2.1.0
pandas                     1.3.5
pexpect                    4.8.0
pickleshare                0.7.5
Pillow                     9.2.0
numpy                      1.21.6
opencv-python              4.6.0.66
openpyxl                   3.0.10
scikit-learn               0.23.2
scipy                      1.7.3
```

需检查cuda版本及是否安装成功：

```
$ nvcc --version $ nvidia-smi
```

本次测试使用cuda版本为：

```
ubuntu20.04
Driver Version: 525.60.11
Cuda compilation tools, release 11.2, V11.2.142
Build cuda_11.2.r11.2/compiler.29558016_0
cuDNN Version: 8.2
```

注：paddlepaddle-gpu暂仅支持ubuntu20.04及以下版本

环境安装：

1. 安装paddlepaddle-gpu:

注：paddlepaddle-gpu最新版为2.4.1，但是存在bug，无法正确使用cuda，因此手动降级版本，经测试2.3.2及以下可以正常运行

```
conda：conda install paddlepaddle-gpu==2.3.2 cudatoolkit=10.2 --channel https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/Paddle

pip: python -m pip install paddlepaddle-gpu==2.3.2 -i https://pypi.tuna.tsinghua.edu.cn/simple

docker:
1. 拉取预安装 PaddlePaddle 的镜像：
nvidia-docker pull registry.baidubce.com/paddlepaddle/paddle:2.3.2-gpu-cuda10.2-cudnn7.6-trt7.0
2. 用镜像构建并进入Docker容器：
nvidia-docker run --name paddle -it -v $PWD:/paddle registry.baidubce.com/paddlepaddle/paddle:2.3.2-gpu-cuda10.2-cudnn7.6-trt7.0
/bin/bash
```

注：和pytorch不同，paddlepaddle-gpu 使用外部cuda，所以需要确保本机正确安装cuda.cudnn等，可以通过查看/usr/local/ 路径下是否有cuda文件夹来判断是否安装。若无安装，推荐使用docker的方式安装paddle。

2. 安装paddlex:

```
pip  install  paddlex==2.1.0  -i  https://mirror.baidu.com/pypi/simple
```

注：若安装失败，可参考源码安装，见：https://github.com/PaddlePaddle/PaddleX/blob/develop/docs/quick_start_API.md

模型剪裁

模型裁剪的一般过程是：在正常训练神经网络模型后，分析模型各层参数在不同的剪裁比例下的敏感度并根据敏感度信息裁剪模型，最后再对剪裁后的模型重新训练。

模型裁剪可以更好地满足在端侧、移动端上部署场景下的性能需求，可以有效得降低模型的体积，以及计算量，加速预测性能。经过比较分析，这里测试PaddleSlim的基于敏感度的通道裁剪算法。

使用方法

模型裁剪相对比我们普通训练一个模型，步骤会多出两步

1. 采用正常的方式训练一个模型
2. 对模型的参数进行敏感度分析
3. 根据第2步得到的敏感度信息，对模型进行裁剪，并以第1步训练好的模型作为预训练权重，继续进行训练

注：目前仅在目标检测模型进行测试，对于OCR以及语义分割模型是否适用还需要进一步测试。

第一步 正常训练目标检测模型

此步骤中采用正常的代码进行模型训练。在此步骤中，基于仪表盘数据集，其中训练集725张，测试集58张，数据集保存路径为：

```
sftp://10.10.2.208:50022/data/data1/zhangyl/meter_det
```

测试所用模型为：

> YOLOv3_ResNet34

> PPyolov2_ResNet50vd_dcn

> YOLOv3_MobileNetV3

训练好的模型保存路径为（以ppyolov2_r50vd_dcn 为例）：

```
sftp://10.10.2.208:50022/data/data1/zhangyl/output1/ppyolov2_r50vd_dcn/best_model
```

训练参数为：

```
num_epochs=270,
train_batch_size=4,
learning_rate=0.000125/2,
warmup_steps=500,
warmup_start_lr=0.0,
lr_decay_epochs=[213,  240],
lr_decay_gamma=0.1,
save_interval_epochs=10,
log_interval_steps=25,
pretrain_weights='COCO'
```

测试模型精度和处理速度为：

模型	内存消耗(MiB)	GPU显存占用(MiB)	精度	运行时间(s)	模型大小(MB)
PPyolov2_ResNet50vd_dcn	3750.9	2774	0.991	2.53	364.6
YOLOv3_ResNet34	4552.4	2534	0.986	2.88	306.3
YOLOv3_MobileNetV3	3988.4	2194	0.978	1.84	185.7

注:计算内存和模型显存方式见文档末

第二步 模型剪裁

此步骤中，我们需要加载第一步训练保存的模型，并通过不断地遍历参数，分析各参数裁剪后在验证数据集上的精度损失，以此判断各参数的敏感度。

注：目标检测模型的剪裁依赖PaddleSlim 2.1.0

模型剪裁代码为 params_analysis.py ,保存路径为（以ppyolov2_r50vd_dcn 为例）：

```
sftp://10.10.2.208:50022/data/data1/zhangyl/yibiaopan/params_analysis.py
```

主要执行了以下API：

- step 1: 分析模型各层参数在不同的剪裁比例下的敏感度

  主要由两个API完成:

```
model  =  pdx.load_model('output/yolov3_darknet53/best_model')
model.analyze_sensitivity(
        dataset=eval_dataset,
        batch_size=1,
        save_dir='output/yolov3_darknet53/prune')
```

  参数分析完后， output/yolov3_darknet53/prune 目录下会得到 model.sensi.data 文件，此文件保存了不同剪裁比例下各层参数的敏感度信息。

注：如果之前运行过该步骤，第二次运行时会自动加载已有的 output/yolov3_darknet53/prune/model.sensi.data *，不再进行敏感度分析。*

- step 2: 根据选择的FLOPs减小比例对模型进行剪裁

```
model.prune(pruned_flops=.2,  save_dir='./')
```

FLOPs *是模型裁剪比例，默认为0.2。在实际使用时，可以根据自己的需求，控制裁剪比例。*

在此步骤中，会得到保存的 sensi.data 文件，这个文件保存了模型中每个参数的敏感度，在后续的裁剪训练中，会根据此文件中保存的信息，对各个参数进行裁剪。

- step 3: 对剪裁后的模型重新训练

在前两步，我们得到了正常训练保存的模型和基于该保存模型得到的参数敏感度信息文件 sensi.data ， 接下来则是进行模型裁剪训练。

```
model.train(
        num_epochs=270,
        train_dataset=train_dataset,
        train_batch_size=8,
        eval_dataset=eval_dataset,
        learning_rate=0.001  /  8,
        warmup_steps=1000,
        warmup_start_lr=0.0,
        save_interval_epochs=5,
        lr_decay_epochs=[216,  243],
        save_dir='output/yolov3_darknet53/prune')
```

重新训练后的模型保存在 output/yolov3_darknet53/prune 。

注： 重新训练时需将 pretrain_weights 设置为 None ， 否则模型会加载 pretrain_weights 指定的预训练模型参数。

裁剪效果

在上述数据集上，经过裁剪训练后，模型的效果如下:

模型	内存消耗(MiB)	GPU显存占用(MiB)	精度	运行时间(s)	模型大小(MB)
PPyolov2_ResNet50vd_dcn	2408.8	945	0.989	0.38	221.4
YOLOv3_ResNet34	2330.8	938	0.983	0.64	306.3
YOLOv3_MobileNetV3	1767.6	909	0.975	0.101	101.9

总结:

可以看到，跟原始的训练模型相比，平均内存消耗减少1000Mb，GPU显存占用减少1500Mb，精度基本保持不变，运行时间降低85%，模型大小减少40%。

可以更好地满足在边缘、移动端如NVIDIA Jetson,Raspberry Pi上部署的性能需求，有效降低模型的体积，以及计算量，加速预测性能。

模型量化

模型量化将模型的计算从浮点型转为整型，从而加速模型的预测计算速度，在移动端/边缘端设备上降低模型的体积。

第一步 正常训练图像分类模型

同上模型裁剪。

第二步 模型量化

```
python  quantize.py.py
```

模型量化代码为 quantize.py ,保存路径为：

```
sftp://10.10.2.208:50022/data/data1/zhangyl/yibiaopan/quantize.py
```

quantize.py 中主要执行了以下API：

step 1: 加载之前训练好的模型

```
model  =  pdx.load_model('output/mobilenet_v3/best_model')
```

step 2: 量化训练

```
model.quant_aware_train(
        num_epochs=100,
        train_dataset=train_dataset,
        train_batch_size=4,
        eval_dataset=eval_dataset,
        learning_rate=0.0001 / 4,
        save_dir='output/mobilenet_v3/quant',
        use_vdl=True)
```

量化训练后的模型保存在 output/mobilenet_v3/quant 。

注： 重新训练时需将 pretrain_weights *设置为* None *， 否则模型会加载* pretrain_weights *指定的预训练模型参数。*

量化效果

在上述数据集上，经过量化训练后，模型的效果如下:

模型	内存消耗(MiB)	GPU显存占用(MiB)	精度	运行时间(s)	模型大小(MB)
PPyolov2_ResNet50vd_dcn	4152.8	950	0.981	0.72	224.2
YOLOv3_MobileNetV3	2586.8	936	0.965	0.34	97.1

总结:

可以看到，跟原始的训练模型相比，模型的平均内存，GPU显存占用，运行时间，模型大小都有所减小，精度基本保持不变。跟模型裁剪相比，模型量化更关注在保持精度基本不变的前提下尽可能多的压缩的模型大小，其他方面虽然有所优化，但是跟模型裁剪相比还是有所不如。

因此，经过对比，模型裁剪拥有更好的效果。但是考虑到两种优化方式的原理不同，是否可以在模型裁剪的基础上再进行模型量化也是一个可以尝试的方向。

计算内存和模型显存消耗方式

经过比较，最终选择Python库memory_profiler和Pytorch-Memory-Utils来内存和显存的消耗。

memory_profiler

安装和引用:

> pip install memory_profiler#Load its magic function

> %load_ext memory_profiler

> from memory_profiler import profile

1. 查找一行的内存消耗： 只需要在代码的前面加上魔法函数 %memit,如

```
%memit  x  =  10+5
#  Output
peak  memory:  54.01  MiB,  increment:  0.27  MiB
```

这里，峰值内存(peak memory)是运行此代码的进程消耗的内存。增量只是由于添加这行代码而需要/消耗的内存。同样的逻辑也适用于以下其他的显示。

2. 查找函数的内存消耗：在调用函数的行的开头添加魔法函数，如

```
def  addition():
       a  =  [1]  *  (10  **  1)
       b  =  [2]  *  (3  *  10  **  2)
       sum  =  a+b
       return  sum

%memit  addition()
#  Output
peak  memory:  36.36  MiB,  increment:  0.01  MiB
```

3. 逐行查找函数的内存消耗:使用@profile 装饰器,如:

```
from memory_profiler import profile

@profile
def addition():
  a = [1] * (10 ** 1)
  b = [2] * (3 * 10 ** 2)
  sum = a+b
  return sum
  %memit addition()


# Output
Line #    Mem usage    Increment  Line Contents
================================================
     2     36.4 MiB     36.4 MiB   @profile
     3                             def addition():
     4     36.4 MiB      0.0 MiB       a = [1] * (10 ** 1)
     5   3851.1 MiB   3814.7 MiB       b = [2] * (3 * 10 ** 2)
     6   7665.9 MiB   3814.8 MiB       sum = a+b
     7   7665.9 MiB      0.0 MiB       return sum
peak memory: 7665.88 MiB, increment: 7629.52 MiB
```

Pytorch-Memory-Utils

通过Pytorch-Memory-Utils工具，在使用显存的代码中间插入检测函数，这样就可以输出在当前行代码时所占用的GPU显存，如：

```
import torch
import inspect

from torchvision import models
```

```
from  gpu_mem_track  import  MemTracker    #  引用显存跟踪代码

device  =  torch.device('cuda:0')

frame  =  inspect.currentframe()
gpu_tracker  =  MemTracker(frame)           #  创建显存检测对象

gpu_tracker.track()                                  #  开始检测
cnn  =  models.vgg19(pretrained=True).to(device)    #  运行模型程序
gpu_tracker.track()
```