PyTorch Trick集锦

z.defying 夕小瑶的卖萌屋 9月15日

星标/置顶小屋，带你解锁

最萌最前沿的NLP、搜索与推荐技术

文 | z.defying@知乎
来源 | https://zhuanlan.zhihu.com/p/76459295

前言

本文整理了13则PyTorch使用的小窍门，包括了指定GPU编号、梯度裁剪、扩展单张图片维度等实用技巧，能够帮助工作者更高效地完成任务。

1、指定GPU编号
2、查看模型每层输出详情
3、梯度裁剪
4、扩展单张图片维度
5、one hot编码
6、防止验证模型时爆显存
7、学习率衰减
8、冻结某些层的参数
9、对不同层使用不同学习率
10、模型相关操作
11、Pytorch内置one hot函数
12、网络参数初始化
13、加载内置预训练模型

1、指定GPU编号

- 设置当前使用的GPU设备仅为0号设备，设备名称为 `/gpu:0`： `os.environ["CUDA_VISIBLE_DEVICES"] = "0"`
- 设置当前使用的GPU设备为0,1号两个设备，名称依次为 `/gpu:0`、`/gpu:1`： `os.environ["CUDA_VISIBLE_DEVICES"] = "0,1"` ，根据顺序表示优先使用0号设备,然后使用1号设备。

指定GPU的命令需要放在和神经网络相关的一系列操作的前面。

2、查看模型每层输出详情

Keras有一个简洁的API来查看模型的每一层输出尺寸，这在调试网络时非常有用。现在在PyTorch中也可以实现这个功能。使用很简单，如下用法：

```
from torchsummary import summary
summary(your_model, input_size=(channels, H, W))
```

`input_size` 是根据你自己的网络模型的输入尺寸进行设置。

3、梯度裁剪

```
import torch.nn as nn

outputs = model(data)
loss= loss_fn(outputs, target)
optimizer.zero_grad()
loss.backward()
nn.utils.clip_grad_norm_(model.parameters(), max_norm=20, norm_type=2)
optimizer.step()
```

`nn.utils.clip_grad_norm_` 的参数：

- **parameters** – 一个基于变量的迭代器，会进行梯度归一化
- **max_norm** – 梯度的最大范数
- **norm_type** – 规定范数的类型，默认为L2

@不椭的椭圆 提出：梯度裁剪在某些任务上会额外消耗大量的计算时间，可移步评论区查看详情。

4、扩展单张图片维度

因为在训练时的数据维度一般都是（batch_size，c，h，w），而在测试时只输入一张图片，所以需要扩展维度，扩展维度有多个方法：

```
import cv2
import torch

image = cv2.imread(img_path)
image = torch.tensor(image)
print(image.size())

img = image.view(1, *image.size())
print(img.size())

# output:
# torch.Size([h, w, c])
# torch.Size([1, h, w, c])
```

或

```
import cv2
import numpy as np

image = cv2.imread(img_path)
print(image.shape)
img = image[np.newaxis, :, :, :]
print(img.shape)

# output:
# (h, w, c)
# (1, h, w, c)
```

或（感谢 **@coldleaf** 的补充）

```
import cv2
import torch
```

```
image = cv2.imread(img_path)
image = torch.tensor(image)
print(image.size())

img = image.unsqueeze(dim=0)
print(img.size())

img = img.squeeze(dim=0)
print(img.size())

# output:
# torch.Size([(h, w, c)])
# torch.Size([1, h, w, c])
# torch.Size([h, w, c])
```

`tensor.unsqueeze(dim)`：扩展维度，dim指定扩展哪个维度。 `tensor.squeeze(dim)`：去除dim指定的且size为1的维度，维度大于1时，squeeze()不起作用，不指定dim时，去除所有size为1的维度。

5、独热编码

在PyTorch中使用交叉熵损失函数的时候会自动把label转化成onehot，所以不用手动转化，而使用MSE需要手动转化成onehot编码。

```
import torch
class_num = 8
batch_size = 4

def one_hot(label):
    """
    将一维列表转换为独热编码
    """
    label = label.resize_(batch_size, 1)
    m_zeros = torch.zeros(batch_size, class_num)
    # 从 value 中取值，然后根据 dim 和 index 给相应位置赋值
```

```
    onehot = m_zeros.scatter_(1, label, 1)  # (dim,index,value)

    return onehot.numpy()  # Tensor -> Numpy

label = torch.LongTensor(batch_size).random_() % class_num  # 对随机数取余
print(one_hot(label))

# output:
[[0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]]
```

注：第11条有更简单的方法。

6、防止验证模型时爆显存

验证模型时不需要求导，即不需要梯度计算，关闭autograd，可以提高速度，节约内存。如果不关闭可能会爆显存。

```
with torch.no_grad():
    # 使用model进行预测的代码
    pass
```

感谢**@zhaz** 的提醒，我把 torch.cuda.empty_cache() 的使用原因更新一下。这是原回答：

> Pytorch 训练时无用的临时变量可能会越来越多，导致 `out of memory` ，可以使用下面语句来清理这些不需要的变量。

官网上的解释为：

> Releases all unoccupied cached memory currently held by the caching allocator so that those can be used in other GPU application and visible innvidia-smi. `torch.cuda.empty_cache()`

意思就是PyTorch的缓存分配器会事先分配一些固定的显存，即使实际上tensors并没有使用完这些显存，这些显存也不能被其他应用使用。这个分配过程由第一次CUDA内存访问触发的。而 `torch.cuda.empty_cache()` 的作用就是释放缓存分配器当前持有的且未占用的缓存显存，以便这些显存可以被其他GPU应用程序中使用，并且通过 `nvidia-smi` 命令可见。注意使用此命令不会释放tensors占用的显存。对于不用的数据变量，Pytorch 可以自动进行回收从而释放相应的显存。更详细的优化可以查看 优化显存使用 和 显存利用问题。

7、学习率衰减

```
import torch.optim as optim
from torch.optim import lr_scheduler

# 训练前的初始化
optimizer = optim.Adam(net.parameters(), lr=0.001)
scheduler = lr_scheduler.StepLR(optimizer, 10, 0.1)  # # 每过10个epoch，学习率乘以0.1

# 训练过程中
for n in n_epoch:
    scheduler.step()
    ...
```

可以随时查看学习率的值： `optimizer.param_groups[0]['lr']` 。还有其他学习率更新的方式：

1、自定义更新公式： `scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lambda epoch:1/(epoch+1))`

2、不依赖epoch更新学习率： `lr_scheduler.ReduceLROnPlateau()` 提供了基于训练中某些测量值使学习率动态下降的方法，它的参数说明到处都可以查到。
提醒一点就是参数 mode='min' 还是'max'，取决于优化的的损失还是准确率，即使用 `scheduler.step(loss)` 还是 `scheduler.step(acc)` 。

8、冻结某些层的参数

参考：

https://www.zhihu.com/question/311095447/answer/589307812

在加载预训练模型的时候，我们有时想冻结前面几层，使其参数在训练过程中不发生变化。我们需要先知道每一层的名字，通过如下代码打印：

```
net = Network()  # 获取自定义网络结构
for name, value in net.named_parameters():
    print('name: {0},\t grad: {1}'.format(name, value.requires_grad))
```

假设前几层信息如下：

```
name: cnn.VGG_16.convolution1_1.weight,  grad: True
name: cnn.VGG_16.convolution1_1.bias,  grad: True
name: cnn.VGG_16.convolution1_2.weight,  grad: True
name: cnn.VGG_16.convolution1_2.bias,  grad: True
name: cnn.VGG_16.convolution2_1.weight,  grad: True
name: cnn.VGG_16.convolution2_1.bias,  grad: True
name: cnn.VGG_16.convolution2_2.weight,  grad: True
name: cnn.VGG_16.convolution2_2.bias,  grad: True
```

后面的True表示该层的参数可训练，然后我们定义一个要冻结的层的列表：

```
no_grad = [
    'cnn.VGG_16.convolution1_1.weight',
    'cnn.VGG_16.convolution1_1.bias',
    'cnn.VGG_16.convolution1_2.weight',
    'cnn.VGG_16.convolution1_2.bias'
]
```

冻结方法如下：

```
net = Net.CTPN()  # 获取网络结构
for name, value in net.named_parameters():
    if name in no_grad:
        value.requires_grad = False
```

```
    else:
        value.requires_grad = True
```

冻结后我们再打印每层的信息：

```
name: cnn.VGG_16.convolution1_1.weight,  grad: False
name: cnn.VGG_16.convolution1_1.bias,  grad: False
name: cnn.VGG_16.convolution1_2.weight,  grad: False
name: cnn.VGG_16.convolution1_2.bias,  grad: False
name: cnn.VGG_16.convolution2_1.weight,  grad: True
name: cnn.VGG_16.convolution2_1.bias,  grad: True
name: cnn.VGG_16.convolution2_2.weight,  grad: True
name: cnn.VGG_16.convolution2_2.bias,  grad: True
```

可以看到前两层的weight和bias的requires_grad都为False，表示它们不可训练。最后在定义优化器时，只对requires_grad为True的层的参数进行更新。

```
optimizer = optim.Adam(filter(lambda p: p.requires_grad, net.parameters()), lr=0.01)
```

9、对不同层使用不同学习率

我们对模型的不同层使用不同的学习率。还是使用这个模型作为例子：

```
net = Network()  # 获取自定义网络结构
for name, value in net.named_parameters():
    print('name: {}'.format(name))

# 输出:
# name: cnn.VGG_16.convolution1_1.weight
# name: cnn.VGG_16.convolution1_1.bias
# name: cnn.VGG_16.convolution1_2.weight
# name: cnn.VGG_16.convolution1_2.bias
# name: cnn.VGG_16.convolution2_1.weight
# name: cnn.VGG_16.convolution2_1.bias
```

```
# name: cnn.VGG_16.convolution2_2.weight
# name: cnn.VGG_16.convolution2_2.bias
```

对 convolution1 和 convolution2 设置不同的学习率，首先将它们分开，即放到不同的列表里：

```
conv1_params = []
conv2_params = []

for name, parms in net.named_parameters():
    if "convolution1" in name:
        conv1_params += [parms]
    else:
        conv2_params += [parms]

# 然后在优化器中进行如下操作：
optimizer = optim.Adam(
    [
        {"params": conv1_params, 'lr': 0.01},
        {"params": conv2_params, 'lr': 0.001},
    ],
    weight_decay=1e-3,
)
```

我们将模型划分为两部分，存放到一个列表里，每部分就对应上面的一个字典，在字典里设置不同的学习率。当这两部分有相同的其他参数时，就将该参数放到列表外面作为全局参数，如上面的`weight_decay`。也可以在列表外设置一个全局学习率，当各部分字典里设置了局部学习率时，就使用该学习率，否则就使用列表外的全局学习率。

10、模型相关操作

这个内容比较多，我写成了一篇文章：

https://zhuanlan.zhihu.com/p/73893187

11、Pytorch内置one_hot函数

感谢@yangyangyang 补充：Pytorch 1.1后，one_hot可以直接用 `torch.nn.functional.one_hot` 。然后我将Pytorch升级到1.2版本，试用了下 one_hot 函数，确实很方便。具体用法如下：

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

tensor =  torch.arange(0, 5) % 3  # tensor([0, 1, 2, 0, 1])
one_hot = F.one_hot(tensor)

# 输出:
# tensor([[1, 0, 0],
#         [0, 1, 0],
#         [0, 0, 1],
#         [1, 0, 0],
#         [0, 1, 0]])
```

`F.one_hot` 会自己检测不同类别个数，生成对应独热编码。我们也可以自己指定类别数：

```
tensor =  torch.arange(0, 5) % 3  # tensor([0, 1, 2, 0, 1])
one_hot = F.one_hot(tensor, num_classes=5)

# 输出:
# tensor([[1, 0, 0, 0, 0],
#         [0, 1, 0, 0, 0],
#         [0, 0, 1, 0, 0],
#         [1, 0, 0, 0, 0],
#         [0, 1, 0, 0, 0]])
```

升级 Pytorch (cpu 版本) 的命令： `conda install pytorch torchvision \-c pytorch` （希望Pytorch升级不会影响项目代码）

12、网络参数初始化

神经网络的初始化是训练流程的重要基础环节，会对模型的性能、收敛性、收敛速度等产生重要的影响。

以下介绍两种常用的初始化操作。

(1) 使用pytorch内置的torch.nn.init方法。常用的初始化操作，例如正态分布、均匀分布、xavier初始化、kaiming初始化等都已经实现，可以直接使用。具体详见PyTorch 中torch.nn.init 中文文档。

```
init.xavier_uniform(net1[0].weight)
```

(2) 对于一些更加灵活的初始化方法，可以借助numpy。对于自定义的初始化方法，有时tensor的功能不如numpy强大灵活，故可以借助numpy实现初始化方法，再转换到tensor上使用。

```
for layer in net1.modules():
    if isinstance(layer, nn.Linear): # 判断是否是线性层
        param_shape = layer.weight.shape
        layer.weight.data = torch.from_numpy(np.random.normal(0, 0.5, size=param_sha
        # 定义为均值为 0，方差为 0.5 的正态分布
```

13、加载内置预训练模型

`torchvision.models` 模块的子模块中包含以下模型：

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet

导入这些模型的方法为：

```
import torchvision.models as models
resnet18 = models.resnet18()
```

```
alexnet = models.alexnet()
vgg16 = models.vgg16()
```

有一个很重要的参数为 `pretrained` ，默认为 `False` ，表示只导入模型的结构，其中的权重是随机初始化的。如果 `pretrained` 为 `True` ，表示导入的是在 `ImageNet` 数据集上预训练的模型。

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
```

更多的模型可以查看：

https://pytorch-cn.readthedocs.io/zh/latest/torchvision/torchvision-models/