Note Information

Author : AbbasXu
Date : 2022-08-18
Title : pytorch基础
Keywords : #pytroch #学习率 #损失函数 #模型微调 #数据增强 #调参

自定义损失函数

损失函数与层的共性

本质上来说，损失函数和自定义层有着很多类似的地方，他们都是通过对输入进行函数运算，得到一个输出，这也就是层的功能。只不过层的函数运算比较不一样，可能是线性组合、卷积运算等，但终归也是函数运算，正是基于这样的共性，所以我们可以统一的使用nn.Module类来定义损失函数，而且定义的方式也和前面的层是大同小异的。

以函数的方式定义

直接以函数定义的方式定义一个自己的函数。
特点：简单

```
def my_loss(output, target):
    loss = torch.mean((output - target)**2)
    return loss
```

以类的方式定义

特点：常用
损失函数类就需要继承自`nn.Module`类
以DiceLoss为例，其数学公式如下所示：

$$DSC = \frac{2|X \cap Y|}{|X| + |Y|}$$

```
class DiceLoss(nn.Module):
    def __init__(self,weight=None,size_average=True):
        super(DiceLoss,self).__init__()
```

```
    def forward(self,inputs,targets,smooth=1):
        inputs = F.sigmoid(inputs)
        inputs = inputs.view(-1)
        targets = targets.view(-1)
        intersection = (inputs * targets).sum()
        dice = (2.*intersection + smooth)/(inputs.sum() + targets.sum() +
smooth)
        return 1 - dice

# 使用方法
criterion = DiceLoss()
loss = criterion(input,targets)
```

通过nn.functional直接定义函数来完成

一般情况下，损失函数是没有参数信息和状态需要维护的，所以更多的时候我们没有必要小题大做，自己去定义一个损失函数的类，我们只需要一个计算的数学函数即可，nn.functional里面定义了一些常见的函数，当然也包括一些常见的损失函数，如下：

```
@weak_script
def smooth_l1_loss(input, target, size_average=None, reduce=None,
reduction='mean'):

@weak_script
def l1_loss(input, target, size_average=None, reduce=None,
reduction='mean'):

@weak_script
def mse_loss(input, target, size_average=None, reduce=None,
reduction='mean'):

@weak_script
def margin_ranking_loss(input1, input2, target, margin=0,
size_average=None,
                        reduce=None, reduction='mean'):

@weak_script
def hinge_embedding_loss(input, target, margin=1.0, size_average=None,
```

```
                          reduce=None, reduction='mean'):

@weak_script
def multilabel_margin_loss(input, target, size_average=None, reduce=None,
reduction='mean'):

@weak_script
def soft_margin_loss(input, target, size_average=None, reduce=None,
reduction='mean'):

@weak_script
def multilabel_soft_margin_loss(input, target, weight=None,
size_average=None,
                                reduce=None, reduction='mean'):
@weak_script
def cosine_embedding_loss(input1, input2, target, margin=0,
size_average=None,
                          reduce=None, reduction='mean'):
@weak_script
def multi_margin_loss(input, target, p=1, margin=1., weight=None,
size_average=None):
```

注:
在自定义损失函数时，涉及到数学运算时，我们最好全程使用PyTorch提供的张量计算接口，这样就不需要我们实现自动求导功能并且我们可以直接调用cuda。

动态调整学习率

使用官方scheduler

PyTorch已经在`torch.optim.lr_scheduler`为我们封装好了一些动态调整学习率的方法供我们使用

- `lr_scheduler.LambdaLR`
- `lr_scheduler.MultiplicativeLR`
- `lr_scheduler.StepLR`
- `lr_scheduler.MultiStepLR`
- `lr_scheduler.ExponentialLR`

- `lr_scheduler.CosineAnnealingLR`
- `lr_scheduler.ReduceLROnPlateau`
- `lr_scheduler.CyclicLR`
- `lr_scheduler.OneCycleLR`
- `lr_scheduler.CosineAnnealingWarmRestarts`
  注：
  我们在使用官方给出的`torch.optim.lr_scheduler`时，需要将`scheduler.step()`放在`optimizer.step()`后面进行使用。

自定义scheduler

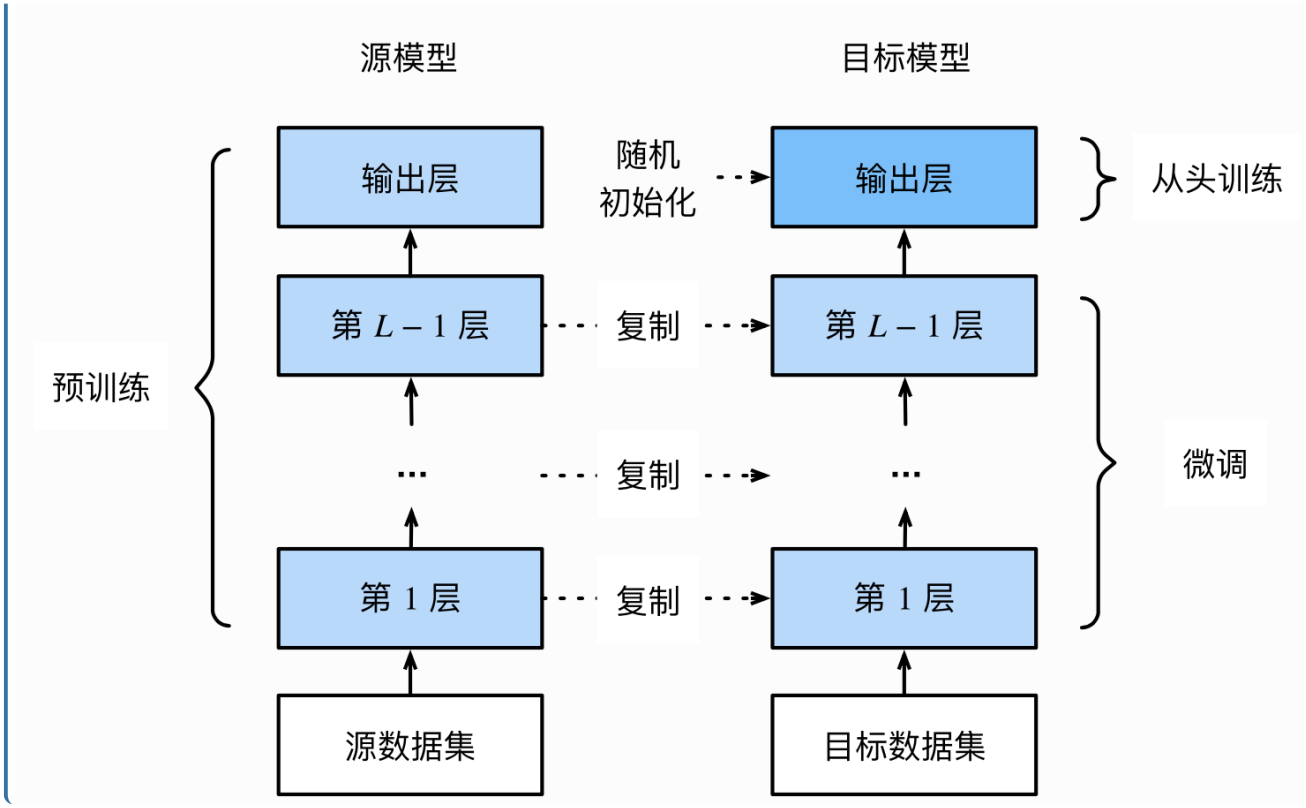
方法是自定义函数`adjust_learning_rate`来改变`param_group`中`lr`的值。以需要学习率每30轮下降为原来的1/10为例。

```
def adjust_learning_rate(optimizer, epoch):
    lr = args.lr * (0.1 ** (epoch // 30))
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
```

模型微调-torchvision

模型微调的流程

1. 在源数据集(如ImageNet数据集)上预训练一个神经网络模型，即源模型。
2. 创建一个新的神经网络模型，即目标模型。它复制了源模型上除了输出层外的所有模型设计及其参数。我们假设这些模型参数包含了源数据集上学习到的知识，且这些知识同样适用于目标数据集。我们还假设源模型的输出层跟源数据集的标签紧密相关，因此在目标模型中不予采用。
3. 为目标模型添加一个输出大小为目标数据集类别个数的输出层，并随机初始化该层的模型参数。
4. 在目标数据集上训练目标模型。我们将从头训练输出层，而其余层的参数都是基于源模型的参数微调得到的。



使用已有模型结构

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
squeezenet = models.squeezenet1_0(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
densenet = models.densenet161(pretrained=True)
inception = models.inception_v3(pretrained=True)
googlenet = models.googlenet(pretrained=True)
shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
mobilenet_v2 = models.mobilenet_v2(pretrained=True)
mobilenet_v3_large = models.mobilenet_v3_large(pretrained=True)
mobilenet_v3_small = models.mobilenet_v3_small(pretrained=True)
resnext50_32x4d = models.resnext50_32x4d(pretrained=True)
wide_resnet50_2 = models.wide_resnet50_2(pretrained=True)
mnasnet = models.mnasnet1_0(pretrained=True)
```

训练特定层

需要通过设置`requires_grad = False`来冻结部分层。

```
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False
```

模型微调 - timm

里面提供了许多计算机视觉的SOTA模型，可以当作是torchvision的扩充版本，并且里面的模型在准确度上也较高。

- Github链接： https://github.com/rwightman/pytorch-image-models
- 官网链接： https://fastai.github.io/timmdocs/
  https://rwightman.github.io/pytorch-image-models/

查看预训练模型种类

`timm.list_models()`

1. 查看特定模型的所有种类

```
all_densnet_models = timm.list_models("*densenet*")
all_densnet_models

['densenet121',
 'densenet121d',
 'densenet161',
 'densenet169',
 'densenet201',
 'densenet264',
 'densenet264d_iabn',
 'densenetblur121d',
 'tv_densenet121']
```

2. 查看模型的具体参数
   过访问模型的`default_cfg`属性来进行查看

```
model = timm.create_model('resnet34',num_classes=10,pretrained=True)
model.default_cfg
```

```
{'url': 'https://github.com/rwightman/pytorch-image-
models/releases/download/v0.1-weights/resnet34-43635321.pth',
 'num_classes': 1000,
 'input_size': (3, 224, 224),
 'pool_size': (7, 7),
 'crop_pct': 0.875,
 'interpolation': 'bilinear',
 'mean': (0.485, 0.456, 0.406),
 'std': (0.229, 0.224, 0.225),
 'first_conv': 'conv1',
 'classifier': 'fc',
 'architecture': 'resnet34'}
```

使用和修改预训练模型

- 创建模型
  通过`timm.create_model()`的方法来进行模型的创建。
- 修改模型

```
model = timm.create_model('resnet34',num_classes=10,pretrained=True)
x = torch.randn(1,3,224,224)
output = model(x)
output.shape
```

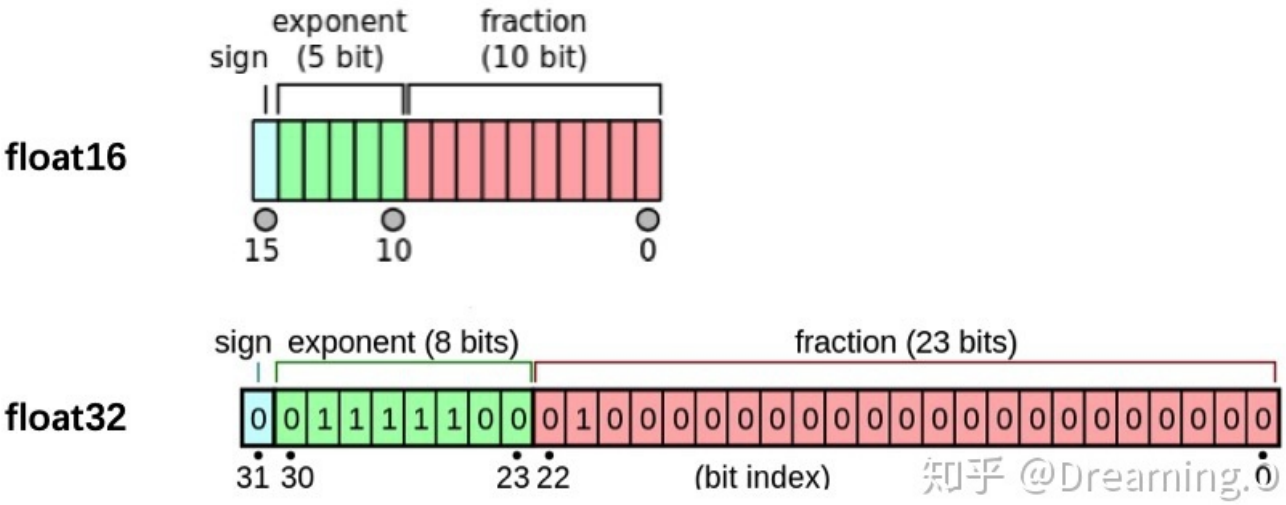
模型的保存

```
torch.save(model.state_dict(),'./checkpoint/timm_model.pth')
model.load_state_dict(torch.load('./checkpoint/timm_model.pth'))
```

半精度训练

使用场景：适用于数据本身的size比较大（比如说3D图像、视频等）

PyTorch默认的浮点数存储方式用的是`torch.float32`，小数点后位数更多固然能保证数据的精确性，但绝大多数场景其实并不需要这么精确，只保留一半的信息也不会影响结果，也就是使用`torch.float16`格式。由于数位减了一半，因此被称为“半精度”。



半精度训练的设置

在PyTorch中使用autocast配置半精度训练，同时需要在下面三处加以设置：

- **import autocast**

```
from torch.cuda.amp import autocast
```

- **模型设置**
  使用python的装饰器方法，用autocast装饰模型中的forward函数。

```
@autocast()
def forward(self, x):
    ...
    return x
```

- **训练过程**

```
for x in train_loader:
        x = x.cuda()
        with autocast():
        output = model(x)
        ...
```

数据增强-imgaug

imgaug简介

`imgaug`是计算机视觉任务中常用的一个数据增强的包，相比于`torchvision.transforms`，它提供了更多的数据增强方法，因此在各种竞赛中，人们广泛使用`imgaug`来对数据进行增强操作。

1. Github地址：imgaug
2. Readthedocs：imgaug
3. 官方提供notebook例程：notebook

imgaug的使用

建议使用`imageio`进行读入图像，再使用`imgaug`操作，如果使用的是opencv进行文件读取的时候，需要进行手动改变通道，将读取的BGR图像转换为RGB图像。除此以外，当我们用PIL.Image进行读取时，因为读取的图片没有shape的属性，所以我们需要将读取到的img转换为np.array()的形式再进行处理。

单张图片处理

可能对一张图片做多种数据增强处理。这种情况下，我们就需要利用`imgaug.augmenters.Sequential()`来构造我们数据增强的pipline。

```
iaa.Sequential(children=None, # Augmenter集合
               random_order=False, # 是否对每个batch使用不同顺序的Augmenter
list
               name=None,
               deterministic=False,
               random_state=None)
# 构建处理序列
aug_seq = iaa.Sequential([
    iaa.Affine(rotate=(-25,25)),
    iaa.AdditiveGaussianNoise(scale=(10,60)),
    iaa.Crop(percent=(0,0.2))
])
# 对图片进行处理，image不可以省略，也不能写成images
image_aug = aug_seq(image=img)
ia.imshow(image_aug)
```

对批次图片进行处理

对批次的图片以同一种方式处理

将待处理的图片放在一个`list`中，并将image改为image即可进行数据增强操作。

对批次的图片分部分处理

可以通过`imgaug.augmenters.Sometimes()`对batch中的一部分图片应用一部分Augmenters，剩下的图片应用另外的Augmenters。

对不同大小的图片进行处理

学习其他数据增强库

Albumentations，Augmentor，imgaug

使用argparse进行调参

argparse简介

这个库可以让我们直接在命令行中就可以向程序中传入参数。

argparse的使用

- 创建`ArgumentParser()`对象
- 调用`add_argument()`方法添加参数
- 使用`parse_args()`解析参数
  例子：argparse模块用法实例详解