

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

扫码关注公众号：【CV技术指南】，专注于计算机视觉的技术总结、论文解读、招聘信息发布等。



扫码加入知识星球：【CV技术指南】。

CV技术指南旨在打造一个完善的计算机视觉知识体系，为入门的人提供服务，为工作的人提供平台。自星球成立以来，星球内有为星友们提供了许多计算机视觉方面的资料，有理论知识，有部署经验，也有论文解读。其中包括目标检测中Head、Neck的设计系列、YOLO系列、anchor-free系列、小目标检测系列、目标检测中的Label Assignment系列、传统目标检测系列、视频中的目标检测系列、图像分割系列、部署系列、CUDA教程系列、论文解读系列、Pytorch源码解读系列、Pytorch实践教程系列。后续还会继续补充~...



QQ群：444129970。QQ群专注于计算机视觉的算法、技术、学习、工作、求职等方面的交流，群内交流氛围极好，有专门的大佬维护，基本99%的提问都会解答。群文件内有很多电子版资源，可供大家免费下载，也欢迎其他人一起上传新资料。



对于初学者，如果需要进行入门辅导，可以看一下下面的海报。

# 计算机视觉入门 1v1辅导班

CV技术指南

## 课程内容

深度学习、机器学习基础、数字图像处理  
pytorch、目标检测、英文文献阅读  
计算机视觉进阶、日后的技术或科研发展路线

## 课程目标

1. 掌握计算机视觉的知识体系
2. 具备搭建模型、分析模型的能力
3. 具备直接阅读英语论文的能力
4. 形成对计算机视觉的理解能力
5. 掌握计算机视觉的正确学习方法
6. 具备独立做科研或项目的能力

## 课程优势

1. 1v1辅导，任何疑问都可以解答到弄懂为止。
2. 由浅及深、循序渐进的学习路线。
3. 全面的计算机视觉知识体系。
4. 阶段考核、进度督促，保证学习效率效果。
5. 注重传授学习方法、而非内容。
6. 根据个人时间灵活制定学习计划与辅导时长
7. 免费加入知识星球一年，享受所有技术教程

## 咨询与报名

扫描右方二维码，备注“入门辅导”，了解详细情况



## (零)概述

浮躁是人性的一个典型的弱点，很多人总擅长看别人分享的现成代码解读的文章，看起来学会了好多东西，实际上仍然不具备自己从零搭建一个pipeline的能力。

在公众号(CV技术指南)的交流群里，常有不少人问到一些问题，根据这些问题明显能看出是对pipeline不了解，却已经在搞项目或论文了，很难想象如果基本的pipeline都不懂，如何分析代码问题所在？如何分析结果不正常的可能原因？遇到问题如何改？

Pytorch在这几年逐渐成为了学术上的主流框架，其具有简单易懂的特点。网上有很多pytorch的教程，如果是一个已经懂的人去看这些教程，确实pipeline的要素都写到了，感觉这教程挺不错的。但实际上更多地像是写给自己看的一个笔记，记录了pipeline要写哪些东西，却没有介绍要怎么写，为什么这么写，刚入门的小白看的时候容易云里雾里。

鉴于此，本教程尝试对于pytorch搭建一个完整pipeline写一个比较明确且易懂的说明。

本教程将介绍以下内容：

## (零)概述

### (一) 数据读取

classdataset的定义

init函数

getitem 函数

len函数

验证classdataset

分布式训练的数据加载方式

数据读取的完整流程

超大数据集的加载思路

问题所在

思路

### (二) 搭建网络

搭建CNN网络

init函数

forward函数

初始化网络

随机初始化

加载预训练模型初始化

搭建Transformer网络

分块

直接分割

卷积分割

Position Embedding

Encoder

Multi-head Self-attention

FeedForward

分类方法

数据的变换

### (三)编写训练过程

参数解析

yaml文件解析

argparser解析

训练日志的配置

设置随机数种子

模型浮动的原因

设置随机数种子的方法

classdataset初始化  
网络的初始化  
学习率的设置  
    学习率调整的方式  
    损失函数的设置  
    TensorboardX的配置

        配置

        显示

    训练过程的搭建  
    断点训练

#### (四) 推理过程

    读取数据  
        单张图片  
        读取监控或视频  
    推理函数

#### (五) 单机多卡训练和多机多卡训练

    单机单卡训练  
    单机多卡训练  
        nn.DataParallel(DP)  
        DDP方式  
    多机多卡训练  
        Horovod

#### (六) 半精度训练和混合精度训练

    Pytorch中的AMP  
    为什么要使用AMP?  
    FP16的优势  
    FP16潜在的问题  
    以上问题的解决办法  
    AMP的工作流程  
    AMP的具体使用

        autocast

        FP32权重备份

        GradScaler

#### (七) 特征图可视化

    写在前面的话  
    初始化配置  
        加载数据并预处理  
        修改网络  
        定义网络并加载预训练模型  
    可视化特征图  
        双线性插值  
        main函数流程  
        可视化效果图

#### (八) 热力图可视化

    热力图可视化方法的原理  
        CAM  
        GradCAM

GradCAM的使用教程

    使用流程

    数据预处理

    GradCAM

其它类型任务的热力图可视化

(九) Pytorch实践部分

    冻结、微调网络

        前言

        迁移学习

        模型微调

        需要微调的情况

        微调的步骤

            步骤示例一

            步骤示例二

        参数冻结

        冻结的方式

            方式一

            方式二

        修改模型参数

        修改模型结构

        文末

NMS

    前言

    NMS原理

    NMS流程

    代码实现

        使用Pytorch自己实现NMS

        调用封装好的NMS

    NMS有哪些缺陷

    IOU

    GIOU

    DIOU

    CIOU

    IOU方法总结

    NMS的改进方法

        soft-nms

        softer-nms

        DIOU-nms

        adaptive nms

        weighted nms

    文末

参数量、计算量统计

    前言

    param.numel()

    thop的profile

Pytorch中查看模型结构、模型参数的函数

    model.state\_dict()

model.modules()  
model.children()  
model.parameters()  
model.named\_modules()、model.named\_children()和model.named\_parameters()

参数量  
FLOPS和FLOPs  
浮点运算量和参数量的区别  
GFLOP  
高效设计网络的准则  
文末

Visdom可视化  
前言  
安装  
服务开启  
    python启动服务  
    浏览器访问  
基本概念  
    Enviroment  
    Window  
    Filter  
使用Visdom  
多Enviroment多窗口管理  
监听Loss等曲线  
热力图可视化  
API接口  
写在后面

数据增强可视化  
1.对图片进行一定比例缩放  
2.对图片进行随机位置的截取  
3.对图片进行随机的水平和竖直翻转  
4.对图片进行随机角度的旋转  
5.对图片进行亮度、对比度和颜色的随机变化  
6.组合使用

目标检测数据集格式转换  
前言  
数据集格式介绍  
    COCO  
    PASCAL VOC  
数据集格式转换  
    From VOC to COCO  
    自定义格式数据集 to COCO  
写在后面

(十一) 自定义损失函数、添加或设计模块  
    自定义损失函数  
    添加或设计模块

(十二) 完整pipeline项目

## (一) 数据读取

本文介绍了classdataset的几个要点，由哪些部分组成，每个部分需要完成哪些事情，如何进行数据增强。然后，介绍了分布式训练的数据加载方式，数据读取的整个流程，当面对超大数据集时，内存不足的改进思路。

### classdataset的定义

先来看一个完整的classdataset

```
import torch.utils.data as data
import torchvision.transforms as transforms

class MyDataset(data.Dataset):
    def __init__(self,data_folder,opt):
        self.data_folder = data_folder
        self.filenames = []
        self.labels = []
        self.opt = opt
        per_classes = os.listdir(data_folder)
        for per_class in per_classes:
            per_class_paths = os.path.join(data_folder, per_class)
            label = torch.tensor(int(per_class))

            per_datas = os.listdir(per_class_paths)
            for per_data in per_datas:
                self.filenames.append(os.path.join(per_class_paths,
per_data))
                self.labels.append(label)

    def __getitem__(self, index):
        image = Image.open(self.filenames[index])
        label = self.labels[index]
        data = self.proprecess(image)
        return data, label

    def __len__(self):
        return len(self.filenames)

    def proprecess(self,data):
        transform_train_list = [
            transforms.Resize((self.opt.h, self.opt.w), interpolation=3),
```

```

        transforms.Pad(self.opt.pad, padding_mode='edge'),
        transforms.RandomCrop((self.opt.h, self.opt.w)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    ]
    data_transform = transforms.Compose(transform_train_list)
    return data_transform(data)

```

classdataset的几个要点：

1. classdataset类继承import torch.utils.data.dataset。
2. classdataset的作用是将任意格式的数据，通过读取、预处理或数据增强后以tensor的形式输出。其中任意格式的数据指可能是以文件夹名作为类别的形式、或以txt文件存储图片地址的形式、或视频、或十几帧图像作为一份样本的形式。而输出则指的是经过处理后的一个batch的tensor格式数据和对应标签。
3. classdataset主要有三个函数要完成：init函数、getitem函数和len函数。

## init函数

init函数主要是完成两个静态变量的赋值。一个是用于存储所有数据路径的变量，变量的每个元素即为一份训练样本，（注：如果一份样本是十几帧图像，则变量每个元素存储的是这十几帧图像的路径），可以命名为self.filenames。一个是用于存储与数据路径变量一一对应的标签变量，可以命名为self.labels。

假如数据集的格式如下：

```

#这里的0, 1指的是类别0, 1
/data_path/0/image0.jpg
/data_path/0/image1.jpg
/data_path/0/image2.jpg
/data_path/0/image3.jpg
.....
/data_path/1/image0.jpg
/data_path/1/image1.jpg
/data_path/1/image2.jpg
/data_path/1/image3.jpg

```

可通过per\_classes = os.listdir(data\_path) 获得所有类别的文件夹，在此处per\_classes的每个元素即为对应的数据标签，通过for遍历per\_classes即可获得每个类的标签，将其转换成int的tensor形式即可。在for下获得每个类下每张图片的路径，通过self.join获得每份样本的路径，通过append添加到self.filenames中。

## getitem 函数

getitem 函数主要是根据索引返回对应的数据。这个索引是在训练前通过dataloader切片获得的，这里先不管。它的参数默认是index，即每次传回在init函数中获得的所有样本中索引对应的数据和标签。因此，可通过下面两行代码找到对应的数据和标签。

```
image = Image.open(self.filenames[index]))  
label = self.labels[index]
```

获得数据后，进行数据预处理。数据预处理主要通过 torchvision.transforms 来完成，这里面已经包含了常用的预处理、数据增强方式。其完整使用方式在官网有详细介绍：<https://pytorch.org/vision/stable/transforms.html>

上面这里介绍了最常用的几种，主要就是resize，随机裁剪，翻转，归一化等。

最后通过transforms.Compose(transform\_train\_list)来执行。

除了这些已经有的数据增强方式外，在《[数据增强方法总结](#)》中还介绍了十几种特殊的数据增强方式，像这种自己设计了一种新的数据增强方式，该如何添加进去呢？

下面以随机擦除作为例子。

```
class RandomErasing(object):  
    """ Randomly selects a rectangle region in an image and erases its  
    pixels.  
    'Random Erasing Data Augmentation' by Zhong et al.  
    See https://arxiv.org/pdf/1708.04896.pdf  
    Args:  
        probability: The probability that the Random Erasing operation will  
        be performed.  
        sl: Minimum proportion of erased area against input image.  
        sh: Maximum proportion of erased area against input image.  
        r1: Minimum aspect ratio of erased area.  
        mean: Erasing value.  
    """  
  
    def __init__(self, probability=0.5, sl=0.02, sh=0.4, r1=0.3, mean=[0.4914, 0.4822, 0.4465]):  
        self.probability = probability  
        self.mean = mean  
        self.sl = sl  
        self.sh = sh  
        self.r1 = r1  
  
    def __call__(self, img):  
        if random.uniform(0, 1) > self.probability:  
            return img  
        for attempt in range(100):
```

```

area = img.size()[1] * img.size()[2]
target_area = random.uniform(self.sl, self.sh) * area
aspect_ratio = random.uniform(self.r1, 1 / self.r1)
h = int(round(math.sqrt(target_area * aspect_ratio)))
w = int(round(math.sqrt(target_area / aspect_ratio)))
if w < img.size()[2] and h < img.size()[1]:
    x1 = random.randint(0, img.size()[1] - h)
    y1 = random.randint(0, img.size()[2] - w)
    if img.size()[0] == 3:
        img[0, x1:x1 + h, y1:y1 + w] = self.mean[0]
        img[1, x1:x1 + h, y1:y1 + w] = self.mean[1]
        img[2, x1:x1 + h, y1:y1 + w] = self.mean[2]
    else:
        img[0, x1:x1 + h, y1:y1 + w] = self.mean[0]
return img
return img

```

如上所示，自己写一个类RandomErasing，继承object，在call函数里完成你的操作。在transform\_train\_list里添加上RandomErasing的定义即可。

```

transform_train_list = [
    transforms.Resize((self.opt.h, self.opt.w), interpolation=3),
    transforms.Pad(self.opt.pad, padding_mode='edge'),
    transforms.RandomCrop((self.opt.h, self.opt.w)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    RandomErasing(probability=self.opt.erasing_p, mean=[0.0, 0.0,
0.0])
    #添加到这里
]

```

## len函数

len函数主要就是返回数据长度，即样本的总数量。前面介绍了self.filenames的每个元素即为每份样本的路径，因此，self.filename的长度就是样本的数量。通过return len(self.filenames)即可返回数据长度。

## 验证classdataset

```
train_dataset = My_Dataset(data_folder=data_folder)
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=False)
print('there are total %s batches for train' % (len(train_loader)))

for i,(data,label) in enumerate(train_loader):
    print(data.size(),label.size())
```

## 分布式训练的数据加载方式

前面介绍的是单卡的数据加载，实际上分布式也是这样，但为了高速高效读取，每张卡上也会保存所有数据的信息，即self.filenames和self.labels的信息。只是在DistributedSampler中会给每张卡分配互不交叉的索引，然后由torch.utils.data.DataLoader来加载。

```
dataset = My_Dataset(data_folder=data_folder)
sampler = DistributedSampler(dataset) if is_distributed else None
loader = DataLoader(dataset, shuffle=(sampler is None), sampler=sampler)
```

## 数据读取的完整流程

结合上面这段代码，在这里，我们介绍以下读取数据的整个流程。

1. 首先定义一个classdataset，在初始化函数里获得所有数据的信息。
2. classdataset中实现getitem函数，通过索引来获取对应的数据，然后对数据进行预处理和数据增强。
3. 在模型训练前，初始化classdataset，通过DataLoader来加载数据，其加载方式是通过DataLoader中分配的索引，调用getitem函数来获取。关于索引的分配，在普通的单卡上，可通过设置shuffle=True来随机生成索引顺序；在多机多卡的分布式训练上，shuffle操作通过DistributedSampler来完成，因此shuffle与sampler只能有一个，另一个必须为None。

## 超大数据集的加载思路

### 问题所在

在提出超大数据集的加载思路之前，有必要先弄清楚，一个超大数据集使用上面的方式读取时，可能会在哪些地方出现问题，知道问题所在才能知道如何改进。

再回顾一下上面这个流程，前面提到所有数据信息在classdataset初始化部分都会保存在变量中，因此当面对超大数据集时，会出现内存不足的情况。

### 思路

将切片获取索引的步骤放到classdataset初始化的位置，此时每张卡都是保存不同的数据子集。通过这种方式，可以将内存用量减少到原来的world\_size倍(world\_size指卡的数量)。

### 参考代码

```

class RankDataset(Dataset):
    ...
    实际流程
    获取rank和world_size 信息 -> 获取dataset长度 -> 根据dataset长度产生随机
    indices ->
        给不同的rank 分配indices -> 根据这些indices产生metas
    ...
    def __init__(self, meta_file, world_size, rank, seed):
        super(RankDataset, self).__init__()
        random.seed(seed)
        np.random.seed(seed)
        self.world_size = world_size
        self.rank = rank
        self.metas = self.parse(meta_file)

    def parse(self, meta_file):
        dataset_size = self.get_dataset_size(meta_file)
            # 获取metafile的行数
        local_rank_index = self.get_local_index(dataset_size, self.rank,
        self.world_size)    # 根据world size和rank, 获取当前epoch, 当前rank需要训练的
        index。
        self.metas = self.read_file(meta_file, local_rank_index)

    def __getitem__(self, idx):
        return self.metas[idx]

    def __len__(self):
        return len(self.metas)

##train
for epoch_num in range(epoch_num):
    dataset = RankDataset("/path/to/meta", world_size, rank, seed=epoch_num)
    sampler = RandomSampler(dataset)
    dataloader = DataLoader(
        dataset=dataset,
        batch_size=32,
        shuffle=False,
        num_workers=4,
        sampler=sampler)

```

但这种思路比较明显的问题时，为了让每张卡上在每个epoch都加载不同的训练子集，因此需要在每个epoch重新build dataloader。

这一节参考链接：<https://zhuanlan.zhihu.com/p/357809861>

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## (二) 搭建网络

搭建网络是一个比较简单的过程，这里会稍微介绍几种常见的方式，介绍初始化方法，模型的保存，然后会介绍一些注意事项，最后会介绍加载预训练模型的原理，以及加载预训练模型指定层参数的方法。

### 搭建CNN网络

首先来看一个CNN网络(以YOLO\_v1的一部分分层为例)。

```
class Flatten(nn.Module):
    def __init__(self):
        super(Flatten, self).__init__()
    def forward(self, x):
        return x.view(x.size(0), -1)

class Yolo_v1(nn.Module):
    def __init__(self, num_class):
        super(Yolo_v1, self).__init__()
        C = num_class
        self.conv_layer1=nn.Sequential(
            nn.Conv2d(in_channels=3,out_channels=64,kernel_size=7,stride=1,padding=7//2),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.1),
            nn.MaxPool2d(kernel_size=2,stride=2)
        )
        self.conv_layer2=nn.Sequential(
            nn.Conv2d(in_channels=64,out_channels=192,kernel_size=3,stride=1,padding=3//2),
            nn.BatchNorm2d(192),
            nn.LeakyReLU(0.1),
            nn.MaxPool2d(kernel_size=2,stride=2)
        )
        #为了简便，这里省去了很多层
        self.flatten = Flatten()
        self.conn_layer1 = nn.Sequential(
```

```

        nn.Linear(in_features=7*7*1024,out_features=4096),
        nn.Dropout(0.5),
        nn.LeakyReLU(0.1)
    )
    self.conn_layer2 =
nn.Sequential(nn.Linear(in_features=4096,out_features=7*7*(2*5 + C)))

    self._initialize_weights()

def forward(self,input):
    conv_layer1 = self.conv_layer1(input)
    conv_layer2 = self.conv_layer2(conv_layer1)
    flatten = self.flatten(conv_layer2)
    conn_layer1 = self.conn_layer1(flatten)
    output = self.conn_layer2(conn_layer1)
    return output

def _initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
            m.weight.data.normal_(0, math.sqrt(2. / n))
            if m.bias is not None:
                m.bias.data.zero_()
        elif isinstance(m, nn.BatchNorm2d):
            m.weight.data.fill_(1)
            m.bias.data.zero_()
        elif isinstance(m, nn.Linear):
            m.weight.data.normal_(0, 0.01)
            m.bias.data.zero_()

```

搭建网络有几个要点：

1. 自定义类要继承torch.nn.Module。有时候自己设计了一些模块，为了使用更方便，通常额外定义一个类，就像这里的Flatten，自定义的类也要继承torch.nn.Module。
2. 完成init函数和forward函数。其中**init函数完成网络的搭建，forward函数完成网络的前传路径。**
3. 完成所有层的参数初始化，一般只有卷积层，归一化层，全连接层要初始化，池化层没有参数。

## init函数

构建网络层有几种方式，一种是pytorch官方已经有了定义的网络，如resnet，vgg，Inception等。一种是自定义层，例如自己设计了一个新的模块。

首先是使用pytorch官方库已经支持的网络，这些网络放在了torchvision.models中，下面选择自己需要的一个。

以下只列举了2D模型的一部分，还有视频类的3D模型。

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained = True)
alexnet = models.alexnet()
vgg16 = models.vgg16()
squeezenet = models.squeeze1_0()
densenet = models.densenet161()
inception = models.inception_v3()
googlenet = models.googlenet()
shufflenet = models.shufflenet_v2_x1_0()
mobilenet_v2 = models.mobilenet_v2()
mobilenet_v3_large = models.mobilenet_v3_large()
mobilenet_v3_small = models.mobilenet_v3_small()
resnext50_32x4d = models.resnext50_32x4d()
wide_resnet50_2 = models.wide_resnet50_2()
mnasnet = models.mnasnet1_0()
efficientnet_b0 = models.efficientnet_b0()
efficientnet_b1 = models.efficientnet_b1()
efficientnet_b2 = models.efficientnet_b2()
regnet_y_400mf = models.regnet_y_400mf()
regnet_y_800mf = models.regnet_y_800mf()
vit_b_16 = models.vit_b_16()
vit_b_32 = models.vit_b_32()
vit_l_16 = models.vit_l_16()
vit_l_32 = models.vit_l_32()
convnext_tiny = models.convnext_tiny()
convnext_small = models.convnext_small()
convnext_base = models.convnext_base()
convnext_large = models.convnext_large()
```

若需要加载该网络在ImageNet上预训练的模型，则在括号内设置参数pretrained=True即可。但这种方式有个不好的问题在于这些预训练模型并不是在本地，因此每次运行都会从网上读取加载模型，非常浪费时间。因此，可以去它官网(<https://pytorch.org/>)上把那个模型下载到本地，通过下面指令完成加载。

```
resnet50.load_state_dict(torch.load('/path/to/resnet50.pth'))
```

另一种自定义层的，一般可以通过torch.nn.Sequential()来构建，在中间插入卷积层、归一化层、激活函数层、池化层即可。

例如下方这种是最常用的。

```

self.conv_layer1=nn.Sequential(
    nn.Conv2d(in_channels=3,out_channels=64,kernel_size=7,stride=1,padding=7//2),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.1),
    nn.MaxPool2d(kernel_size=2,stride=2),
    nn.Conv2d(in_channels=3,out_channels=64,kernel_size=7,stride=1,padding=7//2),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.1),
    nn.MaxPool2d(kernel_size=2,stride=2),
)

```

当网络很深时，上面这种方式构建比较麻烦，例如resnet，总不可能就按找上面这种方式这么写50层。就把它们共同的部分给构建出来，然后通过传参来设置不同的层。

例如：

1.下面这里先构建一个基本的几层作为一个类，每一层的参数(不同输入输出通道数，卷积核大小，有无池化)都通过传参来设置。

```

class BasicBlock(nn.Module):
    expansion = 1
    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(x)

```

```
        out += identity
        out = self.relu(out)
    return out
```

2.下面是设置不同的层。注：上面和下面都不是一个完整的代码，只是用来说明这种很多层的构建方式。

```
layers = []
layers.append(block(self.inplanes, planes, stride, downsample, self.groups,
                     self.base_width, previous_dilation, norm_layer))
self.inplanes = planes * block.expansion
for _ in range(1, blocks):
    layers.append(block(self.inplanes, planes, groups=self.groups,
                        base_width=self.base_width, dilation=self.dilation,
                        norm_layer=norm_layer))
return nn.Sequential(*layers)
```

## forward函数

这里就是网络的传播路径了，一般就是一路往下传就是。return的内容就是网络的输出。

```
def forward(self,x):
    x = self.conv_layer1(x)
    x = self.conv_layer2(x)
    x = self.flatten(x)
    x = self.conn_layer1(x)
    output = self.conn_layer2(x)
    return output
```

如果想将中间某几层的输出拿出来，做一下特征金字塔，可以像下面这么写。

```
def forward(self,x):
    conv_layer1 = self.conv_layer1(x)
    conv_layer2 = self.conv_layer2(conv_layer1)
    conv_layer3 = self.conv_layer2(conv_layer2)
    conv_layer4 = self.conv_layer2(conv_layer3)
    FP = self.YourModule(conv_layer1,conv_layer2,conv_layer3,conv_layer4)
    flatten = self.flatten(FN)
    conn_layer1 = self.conn_layer1(flatten)
    output = self.conn_layer2(conn_layer1)
    return output
```

像可视化特征图里，想要可视化某一层的特征图，就可以像下面这么写。

```

def forward(self,x):
    x = self.conv_layer1(x)
    feature = self.conv_layer2(x)
    x = self.flatten(feature)
    x = self.conn_layer1(x)
    output = self.conn_layer2(x)
    return feature,output

```

## 初始化网络

初始化网络是要放在init函数里完成，分为两类，一类是随机初始化，一类是加载预训练模型。

### 随机初始化

关于随机初始化，目前主要有多种方式：Normal Initialization, Uniform Initialization, Xavier Initialization, He Initialization (也称 kaiming Initialization), LeCun Initialization。关于这些初始化方法，可以看这篇文章[《神经网络的初始化方法总结 | 又名“如何选择合适的初始化方法”》](#)。我们一般使用Kaiming Initialization。

下面是一种方式，直接按自定义的方式初始化

```

def _initialize_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
            m.weight.data.normal_(0, math.sqrt(2. / n))
            if m.bias is not None:
                m.bias.data.zero_()
        elif isinstance(m, nn.BatchNorm2d):
            m.weight.data.fill_(1)
            m.bias.data.zero_()
        elif isinstance(m, nn.Linear):
            m.weight.data.normal_(0, 0.01)
            m.bias.data.zero_()

```

也可以选择pytorch实现了的初始化。

```

for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

```

还可以像下面这么写：

```
from torch.nn import init
def weights_init_kaiming(m):
    classname = m.__class__.__name__
    # print(classname)
    if classname.find('Conv') != -1:
        init.kaiming_normal_(m.weight.data, a=0, mode='fan_in') # For old
pytorch, you may use kaiming_normal.
    elif classname.find('Linear') != -1:
        init.kaiming_normal_(m.weight.data, a=0, mode='fan_out')
        init.constant_(m.bias.data, 0.0)
    elif classname.find('BatchNorm2d') != -1:
        init.normal_(m.weight.data, 1.0, 0.02)
        init.constant_(m.bias.data, 0.0)

def weights_init_classifier(m):
    classname = m.__class__.__name__
    if classname.find('Linear') != -1:
        init.normal_(m.weight.data, std=0.001)
        init.constant_(m.bias.data, 0.0)

self.conv_layer1=nn.Sequential(
    nn.Conv2d(in_channels=3,out_channels=64,kernel_size=7,stride=1,padding=7//2),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.1),
    nn.MaxPool2d(kernel_size=2,stride=2)
)

self.conv_layer1.apply(weights_init_kaiming)
```

反正随便选择一种就好。

## 加载预训练模型初始化

加载预训练模型一般是在train文件里写，但有些网络由于是使用现成的backbone网络，例如使用了resnet50，然后后面加了自定义的模块，所以它想要resnet50预训练模型初始化backbone，而其它层做随机初始化，那加载预训练模型就是在网络定义中做的。因此，既然这里提到了初始化，就干脆写在这里。

最简单的就是直接整个模型都加载。

```
resnet50.load_state_dict(torch.load('/path/to/resnet50.pth'))
```

但也有一些情况下，我只想加载其中一部分层的参数。剩下一部分由于已经改变参数了，无法加载预训练模型，所以要选择上面的随机初始化。

这里有必要来说明网络的每一层是如何表示的。下面以一个例子来说明。

```
class Flatten(nn.Module):
    def __init__(self):
        super(Flatten, self).__init__()
    def forward(self, x):
        return x.view(x.size(0), -1)

class YourNet(nn.Module):
    def __init__(self, stride=2, pool='avg'):
        super(YourNet, self).__init__()
        self.resnet50 = models.resnet50(pretrained=False)
        self.model.load_state_dict(torch.load('/path/to/resnet50.pth'))
        self.flatten = Flatten()
        self.conn_layer1 = nn.Sequential(
            nn.Linear(in_features=7 * 7 * 1024, out_features=4096),
            nn.Dropout(0.5),
            nn.LeakyReLU(0.1)
        )
        self.conn_layer2 = nn.Sequential(nn.Linear(in_features=4096,
out_features=7 * 7 * (2 * 5 + 20)))

    def forward(self, x):
        #这里省略

if __name__ == "__main__":
    model = YourNet()
    for name, value in model.named_parameters():
        print(name)
```

这里简单定义了一个网络。在最后面有这两行：

```
for name, value in model.named_parameters():
    print(name)
```

这两行的输出就是打印网络层的名字，实际上加载预训练模型时，也是按照这个名字来加载的。下面是一部分输出。

```
resnet50.conv1.weight
resnet50.bn1.weight
resnet50.bn1.bias
resnet50.layer1.0.conv1.weight
resnet50.layer1.0.bn1.weight
```

```
resnet50.layer1.0.bn1.bias
resnet50.layer1.0.conv2.weight
resnet50.layer1.0.bn2.weight
resnet50.layer1.0.bn2.bias
resnet50.layer1.0.conv3.weight
resnet50.layer1.0.bn3.weight
resnet50.layer1.0.bn3.bias
resnet50.layer1.0.downsample.0.weight
resnet50.layer1.0.downsample.1.weight
resnet50.layer1.0.downsample.1.bias
...
...
resnet50.layer4.2.bn3.weight
resnet50.layer4.2.bn3.bias
resnet50.fc.weight
resnet50.fc.bias
conn_layer1.0.weight
conn_layer1.0.bias
conn_layer2.0.weight
conn_layer2.0.bias
```

在预训练模型中就是这样，有key即为网络层的名字，value即为它们对应的参数。因此，加载预训练模型可以按照下面这种方式加载。

```
pretrained_dict = torch.load('/path/to/resnet50.pth')
pretrained_dict.pop('fc.weight')
pretrained_dict.pop('fc.bias')
#自己的模型参数变量
model_dict = model.state_dict()
#去除一些不需要的参数
pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in
model_dict}

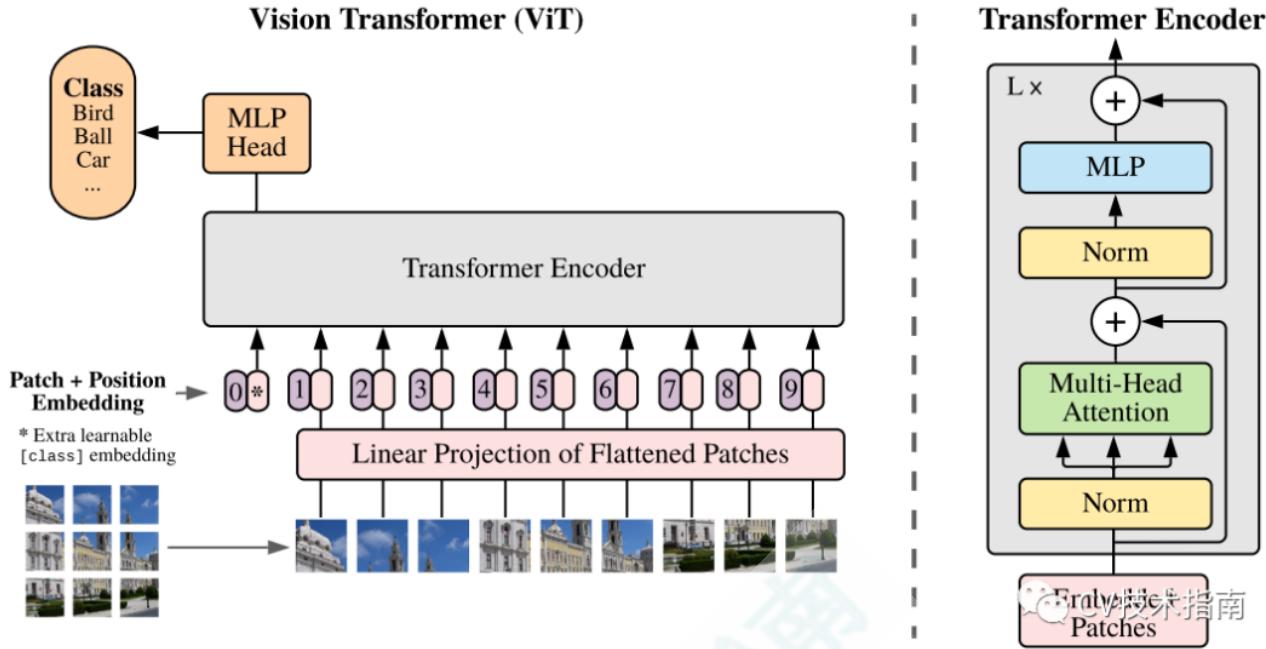
#参数更新
model_dict.update(pretrained_dict)

# 加载我们真正需要的state_dict
model.load_state_dict(model_dict)
```

自己定义的一些层是不会出现在pretrained\_dict中，因此会将其剔除，从而只加载了pretrained\_dict中有的层。

# 搭建Transformer网络

在讲如何搭建之前，先回顾一下Transformer在计算机视觉中的结构是怎样的。这里以最典型的ViT为例。



如图所示，对于一张图像，先将其分割成 $N \times N$ 个patches, 把patches进行Flatten，再通过一个全连接层映射成tokens, 对每一个tokens加入位置编码(position embedding)，会随机初始化一个tokens，concat到通过图像生成的tokens后，再经过transformer的Encoder模块，经过多层Encoder后，取出最后的tokens(即随机初始化的tokens),再通过全连接层作为分类网络进行分类。

下面我们就根据这个流程来一步一步介绍如何搭建一个Transformer模型。、

## 分块

目前有两种方式实现分块，一种是直接分割，一种是通过卷积核和步长都为patch大小的卷积来分割。

### 直接分割

直接分割即把图像直接分成多块。在代码实现上需要使用einops这个库，完成的操作是将 $(B, C, H, W)$ 的shape调整为 $(B, (H/P * W/P), P * P * C)$ 。

```

from einops import rearrange, repeat
from einops.layers.torch import Rearrange

self.to_patch_embedding = nn.Sequential(
    Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 =
patch_height, p2 = patch_width),
    nn.Linear(patch_dim, dim),
)

```

这里简单介绍一下Rearrange。

Rearrange用于对张量的维度进行重新变换排序，可用于替换pytorch中的reshape，view，transpose和permute等操作。举几个例子

```

#假设images的shape为[32, 200, 400, 3]
#实现view和reshape的功能
Rearrange(images, 'b h w c -> (b h) w c')      #shape变为 (32*200, 400, 3)
#实现permute的功能
Rearrange(images, 'b h w c -> b c h w')          #shape变为 (32, 3, 200, 400)
#实现这几个都很难实现的功能
Rearrange(images, 'b h w c -> (b c w) h')        #shape变为 (32*3*400, 200)

```

从这几个例子看可以看出，Rearrange非常简单好用，这里的b, c, h, w都可以理解为表示符号，用来表示操作变化。通过这几个例子似乎也能理解下面这行代码是如何将图像分割的。

```

Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = patch_height, p2 =
patch_width)

```

这里需要解释的是，一个括号内的两个变量相乘表示的是该维度的长度，因此不要把“h”和“w”理解成图像的宽和高。这里实际上 $h = H/p1$ ,  $w = W/p2$ , 代表的是高度上有几块，宽度上有几块。h和w都不需要赋值，代码会自动根据这个表达式计算，b和c也会自动对应到输入数据的B和C。

后面的“b (h w) (p1 p2 c)”表示了图像分块后的shape: (B, (H/P \*W/P), P\*P\*C)

这种方式在分块后还需要通过一层全连接层将分块的向量映射为tokens。

在ViT中使用的就是这种直接分块方式。

## 卷积分割

卷积分割比较容易理解，使用卷积核和步长都为patch大小的卷积对图像卷积一次就可以了。

```
self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size,
stride=patch_size)

x = self.proj(x).flatten(2).transpose(1, 2) # B Ph*Pw C
```

在swin transformer中即使用的是这种卷积分块方式。在swin transformer中卷积后没有再加全连接层。

## Position Embedding

Position Embedding可以分为absolute position embedding和relative position embedding。

在学习最初的transformer时，可能会注意到用的是正余弦编码的方式，但这只适用于语音、文字等1维数据，**图像是高度结构化的数据，用正余弦不合适。**

在ViT和swin transformer中都是直接随机初始化一组与tokens同shape的可学习参数，与tokens相加，即完成了absolute position embedding。

在ViT中实现方式：

```
self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
x += self.pos_embedding[:, :(n + 1)]
#之所以是n+1，是因为ViT中选择随机初始化一个class token，与分块得到的tokens拼接。所以
patches的数量为num_patches+1。
```

在swin transformer中的实现方式：

```
from timm.models.layers import trunc_normal_
self.absolute_pos_embed = nn.Parameter(torch.zeros(1, num_patches,
embed_dim))
trunc_normal_(self.absolute_pos_embed, std=.02)
```

在TimeSformer中的实现方式：

```
self.pos_emb = torch.nn.Embedding(num_positions + 1, dim)
```

以上就是简单的使用方法，这种方法属于absolute position embedding。

还有更复杂一点的方法，以后有机会单独搞一篇文章来介绍。

感兴趣的读者可以先去看看这篇论文《ICCV2021 | Vision Transformer中相对位置编码的反思与改进》。

# Encoder

Encoder由Multi-head Self-attention和FeedForward组成。

## Multi-head Self-attention

Multi-head Self-attention主要是先把tokens分成q、k、v，再计算q和k的点积，经过softmax后获得加权值，给v加权，再经过全连接层。

用公式表示如下：

$$\text{Attention}(q, k, v) = \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)V$$

所谓Multi-head是指把q、k、v在dim维度上分成head份，公式里的dk为每个head的维度。

具体代码如下：

```
class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64, dropout = 0.):
        super().__init__()
        inner_dim = dim_head * heads
        project_out = not (heads == 1 and dim_head == dim)

        self.heads = heads
        self.scale = dim_head ** -0.5

        self.attend = nn.Softmax(dim = -1)
        self.dropout = nn.Dropout(dropout)

        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)

        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim),
            nn.Dropout(dropout)
        ) if project_out else nn.Identity()

    def forward(self, x):
        qkv = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h =
self.heads), qkv)

        dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale

        attn = self.attend(dots)
        attn = self.dropout(attn)

        out = torch.matmul(attn, v)
```

```
    out = rearrange(out, 'b h n d -> b n (h d)')
    return self.to_out(out)
```

这里没有太多可以解释的地方，介绍一下 $q$ 、 $k$ 、 $v$ 的来源，由于这是self-attention，因此 $q=k=v$ (即tokens)，若是普通attention，则 $k=v$ ，而 $q$ 是其它的东西，例如可以是另一个尺度的tokens，或视频领域中的其它帧的tokens。

## FeedForward

这里不用多介绍。

```
class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )
    def forward(self, x):
        return self.net(x)
```

把上面两者组合起来就是Encoder了。

```
class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout = 0.):
        super().__init__()
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                PreNorm(dim, Attention(dim, heads = heads, dim_head =
dim_head, dropout = dropout)),
                PreNorm(dim, FeedForward(dim, mlp_dim, dropout = dropout))
            ]))
    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x
        return x
```

depth指的是Encoder的数量。PreNorm指的是层归一化。

```

class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)

```

## 分类方法

数据通过Encoder后获得最后的预测向量的方法有两种典型。在ViT中是随机初始化一个cls\_token，concat到分块后的token后，经过Encoder后取出cls\_token，最后将cls\_token通过全连接层映射到最后的预测维度。

```

#生成cls_token部分
from einops import repeat
self.cls_token = nn.Parameter(torch.randn(1, 1, dim))

cls_tokens = repeat(self.cls_token, '1 n d -> b n d', b = b)
x = torch.cat((cls_tokens, x), dim=1)
#####
#分类部分
self.mlp_head = nn.Sequential(
    nn.LayerNorm(dim),
    nn.Linear(dim, num_classes)
)
x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

x = self.to_latent(x)
return self.mlp_head(x)

```

在swin transformer中，没有选择cls\_token。而是直接在经过Encoder后将所有数据取了个平均池化，再通过全连接层。

```

self.avgpool = nn.AdaptiveAvgPool1d(1)
self.head = nn.Linear(self.num_features, num_classes) if num_classes > 0 else
nn.Identity()

x = self.avgpool(x.transpose(1, 2)) # B C 1
x = torch.flatten(x, 1)
x = self.head(x)

```

组合以上这些就成了一个完整的模型

```

class ViT(nn.Module):

```

```

    def __init__(self, *, image_size, patch_size, num_classes, dim, depth,
heads, mlp_dim, pool = 'cls', channels = 3, dim_head = 64, dropout = 0.,
emb_dropout = 0.):
        super().__init__()
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)

        num_patches = (image_height // patch_height) * (image_width // patch_width)
        patch_dim = channels * patch_height * patch_width
        assert pool in {'cls', 'mean'}, 'pool type must be either cls (cls token) or mean (mean pooling)'

        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = patch_height, p2 = patch_width),
            nn.Linear(patch_dim, dim),
        )

        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)
        self.transformer = Transformer(dim, depth, heads, dim_head, mlp_dim, dropout)

        self.pool = pool
        self.to_latent = nn.Identity()
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '1 n d -> b n d', b = b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        x = self.dropout(x)
        x = self.transformer(x)
        x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

        x = self.to_latent(x)
        return self.mlp_head(x)

```

# 数据的变换

以上的代码都是比较简单的，整体上最麻烦的地方在于理解数据的变换。

当你想要改进Transformer时，肯定要弄清楚每个维度的数据代表什么，以我目前的经验来看，几乎每个改动，都需要对数据进行维度数量和位置的调整，因此，对数据的形式的了解是非常重要的。

首先输入的数据为  $(B, C, H, W)$ ，在经过分块后，变成了  $(B, n, d)$ 。

在CNN模型中，很好理解(H, W)就是feature map，C是指feature map的数量，那这里的n，d哪个是通道，哪个是图像特征？

回顾一下分块的部分

```
Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = patch_height, p2 = patch_width)
```

根据这个可以知道n为分块的数量，d为每一块的内容。因此，这里的n相当于CNN模型中的C，而d相当于features。

在swin transformer中这种以卷积的形式分块，获得的形式为  $(B, C, L)$ ，然后做了一个 transpose得到  $(B, L, C)$ ，这与ViT通过直接分块方式获得的形式实际上完全一样，在Swin transformer中的L即为ViT中的n，而C为ViT中的d。

因此，要注意的是在 **Multi-head self-attention 中，数据的形式是 (Batchsize, Channel, Features)**，分成多个head的是Features。

前面提到，在ViT中会concat一个随机生成的cls\_token，该cls\_token的维度即为  $(B, 1, d)$ 。可以理解为通道数多了个1。

以上就是Transformer的模型搭建细节了，整体上比较简单，大家看完这篇文章后可以找几篇Transformer的代码来理解理解。如ViT, swin transformer, TimeSformer等。

ViT: [https://github.com/lucidrains/vit-pytorch/blob/main/vit\\_pytorch/vit.py](https://github.com/lucidrains/vit-pytorch/blob/main/vit_pytorch/vit.py)

swin: [https://github.com/microsoft/Swin-Transformer/blob/main/models/swin\\_transformer.py](https://github.com/microsoft/Swin-Transformer/blob/main/models/swin_transformer.py)

TimeSformer: [https://github.com/lucidrains/TimeSformer-pytorch/blob/main/timesformer\\_pytorch/timesformer\\_pytorch.py](https://github.com/lucidrains/TimeSformer-pytorch/blob/main/timesformer_pytorch/timesformer_pytorch.py)

下一篇我们将介绍如何写train函数，以及包括设置优化方式，设置学习率，不同层设置不同学习率，解析参数等。

## (三)编写训练过程

训练过程主要是指编写train.py文件，其中包括参数的解析、训练日志的配置、设置随机数种子、classdataset的初始化、网络的初始化、学习率的设置、损失函数的设置、优化方式的设置、tensorboard的配置、训练过程的搭建等。

本节将逐一介绍以上内容。

### 参数解析

一个模型中包含众多的训练参数，如文件保存目录、数据集目录、学习率、epoch数量、模块中的参数等。

参数解析常用的有两种方式。

一种是将所有参数都放在yaml文件中，通过读取yaml文件来配置参数。这种一般用于比较复杂的项目，例如有多个模型，对应多组参数。这样就可以每个模型配置一个yaml文件，里面对应的是每个模型的对应的参数。

另一种是直接在train.py文件中通过argparser解析器来配置。这种一般用于仅一个模型或比较简单的项目中。每次只需要改一两个参数的。

### yaml文件解析

#### 1. yaml语法规则

大小写敏感

使用缩进表示层级关系

缩进时不允许使用Tab键，只允许使用空格。（可以将你的ide的tab按键输出替换成4个空格）

缩进的空格数目不重要，只要相同层级的元素左侧对齐即可

#表示注释

## 2. yaml文件示例

```
TRAIN:
  RESUME_PATH: "/path/to/your/net.pth"
  DATASET: ucf24 # `ava`, `ucf24` or `jhmdb21`
  BATCH_SIZE: 10
  TOTAL_BATCH_SIZE: 128

SOLVER:
  MOMENTUM: 0.9
  WEIGHT_DECAY: 5e-4
  LR_DECAY_RATE: 0.5
  NOOBJECT_SCALE: 1
  CLASS_SCALE: 1
  COORD_SCALE: 1

DATA:
  TRAIN_JITTER_SCALES: [256, 320]
  TRAIN_CROP_SIZE: 224
  TEST_CROP_SIZE: 224
  MEAN: [0.4345, 0.4051, 0.3775]
  STD: [0.2768, 0.2713, 0.2737]

MODEL:
  NUM_CLASSES: 24
  BACKBONE: darknet

WEIGHTS:
  BACKBONE: "weights/yolo.weights"
  FREEZE_BACKBONE_2D: False

LISTDATA:
  BASE_PTH: "datasets/ucf24"
  TRAIN_FILE: "path/to/your/classdataset/trainlist.txt"
  TEST_FILE: "path/to/your/classdataset/testlist.txt"
  CLASS_NAMES: [
    "Basketball", "BasketballDunk", "Biking", "CliffDiving",
    "CricketBowling",
    "Diving", "Fencing", "FloorGymnastics", "GolfSwing", "HorseRiding",
    "IceDancing", "LongJump", "PoleVault", "RopeClimbing", "SalsaSpin",
    "SkateBoarding", "Skiing", "Skijet", "SoccerJuggling", "Surfing",
    "TennisSwing", "TrampolineJumping", "VolleyballSpiking",
    "WalkingWithDog"
  ].5
```

### 3. yaml的解析

分成两种，一种比较复杂的，像上面这个有两级。解析比较麻烦，代码如下：

```
import yaml
import argparse
from fvcore.common.config import CfgNode
cfg = CfgNode()
cfg.TRAIN= CfgNode()      #每一级都要这样新建一个节点
cfg.TRAIN.RESUME_PATH = "Train"
cfg.TRAIN.DATASET = "ucf24"  # `ava` , `ucf24` or `jhmdb21`
cfg.TRAIN.BATCH_SIZE=10
cfg.TRAIN.TOTAL_BATCH_SIZE=128
...
cfg.SOLVER= CfgNode()    #每一级都要这样新建一个节点
cfg.SOLVER.MOMENTUM=0.9
cfg.SOLVER.WEIGHT_DECAY=5e-4
...
yaml_path = "yaml_test.yaml"
cfg.merge_from_file(yaml_path)

#访问方法
print(cfg.TRAIN.RESUME_PATH)
```

它的麻烦在于需要将所有的元素都初始化一遍，然后通过cfg.merge\_from\_file(yaml\_path)来根据yaml文件更新这些元素。

另一种是比较简单的解析二级的方法。

```
import yaml
with open(yaml_path, 'r') as file:
    opt = yaml.load(file.read(), Loader=yaml.FullLoader)
#访问方法
print(opt['TRAIN']['RESUME_PATH'])
```

这种方式非常简单，相比于前面那种，需要每个都去初始化，简直不要太方便。但这种看着仍然别扭，我不喜欢用字典的访问方式。因此，找了一段时间，找到了一种更好用的方式。

```

import yaml
import argparse
with open(yaml_path, 'r') as file:
    opt = argparse.Namespace(**yaml.load(file.read(), Loader=yaml.FullLoader))
opt.TRAIN = argparse.Namespace(**opt.TRAIN)
opt.SOLVER = argparse.Namespace(**opt.SOLVER)
opt.DATA = argparse.Namespace(**opt.DATA)
opt.MODEL = argparse.Namespace(**opt.MODEL)
opt.WEIGHTS = argparse.Namespace(**opt.WEIGHTS)
opt.WEIGHTS = argparse.Namespace(**opt.WEIGHTS)
opt.LISTDATA = argparse.Namespace(**opt.LISTDATA)
#访问方法
print(opt.TRAIN.RESUME_PATH)

```

这种方式通过argparse.Namespace将字典转换成了空间变量。访问元素的时候就舒服了。

## argparser解析

argparser解析的形式一般放在train.py文件的最前面，适用于参数相对比较少，每次只需要改一两个参数的情况。（当然我本人习惯将它放在其它文件中，例如单独搞一个parser.py或直接放在util.py中，只因为如果放在train前每次都要滑动很长才能到train的部分，相当麻烦）

先来个标准示例

```

import argparse
def get_args():
    parser = argparse.ArgumentParser(description='Training')
    parser.add_argument('--color_jitter', action='store_true', help='use
color jitter in training')
    parser.add_argument('--batchsize', default=8, type=int, help='batchsize')
    parser.add_argument('--gpu_ids', default='0', type=str, help='gpu_ids:
e.g. 0 0,1,2 0,2')
    return parser.parse_args()

#使用方法
python train.py
--color_jitter \
--batchsize=16 \
--gpu_ids='0'

#访问元素
opt = get_args()
print(opt.batchsize)

```

这里列举了三种形式，一种是action的，当action='store\_true'时，默认是false，在设置参数时直接--color\_jitter即可变成True，另外两种如上所示。

## 训练日志的配置

训练日志是用于保存训练过程中的一些信息，方便事后查看模型的训练情况。

首先是准备好基本的配置。

```
import logging
def train_logger(num):
    logger = logging.getLogger(__name__)
    #设置打印的级别，一共有6个级别，从低到高分别为：
    #NOTEST、DEBUG、INFO、WARNING、ERROR、CRITICAL。
    #setLevel设置的是最低打印的级别，低于该级别的将不会打印。
    logger.setLevel(level=logging.INFO)
    #打印到文件，并设置打印的文件名称和路径
    file_log = logging.FileHandler('./run/{}/train.log'.format(num))
    #打印到终端
    print_log = logging.StreamHandler()
    #设置打印格式
    #%(asctime)表示当前时间，%(message)表示要打印的信息，用的时候会介绍。
    formatter = logging.Formatter('%(asctime)s      %(message)s')
    file_log.setFormatter(formatter)
    print_log.setFormatter(formatter)

    logger.addHandler(file_log)
    logger.addHandler(print_log)
    return logger
```

### 使用方法

```
logger = train_logger(0)
logger.info("project_name: {}".format(project_name))
logger.info("batchsize: {}".format(opt.batchsize))
logger.warning("warning message")
```

###

# 设置随机数种子

随机数种子是为了固定数据集每次的打乱顺序，让模型变得可复现。不设置随机数种子的话，由于数据集每次打乱的顺序都不一样，导致模型会略有浮动。

咱们这里稍微介绍一下浮动的原因。

## 模型浮动的原因

目前基本都是使用mini-batch梯度下降，也就是说每次都是前传一个batch的数据后，才会更新权重，与此同时，模型基本都是有使用BN，即对每个batch做归一化。因此，batch数据对模型的性能会有一定的影响。

如果每次随机顺序都不一样，可能会存在某几次的batch组合得非常好，以至于模型训练效果不错，而其它时候的batch的组合不是很合适，以至于达不到组合得很好的时候的效果。

因此，所谓的原因就是每次的不同顺序产生了batch样本的多样性，batch样本多样性对模型的结果有一定的影响。数据集越小，这个影响可能越大，因为造成的batch样本之间的差异性很大，而数据集越大时，batch样本之间的差异性可能受到随机顺序的影响越小。

因此，设置随机数种子时一个必要的事情。

计算机上的随机数都是人工模拟出来的，因此，我们可以任意地设置随机数的范围等。

## 设置随机数种子的方法

```
import random
#基本配置
def setup_seed(seed):
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    random.seed(seed)
    cudnn.deterministic = True
#使用方法
#在train文件下调用下面这行命令，参数随意设置，
#只要这个参数数值一样，每次生成的顺序也就一样
setup_seed(2022)
```

## *classdataset*初始化

*classdataset*初始化比较容易，仅通过以下几行代码即可。

```
from torch.utils.data import DataLoader
from dataloader import MyDataset
train_folder = opt.data_dir + '/train'
train_dataset = MyDataset(data_folder=train_folder,opt=opt)
train_loader = DataLoader(train_dataset, batch_size=opt.batchsize,
shuffle=True, num_workers=8)
```

主要是设置DataLoader，其中shuffler基本默认为True，如果是多级多卡分布式训练，则shuffle为false，而sampler通过下面的代码来获取，num\_workers表示加载数据使用的进程数量。

```
from torch.utils.data.distributed import DistributedSampler
train_sampler =
torch.utils.data.distributed.DistributedSampler(train_dataset)
train_loader = DataLoader(train_dataset, batch_size=opt.batchsize,
sampler=train_sampler)
```

###

## 网络的初始化

网络初始化主要是两件事，一个是初始化网络，另一个是加载预训练模型，由于前面第二篇中我们已经介绍过了如何加载指定层的预训练参数，因此，这里就不多介绍了。这部分也比较简单。

```
from model import Yolo_v1
net = Yolo_v1()
net.load_state_dict(torch.load(trained_path))
net = net.cuda()
```

## 学习率的设置

学习率的设置主要是介绍一下如何在不同的层设定不同的学习率。例如backbone使用的是预训练模型，而全连接层是使用随机初始化的，因此backbone需要小学习率，全连接层需要大学习率。

```
import torch.optim as optim
from torch.optim import lr_scheduler
optim_params = [ {'params': net.backbone.parameters(), 'lr': 0.1 * opt.lr},
                 {'params': net.interaction.parameters(), 'lr': opt.lr},
                 {'params': net.large_conv1_1.parameters(), 'lr': opt.lr},
                 {'params': net.large_conv1_2.parameters(), 'lr': opt.lr},
                 {'params': net.classifier.parameters(), 'lr': opt.lr}
                ]

optimizer = optim.SGD(optim_params, weight_decay=5e-4, momentum=0.9,
nesterov=True)
scheduler = lr_scheduler.StepLR(optimizer, step_size=80, gamma=0.1)
```

首先是构建一个参数和对应学习率的列表，然后作为参数传给optim的这些优化器，例子中用的是SGD。

此外还设置了一个学习率的调度器 lr\_scheduler，用于训练到不同的epoch时调整学习率。这里的step\_size表示在每80个epoch时，学习率乘以gamma值。

还有一个lr\_scheduler.MultiStepLR(optimizer, milestones=[30,50,60], gamma=0.1, last\_epoch=-1)。这与StepLR的区别是MultiStepLR是根据milestones中的时间来调整学习率的，这个例子中表示的是在第30、50、60epoch时调整。

上面这里是学习率的基本配置，下面还涉及到学习率在训练过程中的调整。

## 学习率调整的方式

在PyTorch 1.1.0之前的版本，学习率的调整应该被放在optimizer更新之前。1.1.0版本后用在optimizer更新后。

```
for epoch in range(opt.num_epochs):
    #省略一部分代码
    loss.backward()
    optimizer.step()
    scheduler.step()
```

## 损失函数的设置

通用的损失函数是简单的，直接通过下面几行代码即可。

```
import torch.nn as nn
cls_criterion = nn.CrossEntropyLoss()
dist_criterion = nn.MSELoss() # Use L2 loss function
hinge_criterion = nn.HingeEmbeddingLoss()
```

但有些损失函数是自己设计的，因此需要自己来实现。下面以TripletLoss为例。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class TripletLoss(nn.Module):
    def __init__(self, t1, t2, beta):
        super(TripletLoss, self).__init__()
        self.t1 = t1
        self.t2 = t2
        self.beta = beta
        return

    def forward(self, anchor, positive, negative):
        matched = torch.pow(F.pairwise_distance(anchor, positive), 2)
        mismatched = torch.pow(F.pairwise_distance(anchor, negative), 2)
        part_1 = torch.clamp(matched - mismatched, min=self.t1)
        part_2 = torch.clamp(matched, min=self.t2)
        dist_hinge = part_1 + self.beta * part_2
        loss = torch.mean(dist_hinge)
        return loss
```

简单介绍一下用法，跟定义网络一样，继承nn.Module，然后完成init和forward函数，但与网络不同的是，损失函数中没有可训练的参数，因此通常直接使用torch.nn.functional中的函数即可。

这也是torch.nn.functional中函数与nn中的函数的区别，前者需要自己设置权重，且不会随训练过程更新，而后者不需要自己设置权重，权重会更新。

## TensorboardX的配置

### 配置

tensorboard的使用非常简单。

```
from torch.utils.tensorboard import SummaryWriter
logdir = 'log'
writer = SummaryWriter(log_dir=logdir)

for epoch in range(opt.num_epochs):
    train_acc = ...
    loss = ...
    writer.add_scalar('train/train_acc', train_acc, epoch+1)
    writer.add_scalar('train/loss', loss.item(), epoch+1)
writer.close()
```

add\_scalar主要有三个参数

- tag: 标签, 如下图所示的Train\_loss
- scalar\_value: 标签的值
- global\_step: 标签的x轴坐标

每使用一个add\_scalar函数则会多一个坐标图, 如果有多条曲线想要画在同一个坐标图里, 则可以使用字典的形式。如下所示。注意, 上面用的是add\_scalar, 而这里是复数形式, add\_scalars。

```
writer.add_scalars('Training',
                    {'train_acc': train_acc,
                     'val_acc': val_acc},
                    epoch+1)
```

除了画曲线图, 还可以画一些别的, 例如writer.add\_graph(), writer.add\_image()。

## 显示

上面在代码中配置好后会生成一个logdir文件夹, 里面保存这些writer的数据。通过下面的执行代码来显示。

```
tensorboard --logdir='log'
```

# 训练过程的搭建

上面的部分基本配置和初始化好了所有需要的模块或代码。

下面开始搭建一个train过程。

```

def train(net, criterion, optimizer, scheduler, train_loader, val_loader,
writer):
    for epoch in range(opt.num_epochs - start_epoch):
        #设置start_epoch是因为有时是从恢复训练，不是从零开始的。
        epoch = epoch + start_epoch
        print('Epoch {} / {}'.format(epoch+1, opt.num_epochs))
        print('-' * 10)
        print('lr:{}'.format(optimizer.param_groups[0]['lr']))
        net.train(True)#这一行没写成为很多人训练失败的原因

        train_acc = 0.0
        train_loss = 0.0
        for i, (img_data, label) in enumerate(train_loader):
            if img_data.shape[0] < opt.batchsize: # skip the last batch
                continue
            img_data = Variable(img_data.cuda())
            label = Variable(label.cuda())
            optimizer.zero_grad()

            # forward
            output= net(img_data )
            loss = criterion(pred,label)
            train_loss += loss.data[0]
            pred = output.argmax(dim=1)
            train_acc += float(torch.sum(pred == label.data))

            loss.backward()
            optimizer.step()

        #学习率调整
        scheduler.step()
        #计算准确率，计算损失
        train_acc = train_acc / len(train_loader)
        train_loss = train_loss / len(train_loader)
        writer.add_scalar('Train_loss', train_loss, global_step=epoch)
        writer.add_scalar('Train_acc', train_acc, global_step=epoch)

        if (epoch +1)%5 == 0:
            torch.save({'epoch': epoch,
                       'optimizer_dict': optimizer.state_dict(),
                       'model_dict': model.state_dict()},
                       save_path)
        #验证集
        val(net, criterion, val_loader, writer, epoch)
        writer.close()

def val(net, criterion, val_loader, writer, epoch):

```

```

net = net.eval()
val_acc = 0.0
val_loss = 0.0
with torch.no_grad():
    for i, (img_data, label) in enumerate(val_loader):
        if img_data.shape[0] < opt.batchsize: # skip the last batch
            continue
        img_data = Variable(img_data.cuda())
        label = Variable(label.cuda())

        # forward
        output= net(img_data )
        loss = criterion(pred,label)
        val_loss += loss.data[0]
        pred = output.argmax(dim=1)
        val_acc += float(torch.sum(pred == label.data))

#计算准确率，计算损失
val_acc = val_acc / len(val_loader)
val_loss = val_loss / len(val_loader)
writer.add_scalar('Val_loss', val_loss, global_step=epoch)
writer.add_scalar('Val_acc', val_acc, global_step=epoch)

```

上面是一个基本的训练过程，需要解释的地方并不多，很容易看懂。

1. train\_loader是classdataset中通过getitem()生成的一个迭代器，每次生成一个batch的数据。
2. 准确率的计算是不需要经过softmax的，直接argmax即可。之前星球内有人问为什么不以准确率为优化目标，而要额外设置一个loss，其实就是因为argmax这个过程是不可导的。甚至有人把求loss与求准确率的理论过程给搞混了。

这个代码中，打印训练信息，以及一些其它的东西都没放上去，大家可以根据自己的需要增加。

这个代码是默认使用gpu的，一般来说都会去判断有没有gpu，有的话就gpu训练，没有的话cpu训练。我个人是希望如果没有GPU就直接报错，而不是在cpu上慢吞吞的训练，因此这里没有去判断设备的代码。

## 断点训练

有时候训练到一半，被某些因素给终止了，从零开始训练太浪费时间，想要恢复之前的训练进度继续训练。

首先是保存模型部分

```
torch.save({'epoch': epoch,
            'optimizer_dict': optimizer.state_dict(),
            'model_dict': model.state_dict()},
            save_path)
```

当按照这种模式保存模型时，比较容易恢复训练。

```
def load_model(save_name, optimizer, model):
    model_data = torch.load(save_name)
    start_epoch = model_data['epoch']
    model.load_state_dict(model_data['model_dict'])
    optimizer.load_state_dict(model_data['optimizer_dict'])
    print("model load success")
    return start_epoch, model, optimizer
```

这里的start\_epoch也就对应了train函数中，`epoch=epoch+start_epoch`的部分。

在train文件的main函数中会加上下面的代码。

```
if opt.resume:
    load_model(save_name, optimizer, model)
```

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## (四) 推理过程

推理过程比较简单，其实就跟val函数一样，只不过推理过程需要做后处理和可视化。但不同任务的后处理和可视化都不一样，因此这里不多介绍后处理和可视化。仅介绍一个完整的流程，会包括读取单张图片，读取实时监控。

### 读取数据

推理可能是用于单张图片，也可能是一个实时的监控摄像头，又或者是一堆图片。一堆图片的咱就不介绍了，直接调用classdataset来加载就好。

## 单张图片

读取单张图片就没必要调用前面写的classdataset了，咱们简单一点，往下看。

```
import torch
from PIL import Image
import torchvision.transforms as transforms

def image_proprecess(img_path):
    img = Image.open(img_path)
    data_transforms = transforms.Compose([
        transforms.Resize((384, 384), interpolation=3),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
    data = data_transforms(img)
    data = torch.unsqueeze(data,0)
    return img, data
```

直接读取该图片，并完成初始化。注意，由于是单张图片，因此需要将其变成四维。

## 读取监控或视频

```
import cv2
cap = cv2.VideoCapture(0)
#cap = cv2.VideoCapture("data/monitor.mp4") #读取视频
width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
fps = cap.get(cv2.CAP_PROP_FPS)

#这是为了保存视频，先提前进行的配置
fourcc = cv2.VideoWriter_fourcc('m','p','4','v')
out = cv2.VideoWriter('data/detected_monitor.mp4',fourcc,fps,(width,height))

if not cap.isOpened():
    print("Unable to open camera")
    exit(-1)

#配置图像预处理过程
data_transforms = transforms.Compose([
    transforms.Resize((384, 384), interpolation=3),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

while True:
```

```
res, img = cap.read()
if res:
    data = data_transforms(img)
    data = torch.unsqueeze(data, 0)
    #下面是推理过程,
    pred = inference(data)
    #以目标检测为例, 后处理, 从pred中读出预测框, 这里省略
    bboxes = ...
    #在img原图上画框, 并保存
    draw_img = plot_boxes_cv2(img, bboxes, None, class_names)
    out_img = cv2.resize(draw_img,(width,height))
    out.write(out_img)#保存视频
```

读取数据后就是推理了。

## 推理函数

推理函数中需要加载模型, 推理, 后处理, 可视化就可以了。

```
from model import Yolo_v1
def inference():
    img_path = "dog.png"
    img, data = image_proprecess(img_path)
    data = torch.autograd.Variable(data.cuda())

    net = Yolo_v1()
    net.load_state_dict(torch.load(trained_path))
    net = net.cuda().eval()

    pred = net(data)
    #以目标检测为例, 后处理, 从pred中读出预测框, 这里省略
    bboxes = ...
    #在img原图上画框, 并保存
    draw_img = plot_boxes_cv2(img, bboxes, None, class_names)
    out_img = cv2.resize(draw_img,(width,height))
```

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习, 严禁用于商业行为。

欢迎加入QQ交流群: 444129970。本文档如若任何错误, 请加QQ群联系管理员修改

## (五)单机多卡训练和多机多卡训练

这一部分主要介绍单机多卡训练和多机多卡训练的实现方法和一些注意事项。其中单机多卡训练介绍两种实现方式，一种是DP方式，一种是DDP方式。多机多卡训练主要介绍两种实现方式，一种是通过horovod库，一种是DDP方式。

### 单机单卡训练

前面我们已经介绍了一个完整的训练流程，但这里由于要介绍单机多卡和多机多卡训练的代码，为了能更好地理解它们之间的区别，这里先放一个单机单卡也就是一般情况下的代码流程。

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
train_dataset = ...
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=...)

model = ...
optimizer = optim.SGD(model.parameters())

for epoch in range(opt.num_epoch):
    for i, (input, target) in enumerate(train_loader):
        input= input.to(device)
        target = target.to(device)
        ...
        output = model(input)
        loss = criterion(output, target)
        ...
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

### 单机多卡训练

单机多卡训练的部分有两种实现方式，一种是DP方式，一种是DDP方式。

## nn.DataParallel(DP)

DP方式比较简单，仅仅通过nn.DataParallel对网络进行处理即可。

其它部分基本与单机单卡训练流程相同。

```
import torch

train_dataset = ...
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=...)

model = ...
model = nn.DataParallel(model.to(device), device_ids=None,
output_device=None)
optimizer = optim.SGD(model.parameters())

for epoch in range(opt.num_epoch):
    for i, (input, target) in enumerate(train_loader):
        input= input.cuda()
        target = target.cuda()
        ...
        output = model(input)
        loss = criterion(output, target)
        ...
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

上面唯一关键的一句是在定义model后，使用nn.DataParallel把模型放到各个GPU上。

其中，device\_ids有几种设置方式，如果设置为None，如下几行源码所示，默认使用所有gpu

```
#nn.DataParallel中的源码
if device_ids is None:
    device_ids = list(range(torch.cuda.device_count()))
if output_device is None:
    output_device = device_ids[0]
```

也可以手动指定用哪几个gpu。如下所示

```
gpus = [0, 1, 2, 3]
torch.cuda.set_device('cuda:{}'.format(gpus[0]))
model = nn.DataParallel(model.to(device), device_ids=None,
output_device=gpus[0])
```

## DDP方式

上面DP是比较简单的单机多卡的实现方式，但DDP是更高效的方式，不过实现要多几行代码。

该部分代码由读者投稿，非本人原创。

```
import torch
import argparse
import torch.distributed as dist

parser = argparse.ArgumentParser()
parser.add_argument('--local_rank', default=-1, type=int,
                    help='node rank for distributed training')
opt = parser.parse_args()

# 初始化GPU通信方式（NCCL）和参数的获取方式（env代表通过环境变量）。
dist.init_process_group(backend='nccl', init_method='env://')

torch.cuda.set_device(opt.local_rank)

train_dataset = ...
train_sampler =
torch.utils.data.distributed.DistributedSampler(train_dataset)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=...,
sampler=train_sampler)

# 使用 DistributedDataParallel 包装模型，它能帮助我们为不同 GPU 上求得的梯度进行
# all reduce（即汇总不同 GPU 计算所得的梯度，并同步计算结果）。all reduce 后不同 GPU
# 中模型的梯度均为 all reduce 之前各 GPU 梯度的均值。
model = ...
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_rank])

optimizer = optim.SGD(model.parameters())

for epoch in range(opt.num_epoch):
    for i, (input, target) in enumerate(train_loader):
        input= input.cuda()
        target = target.cuda()
        ...
        output = model(input)
        loss = criterion(output, target)
        ...
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
#运行命令  
CUDA_VISIBLE_DEVICES=0,1,2,3 python -m torch.distributed.launch --  
nproc_per_node=4 train.py
```

下面对这段代码进行解析。

1. 设置local\_rank参数，可以把这个参数理解为进程编号。该参数在运行上面这条指令时就会确定，每块GPU上的该参数都会不一样。
2. 配置初始化方式，一般有tcp方式和env方式。上面是用的env，下面是用tcp方式用法。

```
dist.init_process_group(backend='nccl', init_method='tcp://localhost:23456')
```

3. 通过local\_rank来确定该进程的设备：torch.cuda.set\_device(opt.local\_rank)
4. 数据加载部分我们在该教程的第一篇里介绍过，主要时通过torch.utils.data.distributed.DistributedSampler来获取每个gpu上的数据索引，每个gpu根据索引加载对应的数据，组合成一个batch，与此同时DataLoader里的shuffle必须设置为None。

## 多机多卡训练

多机多卡训练的一般有两种实现方式，一种是上面这个DDP方式，这里我们就不再介绍了，另一种是使用一个额外的库horovod。

### Horovod

Horovod是基于Ring-AllReduce方法的深度分布式学习插件，以支持多种流行架构包括TensorFlow、Keras、PyTorch等。这样平台开发者只需要为Horovod进行配置，而不是对每个架构有不同的配置方法。

来自博客：[https://blog.csdn.net/weixin\\_44388679/article/details/106564349](https://blog.csdn.net/weixin_44388679/article/details/106564349)

该部分代码由读者投稿，非本人原创。

```
import torch  
import horovod.torch as hvd
```

```

hvd.init()

torch.cuda.set_device(hvd.local_rank())

train_dataset = ...
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset, num_replicas=hvd.size(), rank=hvd.rank)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., 
sampler=train_sampler)

model = ...
model.cuda()

optimizer = optim.SGD(model.parameters())
optimizer = hvd.DistributedOptimizer(optimizer,
named_parameters=model.named_parameters())

hvd.broadcast_parameters(model.state_dict(), root_rank=0)

for epoch in range(opt.num_epoch):
    for i, (input, target) in enumerate(train_loader):
        input= input.cuda()
        target = target.cuda()
        ...
        output = model(input)
        loss = criterion(output, target)
        ...
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if hvd.rank()==0:
            print("loss: ")

```

下面对以上代码进行简单的介绍。

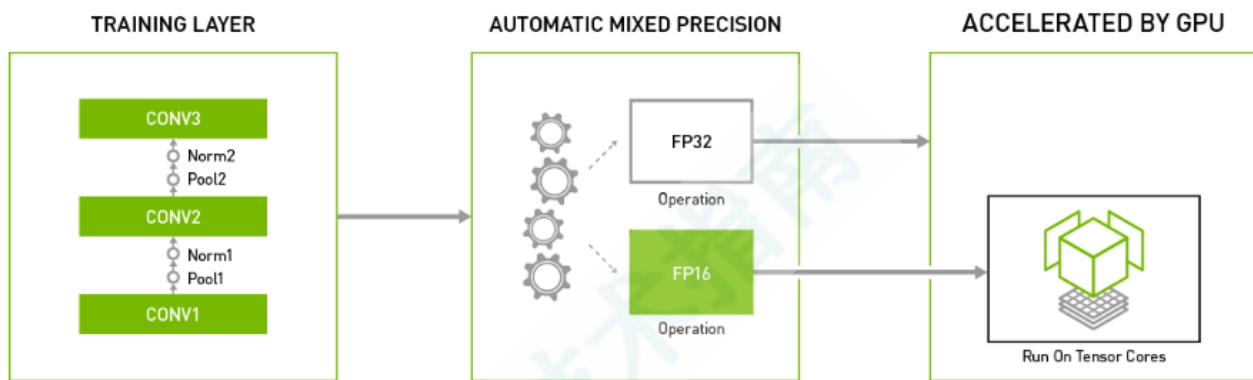
- 与 DDP 相同的是，先初始化，再根据进程设置当前设备，然后使用 `torch.utils.data.distributed.DistributedSampler` 来产生每个GPU读取数据的索引。
- 不同的是接下来几个操作，horovod不需要使用`torch.nn.parallel.DistributedDataParallel`，而是通过使用horovod的两个库，通过`hvd.DistributedOptimizer`和`hvd.broadcast_parameters`分别对优化器和模型参数进行处理。
- 除了训练以外，其它操作基本都在主进程上完成，例如打印信息，保存模型等。通过最后`if hvd.rank()==0`来判定。

除了DDP和horovod这两种方式实现多机多卡以外，实际上在混合精度训练里的库apex也有对应的多机多卡训练实现方式，但这个我们就留到下一篇混合精度训练和半精度训练中来介绍。

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## (六)半精度训练和混合精度训练



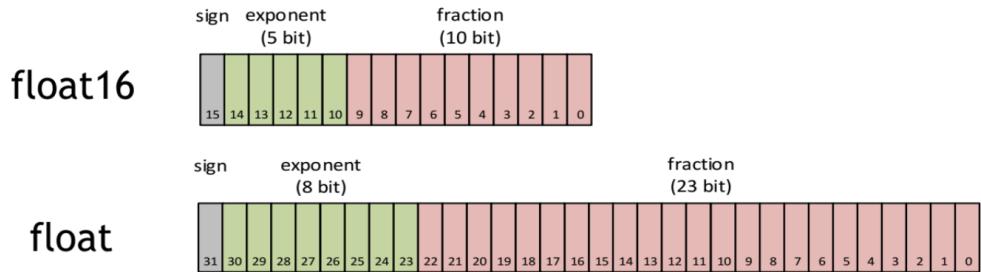
图片来源于[https://mp.weixin.qq.com/s?\\_\\_biz=MzU2NzkyMzUxMw==&mid=2247490135&idx=2&sn=af5a8dc23412db33ce6ca8f3d92f326a&source=41#wechat\\_redirect](https://mp.weixin.qq.com/s?__biz=MzU2NzkyMzUxMw==&mid=2247490135&idx=2&sn=af5a8dc23412db33ce6ca8f3d92f326a&source=41#wechat_redirect)

### Pytorch中的AMP

AMP：是Automatic mixed precision、自动混合精度的意思。在神经网络推理过程中，可以针对不同的层，选取不同的数据精度来进行计算，从而实现节省显卡资源和加快速度。这里所说的不同数据精度，其实就是将FP32单精度和FP16半精度结合在一起，并使用相同的超参数来实现与FP32差不多相同的精度。

在Pytorch1.5以及之前的版本中，是通过NVIDIA提供的apex库来实现amp的功能，但在使用的过程中存在版本兼容的问题。从Pytorch1.6开始，就将AMP集成进了torch.cuda中，支持自动混合精度训练，并且兼容性做得也是比较好，使用AMP后，最多可以节省一般的显存资源。

在使用之前，首先需要判断你的GPU是否支持FP16。



图片来源于<https://blog.csdn.net/junbabab/article/details/119078807>

所谓AMP，关键是自动以及混合精度。

自动：张量的类型会自动变化。框架会根据情况自动调整张量的类型，但是个别地方可能还是需要人为修改。

混合精度：采用不同精度的张量，一般是FP32和FP16这两种精度。

一般来说，大多数深度学习框架如TensorFlow、Pytorch都是采用32位的浮点数来进行训练。与FP32相比，FP16的内存消耗仅为FP32的一半，因此FP16更加适合在移动端侧上进行AI计算。相比FP32来说，占用内存减少了一半，同时也有相应的指令值，所以速度比FP32要快许多。FP16更适合在精度要求不高的场景中。

- FP32指数占8位，尾数占23位，数据的动态范围是 $[2^{-126}, 2^{127}]$ ，是深度学习框架训练时常用的数据类型。
- FP16指数占5位，尾数占10位，相比FP32，FP16的表示范围更窄，最小可表示的正数数值为 $2^{-14}$ ，最大可表示的数据为65504，容易出现数值溢出。

## 为什么要使用AMP?

因为在一些情况下使用FP32有优势，而在另外的一些情况下，FP16有优势。

在PyTorch 1.6的AMP上下文中，如下操作中的Tensor会自动转化为半精度浮点型的格式：

```

__matmul__
addbmm
addmm
addmv
addr
baddbmm
bmm
chain_matmul
conv1d
conv2d
conv3d
conv_transpose1d

```

```
conv_transpose2d  
conv_transpose3d  
linear  
matmul  
mm  
mv  
prelu
```

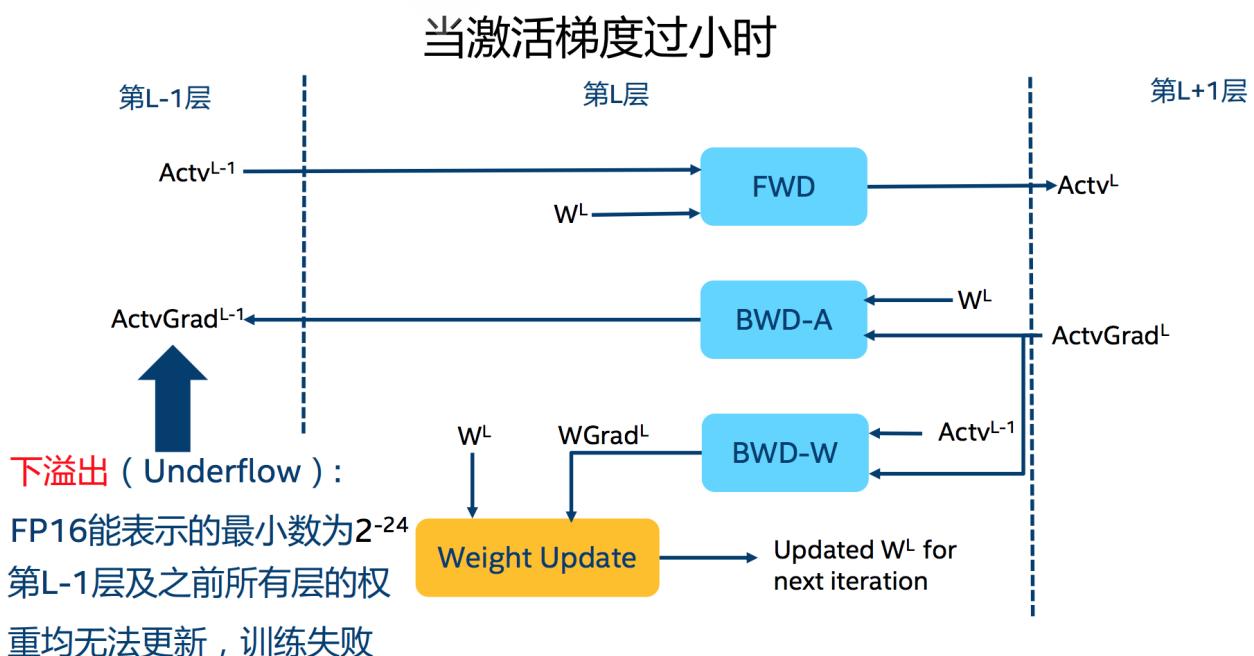
## FP16的优势

1. 减少显存占用；
2. 加速训练和推断的计算，相当于可以提速一倍；
3. 更好的利用CUDA设备上的Tensor Core；

## FP16潜在的问题

其实FP16代替FP32造成最大的问题就是精度损失，其中又分为溢出错误和舍入误差。

1. 溢出错误。因为FP16的范围比FP32要窄，表达范围比它要小，所以在计算过程中很容易出现溢出错误。其中又分为上溢出和下溢出，溢出之后就会在计算中出现"NaN"。在训练时，由于激活函数的梯度比较小，往往小于权重的梯度，所以更加容易落入表达范围外的情况，也就是出现下溢出。



图片来源于<https://blog.csdn.net/junbabab/article/details/119078807>

2. 舍入误差。舍入误差指的是当梯度过小时，即小于当前区间的最小间隔时，这个数值就会被舍弃，那么会导致这次梯度的更新失败。

OK      FP16 weight =  $2^{-3}$  (0.125)

OK      FP16 gradient =  $2^{-14}$  (约等于0.000061)

FP16 weight = weight + gradient

Error       $= 2^{-3} + 2^{-14}$

舍入错误 ( Rounding Error ) :

[ $2^{-3}$ ,  $2^{-2}$ ]间，FP16表示的固定间隔为 $2^{-13}$

即比 $2^{-3}$ 大的下一个数为 $2^{-3} + 2^{-13}$

图片来源于<https://blog.csdn.net/junbabab/article/details/119078807>

如果浮点数越小，那么引入的舍入误差就会越大。那么对足够小的浮点数执行任何操作都会将该值进行四舍五入，从而到0。因为在反向传播时，大多数的梯度更新值都是十分微小的，这些舍入误差累积可能会把这一些值都置零或者变为NaN，从而导致不准确的梯度更新或者梯度更新失败。此外，对于FP16来说，有些参数过小的话的话，也会被置为零。

## 以上问题的解决办法

### 1. 混合精度训练

在训练的FP16矩阵乘法运算中，需要使用FP32来进行矩阵乘法中间的累加计算，然后写入内存时再将FP32的值转化为FP16格式进行存储。即在内存中使用FP16做储存和乘法运算来实现加速，而用FP32进行累加来避免舍入误差。这里主要是认为加法运算主要受限于内存带宽，所以使用混合精度训练既可以避免拖慢速度又可以有效缓解了舍入误差的问题。

### 2. loss放大

即便使用混合精度进行训练，但是可能还是存在无法收敛的情况，因为在训练的后期，激活梯度的值太小，仍会造成下溢出，也就是接近零的数会下溢成零。所以可以通过放大loss来防止梯度的下溢出。注意这里只在梯度回传的时候放大，当实际需要更新权重时，还是会把放大的梯度再缩放回来。

反向传播前，将loss变化手动放大 $2^k$ 倍，因此在反向传播时，所获取的激活函数梯度不会发生溢出；

反向传播后，将权重梯度缩小回 $2^k$ 倍，恢复真实值。

## AMP的工作流程

1. 将输入转为FP16；
2. 进行前向计算，并且在这里使用FP32格式来初始化权重，训练时将FP32的权重转化为FP16来进行前向计算；
3. 计算损失，因为在计算损失过程中，有许多exp, log的计算，可能会造成FP16的溢出，所以这里的loss需要在FP32下进行计算；
4. loss scale：将计算出来的loss乘以一个系数  $\text{scaled\_loss} = \text{loss} * \text{loss\_scale}$ ，经过scaled过后的梯度就会平移到FP16的有效范围中；
5. 反向传播；
6. 梯度计算完后，会将放各梯度乘上相应的缩小的倍数，恢复成原来的大小；
7. 梯度约束，在累许多FP16的数值后，容易发生溢出，所以会在FP32格式下进行约束；
8. 参数更新，根据梯度是否发生溢出来选择是否进行参数的更新并动态调整loss\_scale，如果需要更新，就恢复为FP32格式，并在FP32格式的参数上进行更新，否则继续累加，直至需要更新。

## AMP的具体使用

在通过torch.cuda.amp使用AMP，主要是使用autocast、GradScaler这两个类。

### autocast

使用autocast用于创建AMP上下文环境，在进入autocast的上下文之后，会自动将张量的数据类型转为半精度浮点型，不需要手动调用.half()函数，代码会自动判断哪些层可以转换FP16，而哪些层不可以。

不过，autocast的上下文只包含网络的前向推理过程以及loss的计算，并不包含反向传播，因为反向传播的op会使用和前向推理op一样的类型。

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

model=Net().cuda()
optimizer=optim.SGD(model.parameters(),...)

for input,target in data:
    optimizer.zero_grad()
```

```

with autocast():
    output=model(input)
    loss = loss_fn(output,target)

    loss.backward()
    optimizer.step()

```

## FP32权重备份

这个主要是为了解决舍入误差。在训练过程中会有weights, activations, gradients这几种数据，这些数据在训练时会以FP16格式进行保存，同时复制一份FP32的weights，用于后面的更新。也就是说，梯度更新使用FP16计算，但是是更新在FP32上，这使得应用梯度更新更加安全。

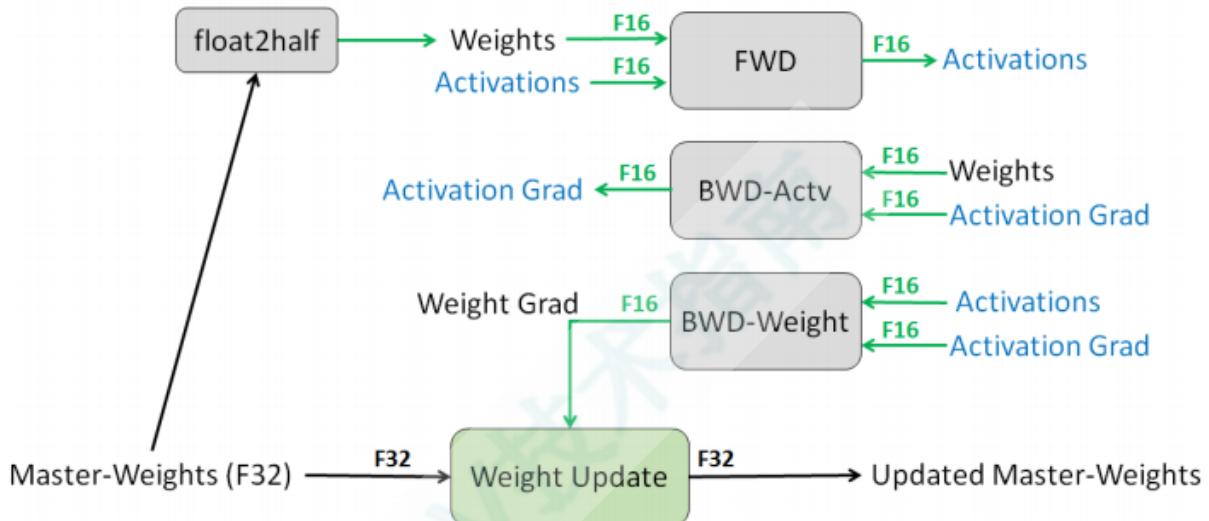
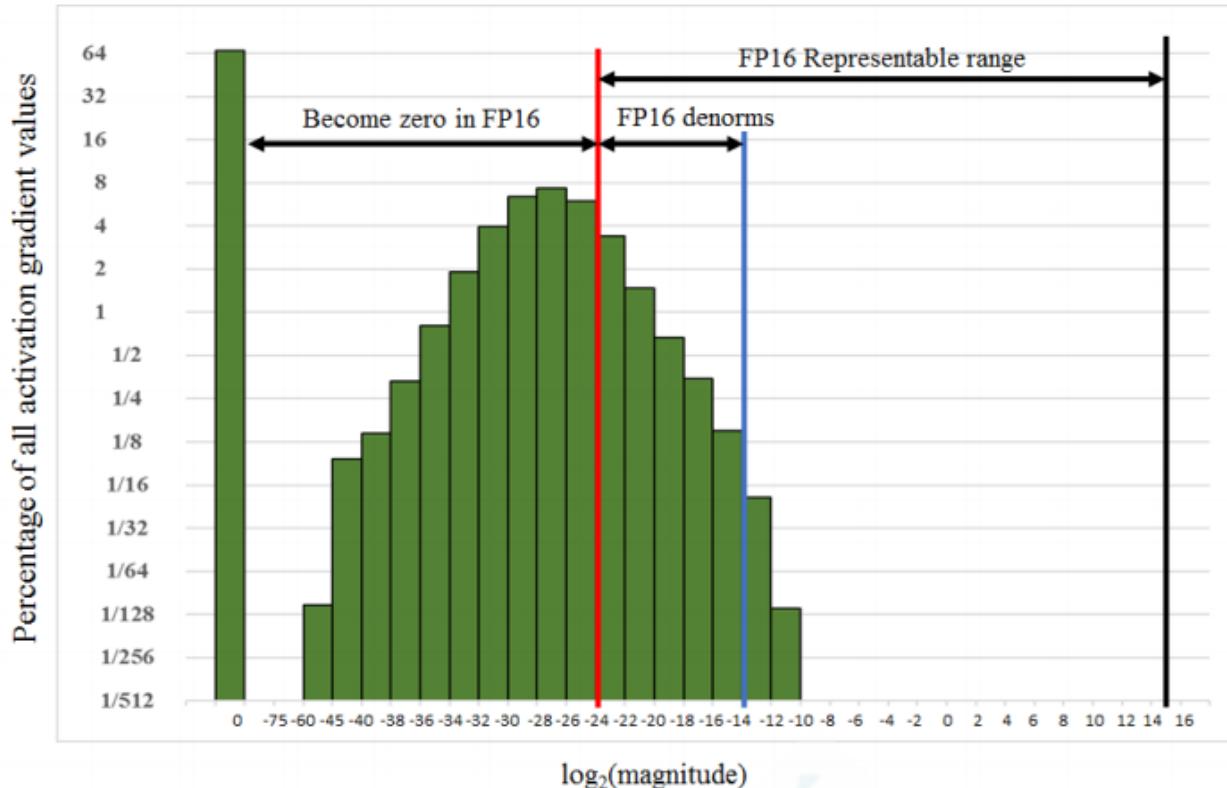


Figure 1: Mixed precision training iteration for a layer.

图片来源于<https://www.cnblogs.com/shona/p/12674011.html>

这主要是因为，在更新权重时，权重 = 旧权重 + lr \* 梯度，而lr \* 梯度这个值一般来说是非常小的，如果再使用FP16进行相加的话，就很可能会出现舍入误差，导致更新无效。

虽然额外拷贝的FP32格式的weight会增加训练时候显存的占用。但是在实际使用中，占绝大部分的还是activations的值，而不是weights。所以，只要使用FP16来保存activation，那么最终FP16的模型大小和FP32的模型相比，显存占用基本能减半。



图片来源于<https://www.cnblogs.com/shona/p/12674011.html>

## GradScaler

在开始训练之前，需要实例化一个GradScaler对象。

scaler的大小会在每次迭代中进行动态估计，为了尽可能减少梯度的下溢出，scaler应该尽可能大；但是如果太大，那么FP16又容易发生上溢出，从而变成inf或NaN。在每次scaler.step(optimizer)中，都会检查是否有inf或NaN的梯度出现，从而实现在不出现inf或NaN的情况下，尽可能的增大scaler。

1. 如果出现inf或NaN，scaler.step(optimizer)会忽略这次权重的更新(optimizer.step())，并且将scaler的大小进行缩小，即乘上backoff\_factor；
2. 如果没有出现inf或NaN，那么会正常更新本次权重，并且当持续growth\_interval次后都没有出现inf或NaN，那么scaler.update()会将scaler的大小乘以growth\_factor。

```

from torch.cuda.amp import autocast as autocast

model=Net().cuda()
optimizer=optim.SGD(model.parameters(),...)

scaler = GradScaler() # 训练前先实例化出一个GradScaler对象

for epoch in epochs:
    for input,target in data:
        optimizer.zero_grad() # 正常更新权重

```

```
with autocast(): # 开启autocast上下文，在autocast上下文范围内，进行模型的前向  
推理和loss计算  
    output=model(input)  
    loss = loss_fn(output,targt)  
  
    scaler.scale(loss).backward() # 对loss进行放大，针对放大后的loss进行反向传播  
    scaler.step(optimizer) # 在这里，首先会把梯度值缩放回来，如果缩放后的梯度不是  
inf或NaN，那么就会调用optimizer.step()来更新权重，否则，忽略step调用，从而保证权重不  
更新。  
    scaler.update() # 看是否要增大scaler，更新scalar的缩放信息
```

训练完一个batch后，不马上使用所获取到的梯度来更新模型，而是继续训练下一个batch的数据，当完成多次循环后累计到一定次数，用这些累加的梯度来更新参数，这样可以起到变相扩大batch\_size的作用。

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

扫码关注公众号CV技术指南，专注于计算机视觉的技术总结、论文解读、招聘信息发布等。



扫码加入CV技术指南知识星球。

CV技术指南旨在打造一个完善的计算机视觉知识体系，为入门的人提供服务，为工作的人提供平台。自星球成立以来，星球内有为星友们提供了许多计算机视觉方面的资料，有理论知识，有部署经验，也有论文解读。其中包括目标检测中Head、Neck的设计系列、YOLO系列、anchor-free系列、小目标检测系列、目标检测中的Label Assignment系列、传统目标检测系列、视频中的目标检测系列、图像分割系列、部署系列、CUDA教程系列、论文解读系列、Pytorch源码解读系列、Pytorch实践教程系列。后续还会继续补充~...



欢迎加入QQ群：444129970。QQ群专注于计算机视觉的算法、技术、学习、工作、求职等方面的交流，群内交流氛围极好，有专门的大佬维护，基本99%的提问都会解答。群文件内有很多电子版资源，可供大家免费下载，也欢迎其他人一起上传新资料。



对于不习惯用QQ的朋友，请扫描下方二维码添加编辑微信，可邀请加入微信群：添加前请备注“研究方向-城市-id”。



## (七) 特征图可视化

**前言** 本文给大家分享一份我用的特征图可视化代码。

### 写在前面的话

**特征图可视化是很多论文所需要做的一份工作，其作用可以是用于证明方法的有效性，也可以是用来增加工作量，给论文凑字数。**

具体来说就是可视化两个图，使用了新方法的和使用之前的，对比有什么区别，然后看图写论文说明新方法体现的作用。

吐槽一句，有时候这个图 论文作者自己都不一定能看不懂，虽然确实可视化的图有些改变，但并不懂这个改变说明了什么，反正就吹牛，强行往自己新方法编的故事上扯，就像小学一年级的作文题--看图写作文。

之前知乎上有一个很热门的话题，如果我在baseline上做了一点小小的改进，却有很大的效果，这能写论文吗？

这种情况最大的问题就在于要如何写七页以上，那一点点的改进可能写完思路，公式推理，画图等内容才花了不到一页，剩下的内容如何搞？可视化特征图！！！

这一点可以在我看过的甚多论文上有所体现，反正我是没看明白论文给的可视化图，作者却能扯那么多道道。这应该就是用来增加论文字数和增加工作量的。

总之一句话，**可视化特征图是很重要的工作，最好要会**。

## 初始化配置

这部分先完成加载数据，修改网络，定义网络，加载预训练模型。

### 加载数据并预处理

这里只加载一张图片，就不用通过classdataset了，因为classdataset是针对大量数据的，生成一个迭代器一批一批地将图片送给网络。但我们仍然要完成classdataset中数据预处理的部分。

数据预处理所必须要有的操作是调整大小，转化为Tensor格式，归一化。至于其它数据增强或预处理的操作，自己按需添加。

```
def image_proprecess(img_path):
    img = Image.open(img_path)
    data_transforms = transforms.Compose([
        transforms.Resize((384, 384), interpolation=3),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
    data = data_transforms(img)
    data = torch.unsqueeze(data, 0)
    return data
```

这里由于只加载一张图片，因此后面要使用torch.unsqueeze将三维张量变成四维。

### 修改网络

假如你要可视化某一层的特征图，则需要将该层的特征图返回出来，因此需要先修改网络中的forward函数。具体修改方式如下所示。

```
def forward(self, x):
    x = self.model.conv1(x)
    x = self.model.bn1(x)
    x = self.model.relu(x)
    x = self.model.maxpool(x)
    feature = self.model.layer1(x)
    x = self.model.layer2(feature)
    x = self.model.layer3(x)
    x = self.model.layer4(x)
    return feature,x
```

## 定义网络并加载预训练模型

```
def Init_Setting(epoch):
    dirname = '/mnt/share/VideoReID/share/models/Methods5_trial1'
    model = siamese_resnet50(701, stride=1, pool='avg')
    trained_path = os.path.join(dirname, 'net_%03d.pth' % epoch)
    print("load %03d.pth" % epoch)
    model.load_state_dict(torch.load(trained_path))
    model = model.cuda().eval()
    return model
```

这部分需要说明的是最后一行，要将网络设置为推理模式。

## 可视化特征图

这部分主要是将特征图的某一通道转化为一张图来可视化。

```
def visualize_feature_map(img_batch,out_path,type,BI):
    feature_map = torch.squeeze(img_batch)
    feature_map = feature_map.detach().cpu().numpy()

    feature_map_sum = feature_map[0, :, :]
    feature_map_sum = np.expand_dims(feature_map_sum, axis=2)
    for i in range(0, 2048):
        feature_map_split = feature_map[i, :, :]
        feature_map_split = np.expand_dims(feature_map_split, axis=2)
        if i > 0:
```

```

        feature_map_sum += feature_map_split
feature_map_split = BI.transform(feature_map_split)

plt.imshow(feature_map_split)
plt.savefig(out_path + str(i) + "_{}.jpg".format(type) )
plt.xticks()
plt.yticks()
plt.axis('off')

feature_map_sum = BI.transform(feature_map_sum)
plt.imshow(feature_map_sum)
plt.savefig(out_path + "sum_{}.jpg".format(type))
print("save sum_{}.jpg".format(type))

```

这里一行一行来解释。

- \1. 参数img\_batch是从网络中的某一层传回来的特征图，BI是双线性插值的函数，自定义的，下面会讲。
- \2. 由于只可视化了一张图片，因此img\_batch是四维的，且batchsize维为1。第三行将它从GPU上弄到CPU上，并变成numpy格式。
- \3. 剩下部分主要完成将每个通道变成一张图，以及将所有通道每个元素对应位置相加，并保存。

## 双线性插值

由于经过多次网络降采样，后面层的特征图往往变得只有7x7,16x16大小。可视化后特别小，因此需要将它上采样，这里采样的方式是双线性插值。因此，这里给一份双线性插值的代码。

```

class BilinearInterpolation(object):
    def __init__(self, w_rate: float, h_rate: float, *, align='center'):
        if align not in ['center', 'left']:
            logging.exception(f'{align} is not a valid align parameter')
        align = 'center'
        self.align = align
        self.w_rate = w_rate
        self.h_rate = h_rate

    def set_rate(self, w_rate: float, h_rate: float):
        self.w_rate = w_rate      # w 的缩放率
        self.h_rate = h_rate      # h 的缩放率

    # 由变换后的像素坐标得到原图像的坐标      针对高

```

```

def get_src_h(self, dst_i, source_h, goal_h) -> float:
    if self.align == 'left':
        # 左上角对齐
        src_i = float(dst_i * (source_h/goal_h))
    elif self.align == 'center':
        # 将两个图像的几何中心重合。
        src_i = float((dst_i + 0.5) * (source_h/goal_h) - 0.5)
    src_i += 0.001
    src_i = max(0.0, src_i)
    src_i = min(float(source_h - 1), src_i)
    return src_i

# 由变换后的像素坐标得到原图像的坐标    针对宽
def get_src_w(self, dst_j, source_w, goal_w) -> float:
    if self.align == 'left':
        # 左上角对齐
        src_j = float(dst_j * (source_w/goal_w))
    elif self.align == 'center':
        # 将两个图像的几何中心重合。
        src_j = float((dst_j + 0.5) * (source_w/goal_w) - 0.5)
    src_j += 0.001
    src_j = max(0.0, src_j)
    src_j = min((source_w - 1), src_j)
    return src_j

def transform(self, img):
    source_h, source_w, source_c = img.shape # (235, 234, 3)
    goal_h, goal_w = round(
        source_h * self.h_rate), round(source_w * self.w_rate)
    new_img = np.zeros((goal_h, goal_w, source_c), dtype=np.uint8)

    for i in range(new_img.shape[0]):          # h
        src_i = self.get_src_h(i, source_h, goal_h)
        for j in range(new_img.shape[1]):
            src_j = self.get_src_w(j, source_w, goal_w)
            i2 = ceil(src_i)
            i1 = int(src_i)
            j2 = ceil(src_j)
            j1 = int(src_j)
            x2_x = j2 - src_j
            x_x1 = src_j - j1
            y2_y = i2 - src_i
            y_y1 = src_i - i1
            new_img[i, j] = img[i1, j1]*x2_x*y2_y + img[i1, j2] *
                x_x1*y2_y + img[i2, j1]*x2_x*y_y1 + img[i2, j2]*x_x1*y_y1
    return new_img

#使用方法
BI = BilinearInterpolation(8, 8)

```

```
feature_map = BI.transform(feature_map)
```

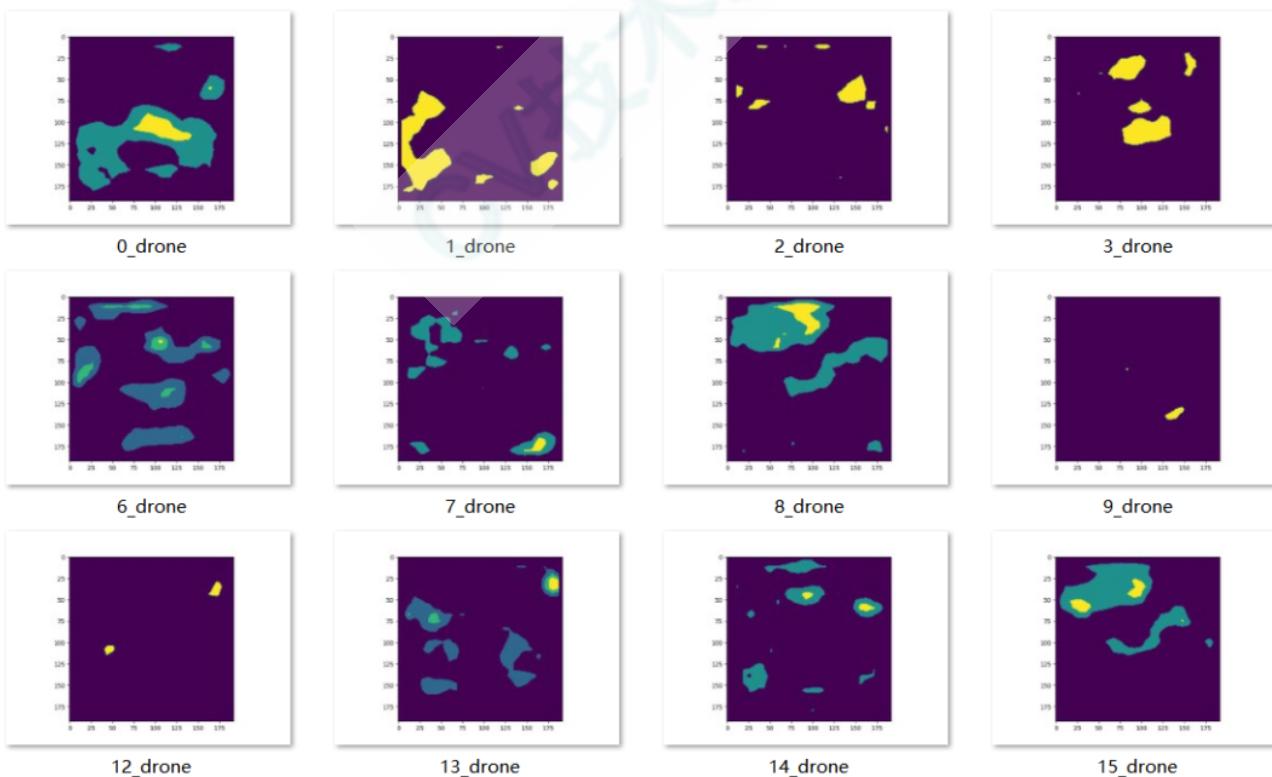
## main函数流程

上面介绍了各个部分的代码，下面就是整体流程。比较简单。

```
imgs_path = "/path/to/imgs/"
save_path = "/save/path/to/output/"
model = Init_Setting(120)
BI = BilinearInterpolation(8, 8)

data = image_proprecess(out_path + "0836.jpg")
data = data.cuda()
output, _ = model(data)
visualize_feature_map(output, save_path, "drone", BI)
```

## 可视化效果图



本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## (八) 热力图可视化

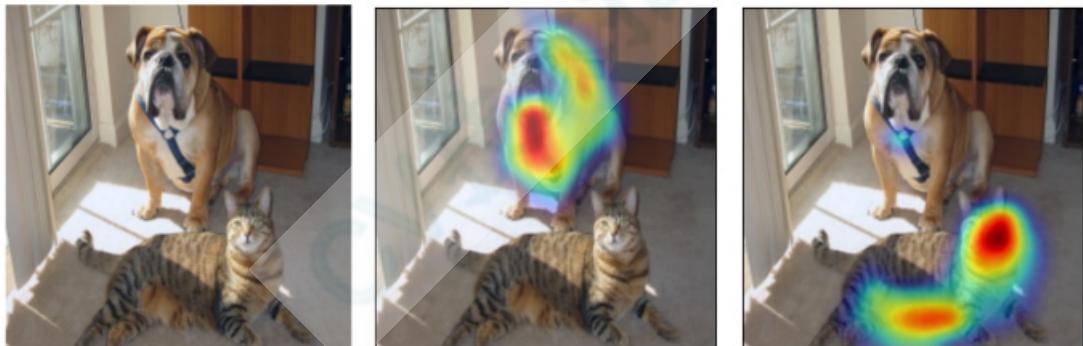
本教程共包含热力图可视化和特征图可视化两个方法的使用。

特征图可视化与热力图可视化是论文中比较常用的两种可视化方法。上一篇文章《一份可视化特征图的代码》介绍了特征图可视化的代码，本篇将对如何进行热力图可视化做一个使用说明。

本文介绍了CAM、GradCAM的原理和缺陷，介绍了如何使用GradCAM算法实现热力图可视化，介绍了目标检测、语义分割、transformer模型等其它类型任务的热力图可视化。

### 热力图可视化方法的原理

在一个神经网络模型中，图片经过神经网络得到类别输出，我们并不知道模型是根据什么来作出预测的，换言之，我们需要了解图片中各个区域对模型作出预测的影响有多大。这就是热力图的作用，它通过得到图像不同区域之间对模型的重要性而生成一张类似于等温图的图片。

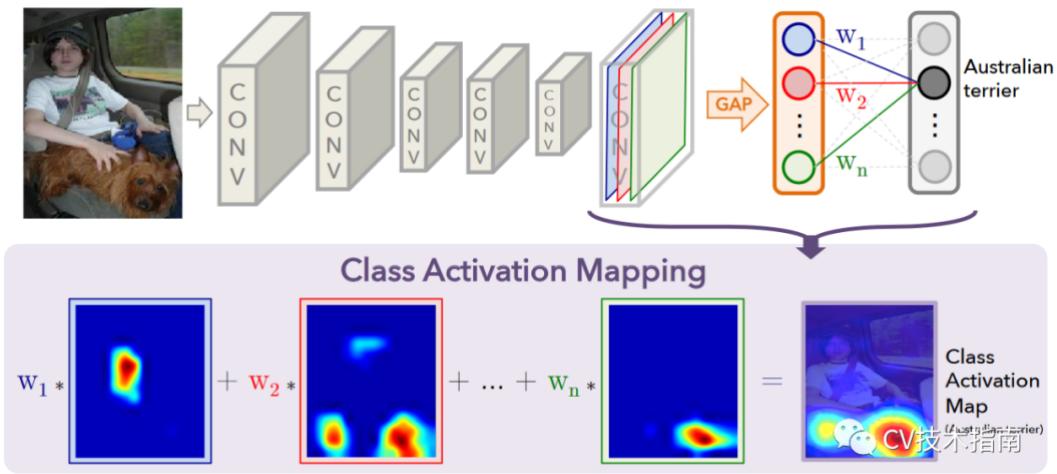


热力图可视化方法经过了从CAM，GradCAM，到GradCAM++的过程，比较常用的是GradCAM算法。

#### CAM

CAM论文：Learning Deep Features for Discriminative Localization

CAM的原理是取出全连接层中得到类别C的概率的那一维权值，用W表示。然后对GAP前的feature map进行加权求和，由于此时feature map不是原图像大小，在加权求和后还需要进行上采样，即可得到Class Activation Map。



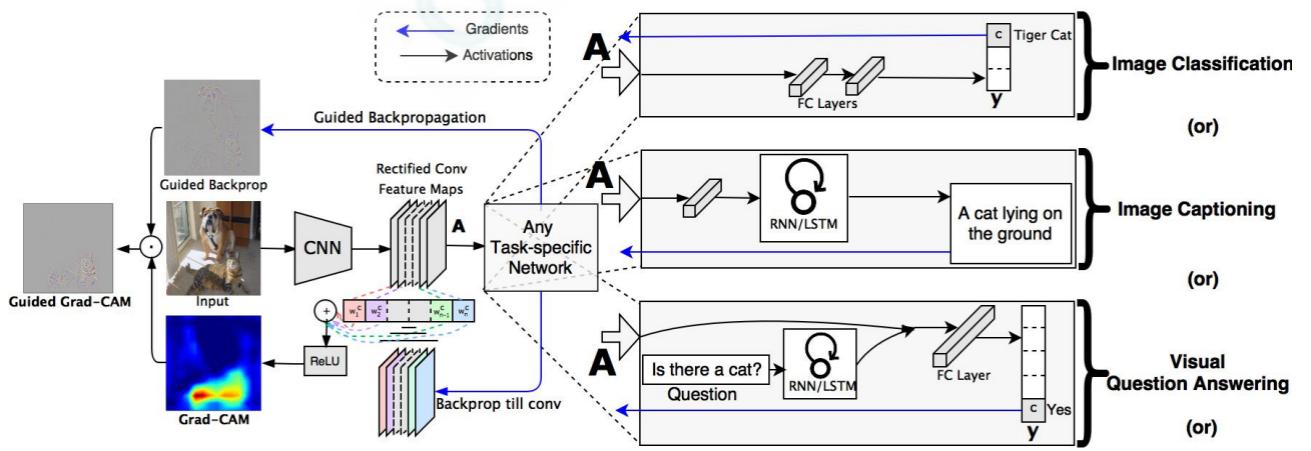
CAM有个很致命的缺陷，它的结构是由CNN + GAP + FC + Softmax组成。也就是说如果想要可视化某个现有的模型，对于没有GAP的模型来说需要修改原模型结构，并重新训练，相当麻烦，且如果模型很大，在修改后重新训练不一定能达到原效果，可视化也就没有意义了。

因此，针对这个缺陷，其后续有了改进版Grad-CAM。

## GradCAM

Grad-CAM 论文：Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization

Grad-CAM的最大特点就是不再需要修改现有的模型结构了，也不需要重新训练了，直接在原模型上即可可视化。



原理：同样是处理CNN特征提取网络的最后一层feature maps。Grad-CAM对于想要可视化的类别C，使最后输出的类别C的概率值通过反向传播到最后一层feature maps，得到类别C对该feature maps的每个像素的梯度值，对每个像素的梯度值取全局平均池化，即可得到对feature maps的加权系数alpha，论文中提到这样获取的加权系数跟CAM中的系数的计算量几乎是等价的。接下来对特征图加权求和，使用ReLU进行修正，再进行上采样。

使用ReLU的原因是对于那些负值，可认为与识别类别C无关，这些负值可能是与其他类别有关，而正值才是对识别C有正面影响的。

具体公式如下：

$$\alpha_k^c = \underbrace{\frac{1}{Z} \sum_i \sum_j}_{\text{global average pooling}} \underbrace{\frac{\partial y^c}{\partial A_{ij}^k}}_{\text{gradients via backprop}}$$
$$L_{\text{Grad-CAM}}^c = \text{ReLU} \left( \underbrace{\sum_k \alpha_k^c A^k}_{\text{linear combination}} \right)$$

Grad-CAM后续还有改进版Grad-CAM++，其主要的改进效果是定位更准确，更适合同类多目标的情况，所谓同类多目标是指一张图像中对于某个类出现多个目标，例如七八个人。改进方法是对加权系数的获取提出新的方法，该方法很复杂，这里不介绍。

## GradCAM的使用教程

这份代码来自GradCAM论文作者，原链接中包含了很多其它的CAM，这里将GradCAM摘出来对其做一个使用说明。

代码原链接：[https://github.com/jacobgil/pytorch-grad-cam/tree/master/pytorch\\_grad\\_cam](https://github.com/jacobgil/pytorch-grad-cam/tree/master/pytorch_grad_cam)

本教程代码链接：<https://github.com/CV-Tech-Guide/Visualize-feature-maps-and-heatmap>

## 使用流程

使用起来比较简单，仅了解主函数即可。

```
if __name__ == "__main__":
    imgs_path = "path/to/image.png"
    model = models.mobilenet_v3_large(pretrained=True)
    model.load_state_dict(torch.load('model.pth'))
    model = model.cuda().eval()

#target_layers指的是需要可视化的层，这里可视化最后一层
target_layers = [model.features[-1]]
img, data = image_preprocess(imgs_path)
data = data.cuda()
```

```
cam = GradCAM(model=model, target_layers=target_layers)
#指定可视化的类别，指定为None，则按照当前预测的最大概率的类作为可视化类。
target_category = None

grayscale_cam = cam(input_tensor=data, target_category=target_category)
grayscale_cam = grayscale_cam[0, :]
visualization = show_cam_on_image(np.array(img) / 255., grayscale_cam)
plt.imshow(visualization)
plt.xticks()
plt.yticks()
plt.axis('off')
plt.savefig("path/to/gradcam_image.jpg")
```

如上代码所示，仅需要自主设置输入图片，模型，可视化层，可视化类别即可，其它的部分可完全照用。

下面细节部分的介绍。

## 数据预处理

这里跟上次可视化特征图的代码一样，将图片读取，resize，转化为Tensor，格式化，若只有一张图片，则还需要将其扩展为四维。

```
def image_proprecess(img_path):
    img = Image.open(img_path)
    data_transforms = transforms.Compose([
        transforms.Resize((384, 384), interpolation=3),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
    data = data_transforms(img)
    data = torch.unsqueeze(data, 0)
    img_resize = img.resize((384, 384))
    return img_resize, data
```

## GradCAM

GradCAM这个类是按照前面第一节中介绍的原理封装的，因此了解原理后再了解这个类的代码就比较简单了。

```
class GradCAM:  
    def __init__(self, model, target_layers, reshape_transform=None):  
        self.model = model.eval()  
        self.target_layers = target_layers  
        self.reshape_transform = reshape_transform  
        self.cuda = use_cuda  
        self.activations_and_grads = ActivationsAndGradients(  
            self.model, target_layers, reshape_transform)  
  
    """ Get a vector of weights for every channel in the target layer.  
    Methods that return weights channels,  
    will typically need to only implement this function. """  
  
    @staticmethod  
    def get_cam_weights(grads):  
        return np.mean(grads, axis=(2, 3), keepdims=True)  
  
    @staticmethod  
    def get_loss(output, target_category):  
        loss = 0  
        for i in range(len(target_category)):  
            loss = loss + output[i, target_category[i]]  
        return loss  
  
    def get_cam_image(self, activations, grads):  
        weights = self.get_cam_weights(grads)  
        weighted_activations = weights * activations  
        cam = weighted_activations.sum(axis=1)  
  
        return cam  
  
    @staticmethod  
    def get_target_width_height(input_tensor):  
        width, height = input_tensor.size(-1), input_tensor.size(-2)  
        return width, height  
  
    def compute_cam_per_layer(self, input_tensor):  
        activations_list = [a.cpu().data.numpy()  
                            for a in self.activations_and_grads.activations]  
        grads_list = [g.cpu().data.numpy()  
                     for g in self.activations_and_grads.gradients]  
        target_size = self.get_target_width_height(input_tensor)
```

```

cam_per_target_layer = []
# Loop over the saliency image from every layer

    for layer_activations, layer_grads in zip(activations_list,
grads_list):
        cam = self.get_cam_image(layer_activations, layer_grads)
        cam[cam < 0] = 0 # works like mute the min-max scale in the
function of scale_cam_image
        scaled = self.scale_cam_image(cam, target_size)
        cam_per_target_layer.append(scaled[:, None, :])

return cam_per_target_layer

def aggregate_multi_layers(self, cam_per_target_layer):
    cam_per_target_layer = np.concatenate(cam_per_target_layer, axis=1)
    cam_per_target_layer = np.maximum(cam_per_target_layer, 0)
    result = np.mean(cam_per_target_layer, axis=1)
    return self.scale_cam_image(result)

@staticmethod
def scale_cam_image(cam, target_size=None):
    result = []
    for img in cam:
        img = img - np.min(img)
        img = img / (1e-7 + np.max(img))
        if target_size is not None:
            img = cv2.resize(img, target_size)
        result.append(img)
    result = np.float32(result)

return result

def __call__(self, input_tensor, target_category=None):
    # 正向传播得到网络输出logits(未经过softmax)
    output = self.activations_and_grads(input_tensor)
    if isinstance(target_category, int):
        target_category = [target_category] * input_tensor.size(0)

    if target_category is None:
        target_category = np.argmax(output.cpu().data.numpy(), axis=-1)
        print(f"category id: {target_category}")
    else:
        assert (len(target_category) == input_tensor.size(0))

    self.model.zero_grad()
    loss = self.get_loss(output, target_category)

```

```

loss.backward(retain_graph=True)

# In most of the saliency attribution papers, the saliency is
# computed with a single target layer.
# Commonly it is the last convolutional layer.
# Here we support passing a list with multiple target layers.
# It will compute the saliency image for every image,
# and then aggregate them (with a default mean aggregation).
# This gives you more flexibility in case you just want to
# use all conv layers for example, all Batchnorm layers,
# or something else.
cam_per_layer = self.compute_cam_per_layer(input_tensor)
return self.aggregate_multi_layers(cam_per_layer)

def __del__(self):
    self.activations_and_grads.release()

def __enter__(self):
    return self

def __exit__(self, exc_type, exc_value, exc_tb):
    self.activations_and_grads.release()
    if isinstance(exc_value, IndexError):
        # Handle IndexError here...
        print(
            f"An exception occurred in CAM with block: {exc_type}.
Message: {exc_value}")
    return True

```

简要说明一下整体在做什么，先通过下方的ActivationsAndGradients获取模型推理过程中的梯度和激活函数值，计算要可视化的类的loss（其它类的都忽略），通过这个loss计算可视化类对应的梯度图，将其进行全局平均池化获得每个feature maps通道的加权系数，与feature maps进行通道上加权，并在通道上做均值获得单通道图，再ReLU即输出对应的图。注：此图还不是热力图，还需要与原图相加才能获得最终的热力图。

GradCAM这个类主要就是先定义，再调用执行。定义须输入网络和需要可视化的层，执行则需要输入图片和可视化的类别。

执行返回的是区域重要性图。

```

cam = GradCAM(model=model, target_layers=target_layers)
#指定可视化的类别，指定为None，则按照当前预测的最大概率的类作为可视化类。
target_category = None

grayscale_cam = cam(input_tensor=data, target_category=target_category)

```

获取推理过程中的梯度主要是通过以下这个类来完成。这里不多介绍。

```
class ActivationsAndGradients:
    """ Class for extracting activations and
    registering gradients from targeted intermediate layers """
    def __init__(self, model, target_layers, reshape_transform):
        self.model = model
        self.gradients = []
        self.activations = []
        self.reshape_transform = reshape_transform
        self.handles = []
        for target_layer in target_layers:
            self.handles.append(
                target_layer.register_forward_hook(
                    self.save_activation))
        # Backward compatibility with older pytorch versions:
        if hasattr(target_layer, 'register_full_backward_hook'):
            self.handles.append(
                target_layer.register_full_backward_hook(
                    self.save_gradient))
        else:
            self.handles.append(
                target_layer.register_backward_hook(
                    self.save_gradient))

    def save_activation(self, module, input, output):
        activation = output
        if self.reshape_transform is not None:
            activation = self.reshape_transform(activation)
        self.activations.append(activation.cpu().detach())

    def save_gradient(self, module, grad_input, grad_output):
        # Gradients are computed in reverse order
        grad = grad_output[0]
        if self.reshape_transform is not None:
            grad = self.reshape_transform(grad)
        self.gradients = [grad.cpu().detach()] + self.gradients

    def __call__(self, x):
        self.gradients = []
        self.activations = []
        return self.model(x)

    def release(self):
        for handle in self.handles:
            handle.remove()
```

然后就是将GradCAM输出的重要性图在原图上显示，通过下面这个函数完成。

```
def show_cam_on_image(img: np.ndarray,
                      mask: np.ndarray,
                      use_rgb: bool = False,
                      colormap: int = cv2.COLORMAP_JET) -> np.ndarray:
    """ This function overlays the cam mask on the image as an heatmap.
    By default the heatmap is in BGR format.

    :param img: The base image in RGB or BGR format.
    :param mask: The cam mask.
    :param use_rgb: Whether to use an RGB or BGR heatmap, this should be set
    to True if 'img' is in RGB format.
    :param colormap: The OpenCV colormap to be used.
    :returns: The default image with the cam overlay.
    """

    heatmap = cv2.applyColorMap(np.uint8(255 * mask), colormap)
    if use_rgb:
        heatmap = cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB)
    heatmap = np.float32(heatmap) / 255

    if np.max(img) > 1:
        raise Exception(
            "The input image should np.float32 in the range [0, 1]")

    cam = heatmap + img
    cam = cam / np.max(cam)
    return np.uint8(255 * cam)
```

前面介绍的仅仅是分类任务的热力图可视化，但对于目标检测、语义分割等这些包含多任务的应用如何做？

## 其它类型任务的热力图可视化

在gradCAM论文作者给出的代码中还介绍了如何可视化目标检测、语义分割、transformer的代码。由于作者提供了使用方法，这里不多介绍，直接给出作者写得教程。

- [Notebook tutorial: Class Activation Maps for Object Detection with Faster-RCNN](#)
- [Notebook tutorial: Class Activation Maps for Semantic Segmentation](#)
- [How it works with Vision/SwinT transformers](#)

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## (九) Pytorch实践部分

### 冻结、微调网络

## 前言

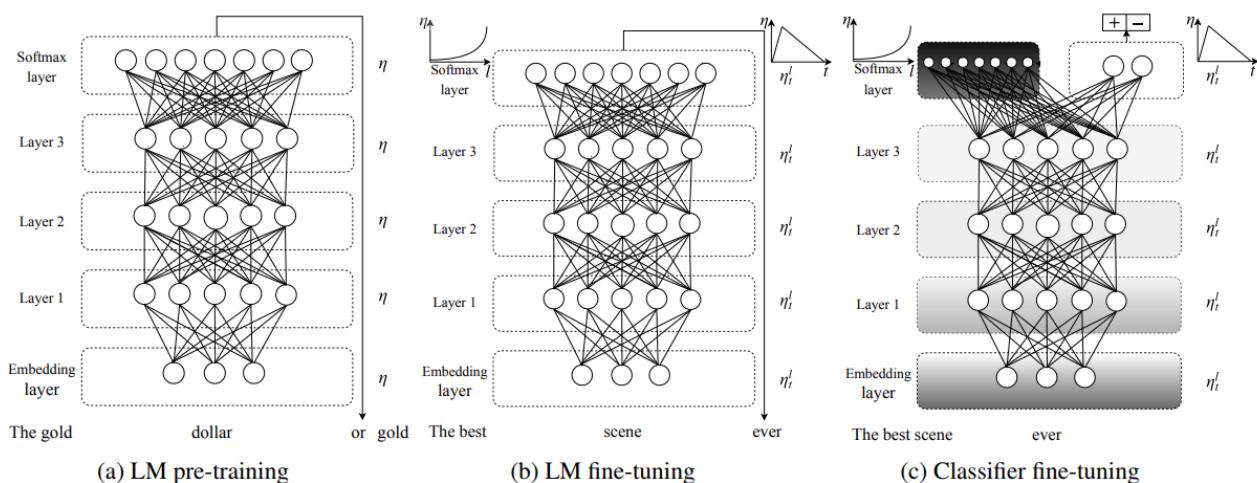
在讲关于模型的冻结、微调前，需要引入一个概念，叫迁移学习。迁移学习是指利用旧知识来学习新知识，主要目标是将已经学会的知识很快地迁移到一个新的领域中。

## 迁移学习

当我们训练好了一个模型之后，如果想应用到其他任务中，可以在这个模型的基础上进行训练，来作微调网络。这也是迁移学习的概念，可以节省训练的资源以及训练的时间。

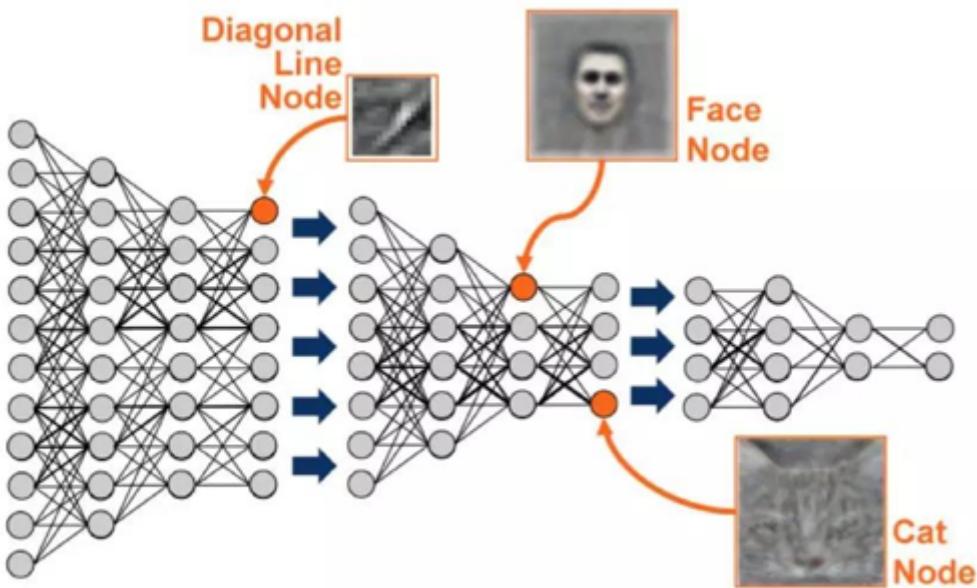
## 模型微调

微调，就是基于一个预训练模型进行训练，比如使用大型数据集ImageNet或者COCO训练好的模型。当然，自己训练好的模型也可以当做预训练模型，然后再在自己的数据集上进行训练，来使模型适用于自己的场景、自己的任务。



为什么要用预训练模型作微调呢？

因为预训练模型用了大量数据做训练，已经具备了提取浅层基础特征和深层抽象特征的能力。



如果数据不够多，泛化性不够强，那么可能存在模型不收敛，准确率低，模型泛化能力差，过拟合等问题，所以这时就需要使用预训练模型来做微调了。

注意的是，进行微调时，应该使用**较小的学习率**。因为预训练模型的权重相对于随机初始化的权重来说已经很不错了，所以不希望使用太大的学习率来破坏原本的权重。

通常用于微调的初始学习率会比从头开始训练的学习率小10倍。

## 需要微调的情况

其中微调的方法又要根据自身数据集和预训练模型数据集的相似程度，以及自己数据集的大小来抉择。

不同情况下的微调：

**数据少，数据类似程度高：**可以只修改最后几层或者最后一层进行微调。

**数据少，数据类似程度低：**冻结预训练模型的前几层，训练剩余的层。因为数据集之间的相似度较低，所以根据自身的数据集对较高层进行重新训练会比较有效。

**数据多，数据类似程度高：**这是最理想的情况。使用预训练的权重来初始化模型，然后重新训练整个模型。这也是最简单的微调方式，因为不涉及修改、冻结模型的层。

**数据多，数据类似程度低：**微调的效果估计不好，可以考虑直接重新训练整个模型。如果你用的预训练模型的数据集是ImageNet，而你要做的是文字识别，那么预训练模型自然不会起到太大作用，因为它们的场景特征相差太大了。

## 微调的步骤

微调的步骤有很多，看你自身数据和计算资源的情况而定。虽然各有不同，但是总体的流程大同小异。

## 步骤示例一

1. 在已经训练好的网络上进行修改；
2. 冻结网络的原来那一部分；
3. 训练新添加的部分；
4. 解冻原来网络的部分层；
5. 联合训练解冻的层和新添加的部分。

## 步骤示例二

1. 在源数据集如ImageNet数据集上预训练一个源模型；
2. 创建一个新的模型，即目标模型。它沿用了源模型上除了输出层外的所有模型结果及其参数；
3. 为目标模型添加一个合适的输出大小，并随机初始化该层的模型参数，或者其它初始化方法；
4. 在目标数据集上训练目标模型。可以从头训练输出层，而剩下的层都是基于源模型的参数进行微调。

## 参数冻结

我们所提到的冻结模型、冻结部分层，其实归根结底都是对参数进行冻结。冻结训练可以加快训练速度。

在这里，有两种方式：全程冻结与非全程冻结。

非全程冻结比全程冻结多了一个步骤：解冻，因此这里就讲解非全程冻结。看完非全程冻结之后，就明白全程冻结是如何进行的了。

非全程冻结训练分为两个阶段，分别是冻结阶段和解冻阶段。当处于冻结阶段时，被冻结的参数就不会被更新，在这个阶段，可以看做是全程冻结；而处于解冻阶段时，就和普通的训练一样了，所有参数都会被更新。

当进行冻结训练时，占用的显存较小，因为仅对部分网络进行微调。如果计算资源不够，也可以通过冻结训练的方式来减少训练时资源的占用。

## 冻结的方式

我们经常提到的模型，就是一个可遍历的字典。既然是字典，又是可遍历的，那么就有两种方式进行索引：一是通过数字，二是通过名字。

其实使用冻结很简单，没有太高深的魔法，只用设置模型的参数`requires_grad`为`False`就可以了。

## 方式一

通过数字来遍历模型中的层的参数，冻结所指定的若干个参数。

```
count = 0
for layer in model.children():
    count = count + 1
    if count < 10:
        for param in layer.parameters():
            param.requires_grad = False
```

然后将需要训练的参数传入优化器，也就是过滤掉被冻结的参数。

```
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad,
model.parameters()), lr=LR)
```

## 方式二

因为ImageNet有1000个类别，所以提供的ImageNet预训练模型也是1000分类。如果我要训练一个10分类模型，理论上来说只需要修改最后一层的全连接层即可。

如果前面的参数不冻结就表示所有特征提取的层会使用预训练模型的参数来进行参数初始化，而最后一层的参数还是保持某种初始化的方式来进行初始化。

在模型中，每一层的参数前面都有前缀，比如conv1、conv2、fc3、backbone等等，我们可以通过这个前缀来进行判断，也就是通过名字来判断，如：if "backbone" in param.name，最终选择需要冻结与不需要冻结的层。

```
if freeze_layers:
    for name, param in model.named_parameters():
        # 除最后的全连接层外，其他权重全部冻结
        if "fc" not in name:
            param.requires_grad_(False)

    pg = [p for p in model.parameters() if p.requires_grad]

optimizer = optim.SGD(pg, lr=0.01, momentum=0.9, weight_decay=4E-5)
```

或者判断该参数位于模型的哪些模块层中，如param in model.backbone.parameters()，然后对于该模块层的全部参数进行批量设置，将requires\_grad置为False。

```
if Freeze_Train:
    for param in model.backbone.parameters():
        param.requires_grad = False
```

同样，方式二最后也是需要和上面方式一的末尾一样，将训练的参数传入优化器进行配置。

## 修改模型参数

前面说道，冻结模型就是冻结参数，那么这里的修改模型参数更多的是修改模型参数的名称。

值得一提的是，由于训练方式（单卡、多卡训练）、模型定义的方式不同，参数的名称也会有所区别，但是此时模型的结构是一样的，依旧可以加载预训练模型。不过却无法直接载入预训练模型的参数，因为名称不同，会出现KeyError的错误，所以载入前可能需要修改参数的名称。

比如说，使用多卡训练时，保存的时候每个参数前面多会多出'module.'这几个字符，那么当使用单卡载入时，可能就会报错了。

```
('module.mlp_head.1.weight', tensor([[-0.0465,  0.0419,  0.0044,
```

通过以下方式，就可以使用'conv1'来替代'module.conv1'这个key的方式来将更新后的key和原来的value相匹配，再载入自己定义的模型中。

```
model_dict = pretrained_model.state_dict()

pretrained_dict={k: v for k, v in pretrained_dict.items() if k[7:] in
model_dict}

model_dict.update(pretrained_dict)
```

## 修改模型结构

修改模型结构会稍微复杂一些，但其实无非就是增删改：给模型增加一个分支、删除模型的某一层，修改模型的输出维度等等。下面就来看看是如何进行模型修改的：

```
import torch.nn as nn
import torch

class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.features=nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2), # 使用卷积
            nn.ReLU(inplace=True), # 使用激活函数
            nn.MaxPool2d(kernel_size=3, stride=2), # 使用最大池化，这里的大小为
            3, 步长为2
```

```

        nn.Conv2d(64, 192, kernel_size=5, padding=2), # 使用卷积层，输入为
64, 输出为192, 核大小为5, 步长为2
        nn.ReLU(inplace=True),# 使用激活函数
        nn.MaxPool2d(kernel_size=3, stride=2), # 使用最大池化，这里的大小为
3, 步长为2
        nn.Conv2d(192, 384, kernel_size=3, padding=1), # 使用卷积层，输入为
192, 输出为384, 核大小为3, 步长为1
        nn.ReLU(inplace=True),# 使用激活函数
        nn.Conv2d(384, 256, kernel_size=3, padding=1),# 使用卷积层，输入为
384, 输出为256, 核大小为3, 步长为1
        nn.ReLU(inplace=True),# 使用激活函数
        nn.Conv2d(256, 256, kernel_size=3, padding=1),# 使用卷积层，输入为
256, 输出为256, 核大小为3, 步长为1
        nn.ReLU(inplace=True),# 使用激活函数
        nn.MaxPool2d(kernel_size=3, stride=2), # 使用最大池化，这里的大小为
3, 步长为2
    )
    self.avgpool=nn.AdaptiveAvgPool2d((6, 6))
    self.classifier=nn.Sequential(
        nn.Dropout(),# 使用Dropout来减缓过拟合
        nn.Linear(256 * 6 * 6, 4096), # 全连接，输出为4096
        nn.ReLU(inplace=True),# 使用激活函数
        nn.Dropout(),# 使用Dropout来减缓过拟合
        nn.Linear(4096, 4096), # 维度不变，因为后面引入了激活函数，从而引入非
线性
        nn.ReLU(inplace=True), # 使用激活函数
        nn.Linear(4096, 1000), #ImageNet默认为1000个类别，所以这里进行1000
个类别分类
    )
}

def forward(self, x):
    x=self.features(x)
    x=self.avgpool(x)
    x=torch.flatten(x, 1)
    x=self.classifier(x)
    return x

def alexnet(num_classes, device, pretrained_weights=""):
    net=AlexNet() # 定义AlexNet
    if pretrained_weights: # 判断预训练模型路径是否为空，如果不为空则加载

        net.load_state_dict(torch.load(pretrained_weights, map_location=device))

        num_fc=net.classifier[6].in_features # 获取输入到全连接层的输入维度信息
        net.classifier[6]=torch.nn.Linear(in_features=num_fc,
out_features=num_classes) # 根据数据集的类别数来指定最后输出的out_features数目

```

```
return net
```

在上述代码中，我是先将权重载入全部网络结构中。此时，模型的最后一层大小并不是我想要的，因此我获取了输入到最后一层全连接层之前的维度大小，然后根据数据集的类别数来指定最后输出的out\_features数目，以此代替原来的全连接层。

你也可以先定义好具有指定全连接大小的网络结构，然后除了最后一层全连接层之外，全部层都载入预训练模型；你也可以先将权重载入全部网络结构中，然后删掉最后一层全连接层，最后再加入一层指定大小的全连接层。

方法很多，读者朋友们都可以试试。

## 文末

可以看出，不管是微调还是冻结，甚至是修改网络方法、方式都很多，需要根据自身地情况去调整，去适应。至于哪种方式最好，只有通过具体实践才能领悟；又或许没有最好的方式，只有最适合这种场景的方式。

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

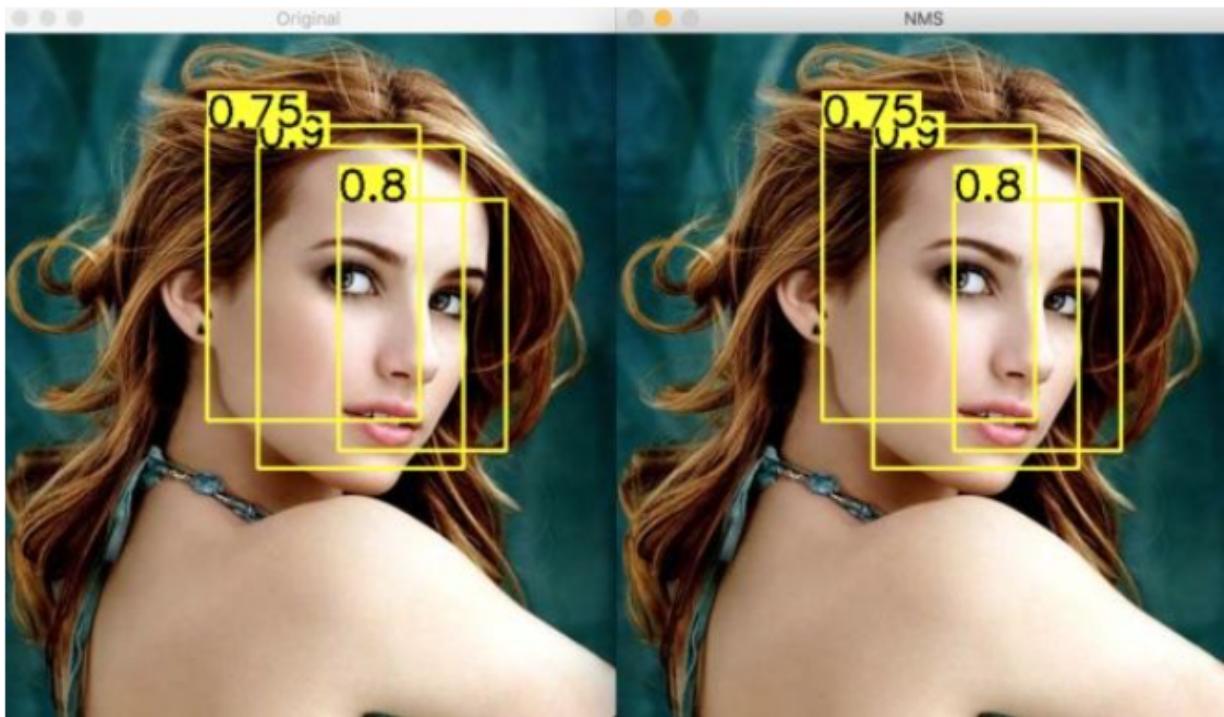
## NMS

### 前言

当两个预测框十分贴近时，先以具有最高置信度的那个框作为基准，计算两者的IOU重合程度，如果重合度超过阈值，就去掉置信度低的，只保留高的。重复这个过程，直至所有框的重合度低于阈值。在目标检测中，常用NMS来消除冗余的预测框，最终只留下少量的预测框。

### NMS原理

NMS也叫做非极大抑制值。在目标检测网络中，不论one-stage还是two-stage，都会产生许多预测框，它们大多都指向了同一个目标，因此需要通过极大值抑制来筛选掉多余的预测框，来找到每个目标最优的预测框。



NMS的本质是搜索局部极大值，抑制非极大值元素。非极大值抑制，主要就是用来抑制检测时冗余的框。因为在目标检测中，在同一目标的位置上会产生大量的候选框，这些候选框相互之间可能会有重叠，所以我们需要利用非极大值抑制找到最佳的目标边界框，消除冗余的边界框。

## NMS流程

1. 对所有预测框的置信度降序排序；
2. 选出置信度最高的预测框，确认其为正确预测，并计算他与其他预测框的 IOU；
3. 根据步骤2中计算的 IOU 去除重叠度高的， $IOU > \text{阈值}$  就直接删除；
4. 剩下的预测框返回第1步，直到没有剩下的为止。

一般来说，NMS一次处理只会一个类别，所以如果有N个类别，那么就需要执行N次。但你也可以只进行一次NMS就处理全部类。

## 代码实现

### 使用Pytorch自己实现NMS

```
import torch
import cv2
import numpy as np

def box_area(boxes):

    return (boxes[:, 2] - boxes[:, 0]) * (boxes[:, 3] - boxes[:, 1])
```

```

def box_iou(boxes1, boxes2):

    area1 = box_area(boxes1) # 每个框的面积 (N,)
    area2 = box_area(boxes2) # (M,)

    lt = torch.max(boxes1[:, None, :2], boxes2[:, :2]) # [N,M,2] # N中一个和M
    个比较; 所以由N, M 个
    rb = torch.min(boxes1[:, None, 2:], boxes2[:, 2:]) # [N,M,2]

    wh = (rb - lt).clamp(min=0) # [N,M,2] # 小于0的为0 clamp 锯; 夹锯;
    inter = wh[:, :, 0] * wh[:, :, 1] # [N,M]

    iou = inter / (area1[:, None] + area2 - inter)
    return iou # NxM, boxes1中每个框和boxes2中每个框的IoU值;

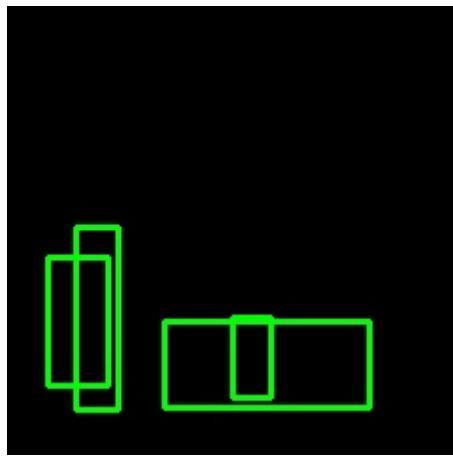
def ori_nms(boxes, scores, iou_threshold):
    """
    :param boxes: [N, 4], 此处传进来的框, 是经过筛选(NMS之前选取过得分TopK)之
    后, 在传入之前处理好的;
    :param scores: [N]
    :param iou_threshold: 0.7
    :return:
    """

    keep = [] # 最终保留的结果, 在boxes中对应的索引;
    idxs = scores.argsort() # 值从小到大的 索引
    while idxs.numel() > 0: # 循环直到null; numel(): 数组元素个数
        # 得分最大框对应的索引, 以及对应的坐标
        max_score_index = idxs[-1]
        max_score_box = boxes[max_score_index][None, :] # [1, 4]
        keep.append(max_score_index)
        if idxs.size(0) == 1: # 就剩余一个框了;
            break
        idxs = idxs[:-1] # 将得分最大框 从索引中删除; 剩余索引对应的框 和 得分最
        大框 计算IoU;
        other_boxes = boxes[idxs] # [?, 4]
        ious = box_iou(max_score_box, other_boxes) # 一个框和其余框比较 1XM
        idxs = idxs[ious[0] <= iou_threshold]

    keep = idxs.new(keep) # Tensor
    return keep

```

下面对四个box来进行NMS操作，并通过OpenCV来看这四个box的位置。



```
img = np.zeros([224, 224, 3], np.uint8)
box1 = [20, 50, 125, 189]
box2 = [34, 55, 110, 201]
box4 = [78, 180, 157, 200]
box3 = [112, 131, 155, 195]

cv2.rectangle(img, (box1[0], box1[1]), (box1[2], box1[3]), (0, 255, 0), 2)
cv2.rectangle(img, (box2[0], box2[1]), (box2[2], box2[3]), (0, 255, 0), 2)
cv2.rectangle(img, (box3[0], box3[1]), (box3[2], box3[3]), (0, 255, 0), 2)
cv2.rectangle(img, (box4[0], box4[1]), (box4[2], box4[3]), (0, 255, 0), 2)
cv2.imwrite('test.jpg', img)

box1 = [20.0, 50.0, 125.0, 189.05]
box2 = [34.0, 55.0, 110.0, 201.0]
box4 = [78.0, 180.0, 157.0, 200.0]
box3 = [112.0, 131.0, 155.0, 195.0]

bbox = torch.tensor([box1, box2, box3, box4])
score = torch.tensor([0.5, 0.3, 0.2, 0.4])

output = ori_nms(boxes=bbox, scores=score, iou_threshold=0.3)
print(output)
```

我们知道，通过目标检测算法会得出框的位置以及置信度，所以bbox是四个box的集合，而score是四个box的置信度列表。通过运行以上代码，可以得到：

```
tensor([0, 3, 2])
```

可以看到，因为其中有两个框的IOU较大，所以通过NMS算法过滤掉了其中一个。

## 调用封装好的NMS

```
from torchvision.ops import nms

keep = nms(boxes=bbox, scores=score, iou_threshold=0.3)
print(keep)
```

得出的结果如下：

```
tensor([0, 3, 2])
```

可以看到，通过两种方式得出来的结果是一样的。

## NMS有哪些缺陷

NMS的阈值不好确定。阈值低了容易漏检，阈值高了又容易误检。并且计算IOU的方式有缺陷。如果两个框没有相交，根据定义， $\text{IOU}=0$ ，不能反映两者之间的距离大小，即重合度。

## IOU

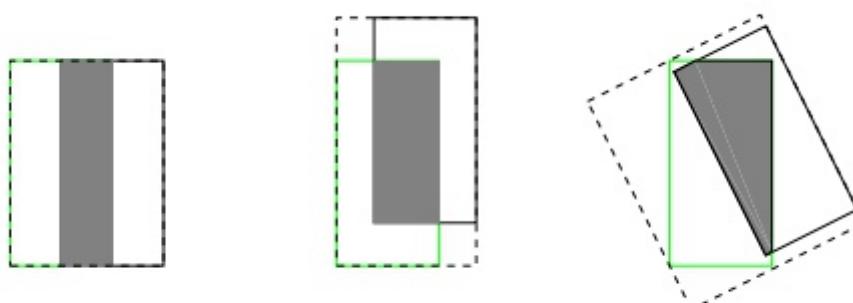
IOU就是我们一直所说的交并比，是目标检测中最常用的指标，在 anchor-based 的方法中，它的作用不仅用来确定正样本和负样本，还可以用来评价输出框和 ground-truth 之间的距离。

可以说它可以反映预测检测框与真实检测框的检测效果。

它还有一个很好的特性就是尺度不变性，也就是对尺度不敏感，在回归任务中，判断输出框和 ground-truth 的距离最直接的指标就是 IOU。因为其满足非负性；同一性；对称性；三角不等性。

IOU的缺点：

- 如果两个框没有相交，根据定义， $\text{IOU} = 0$ ，不能反映两者之间的距离大小，即重合度。同时因为 loss = 0 时，没有梯度回传，无法进行学习训练。
- IOU 无法精确反映两者的重合度大小。如下图所示，三种情况 IOU 都相等，但看得出来他们的重合度是不一样的，左边的图回归的效果最好，右边的图最差。



## GIOU

GIOU对scale不敏感:

- GIOU 是 IOU 的下界，在两个框无限重合的情况下， $\text{IOU} = \text{GIOU} = 1$ 。
- IOU取值[0,1]，但 GIOU 有对称区间，取值范围[-1,1]。在两者重合的时候取最大值1，在两者无交集且无限远的时候取最小值-1，因此 GIOU 是一个非常好的距离度量指标。

GIOU与 IOU 只关注重叠区域不同，GIOU 不仅关注重叠区域，还关注其他的非重合区域，能更好的反映两者的重合度。

GIOU会先计算两个框的最小闭包区域面积，通俗理解：同时包含了预测框和真实框的最小框的面积，再计算出 IOU，再计算闭包区域中不属于两个框的区域占闭包区域的比重，最后用 IOU 减去这个比重，最后得到 GIOU。

当矩形框同宽高并且平行或者垂直的话，退化成 IOU。

## DIOU

1. 将目标与anchor之间的距离，重叠率以及尺度都考虑进去，使得目标框回归变得更加稳定，不会像IOU和GIOU一样出现训练过程中发散等问题。
2. 与 GIOU loss类似，DIOU loss、在与目标框不重叠时，仍然可以为边界框提供移动方向。
  1. DIOU loss可以直接最小化两个目标框的距离，因此比GIOU loss收敛快得多。
  2. 对于包含两个框在水平方向和垂直方向上这种情况，
3. DIOU 损失可以使回归非常快，而 GIOU 损失几乎退化为 IOU 损失。DIOU 还可以替换普通的 IOU 评价策略，应用于NMS中，使得NMS得到的结果更加合理和有效。

## CIOU

论文考虑到bbox回归三要素中的长宽比还没被考虑到计算中，因此，进一步在DIOU的基础上提出了CIOU。其惩罚项如下面公式：

$$\mathcal{R}_{CIoU} = \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{c^2} + \alpha v$$

$$v = \frac{4}{\pi^2} \left( \arctan \frac{w^{gt}}{h^{gt}} - \arctan \frac{w}{h} \right)^2$$

其中  $\alpha$  是权重函数，而  $v$  用来度量长宽比的相似性。在使用的过程中，要考虑  $v$  的梯度，长宽比在  $[0, 1]$  情况下，容易导致梯度爆炸。

## 代码实现

以下是YOLOv5中，集成各个这四个IOU损失函数的实现：

```
def bbox_iou(box1, box2, x1y1x2y2=True, GIoU=False, DIoU=False, CIoU=False, eps=1e-7):
    # Returns the IoU of box1 to box2. box1 is 4, box2 is nx4
    box2 = box2.T

    # Get the coordinates of bounding boxes
    if x1y1x2y2: # x1, y1, x2, y2 = box1
        b1_x1, b1_y1, b1_x2, b1_y2 = box1[0], box1[1], box1[2], box1[3]
        b2_x1, b2_y1, b2_x2, b2_y2 = box2[0], box2[1], box2[2], box2[3]
    else: # transform from xywh to xxyy
        b1_x1, b1_x2 = box1[0] - box1[2] / 2, box1[0] + box1[2] / 2
        b1_y1, b1_y2 = box1[1] - box1[3] / 2, box1[1] + box1[3] / 2
        b2_x1, b2_x2 = box2[0] - box2[2] / 2, box2[0] + box2[2] / 2
        b2_y1, b2_y2 = box2[1] - box2[3] / 2, box2[1] + box2[3] / 2

    # Intersection area
    inter = (torch.min(b1_x2, b2_x2) - torch.max(b1_x1, b2_x1)).clamp(0) * \
            (torch.min(b1_y2, b2_y2) - torch.max(b1_y1, b2_y1)).clamp(0)

    # Union Area
    w1, h1 = b1_x2 - b1_x1, b1_y2 - b1_y1 + eps
    w2, h2 = b2_x2 - b2_x1, b2_y2 - b2_y1 + eps
    union = w1 * h1 + w2 * h2 - inter + eps

    iou = inter / union
    if GIoU or DIoU or CIoU:
        cw = torch.max(b1_x2, b2_x2) - torch.min(b1_x1, b2_x1) # convex
        (smallest enclosing box) width
        ch = torch.max(b1_y2, b2_y2) - torch.min(b1_y1, b2_y1) # convex
        height
        if CIoU or DIoU: # Distance or Complete IoU
            https://arxiv.org/abs/1911.08287v1
            c2 = cw ** 2 + ch ** 2 + eps # convex diagonal squared
            rho2 = ((b2_x1 + b2_x2 - b1_x1 - b1_x2) ** 2 +
                    (b2_y1 + b2_y2 - b1_y1 - b1_y2) ** 2) / 4 # center
            distance squared
            if DIoU:
                return iou - rho2 / c2 # DIoU
            elif CIoU: # https://github.com/Zzh-tju/DIoU-SSD-
                pytorch/blob/master/utils/box/box_utils.py#L47
```

```

v = (4 / math.pi ** 2) * torch.pow(torch.atan(w2 / h2) -
torch.atan(w1 / h1), 2)
    with torch.no_grad():
        alpha = v / (v - iou + (1 + eps))
    return iou - (rho2 / c2 + v * alpha) # CIoU
else: # GIoU https://arxiv.org/pdf/1902.09630.pdf
    c_area = cw * ch + eps # convex area
    return iou - (c_area - union) / c_area # GIoU
else:
    return iou # IoU

```

## IOU方法总结

IOU\_Loss：主要考虑检测框和目标框重叠面积。 GIOU\_Loss：在 IOU 的基础上，解决边界框不重合时的问题。 DIOU\_Loss：在 IOU 和 GIOU 的基础上，考虑边界框中心点距离的信息。 CIoU\_Loss：在 DIOU 的基础上，考虑边界框宽高比的尺度信息。

## NMS的改进方法

### soft-nms

可以不要那么暴力地删除所有IOU大于阈值的框，而是降低其置信度。

### softer-nms

传统NMS用到的score仅仅是分类置信度得分，不能反映预测框的定位精准度，既分类置信度和定位置信非正相关的。

softer-nms是基于soft-nms的，对预测标注方差范围内的候选框加权平均，使得高定位置信度的预测框具有较高的分类置信度。其实很简单，预测的四个顶点坐标，分别对IOU > Nt的预测加权平均计算，得到新的4个坐标点。

### DIOU-nms

使用DIOU的方式计算IOU。DIOU是将框与框之间的距离，重叠率以及尺度都考虑进去进行计算。

### adaptive nms

自适应地调整NMS阈值，当检测目标不密集时，就使用较低的NMS阈值去掉其他冗余框；当检测目标密集出现许多重叠时，就使用较高的NMS阈值尽可能保留想要的框。

## weighted nms

weighted nms也就是加权非极大值抑制，它认为，NMS每次迭代所选出的最大得分框未必是精确定位的，冗余框也有可能是定位良好的。

weighted nms与NMS相比，是在过滤矩形框的过程中，并没有直接将那些与当前矩形框IOU大于阈值，且类别相同的框直接剔除掉，而是根据网络预测的置信度进行加权，来得到新的矩形框，然后把该矩形框代替之前的矩形框。这里加权平均的对象包括这个框以及其他IOU大于NMS阈值的相邻框。在做加权的时候，本身的矩形框也会根据置信度一起进行加权。

## 文末

以上就是关于NMS的相关知识。在目标检测中，不管是模型训练时还是模型推理后处理时，NMS和IOU都可以说是标配了，对于模型精度的提升还是有所帮助的。

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## 参数量、计算量统计

## 前言

在Pytorch中已经有封装好的函数来查看模型的结构、模型的参数，以及计算模型的参数量。下面我们来看看它们是如何实现的。

### param.numel()

在Pytorch中，统计模型的参数量可以使用param.numel()函数来实现。

```
def get_parameter_number(model):
    total_num = sum(p.numel() for p in model.parameters())
    trainable_num = sum(p.numel() for p in model.parameters() if
p.requires_grad)
    return {'Total': str(total_num / 1000 ** 2) + 'M', 'Trainable':
str(trainable_num / 1000 ** 2) + 'M'}

print(get_parameter_number(net))
```

结果为：

```
{'Total': '23.710883M', 'Trainable': '23.710883M'}
```

## thop的profile

统计参数量也可以通过thop的profile来实现，除了Params之外还可以统计FLOPS。

```
from thop import profile
from torchvision.models.resnet import resnet50
import torch

dummy_input = torch.rand(1, 3, 224, 224).cpu()
net = resnet50(num_classes=99, pretrained=False).cpu()
flops, params = profile(net, inputs=(dummy_input, ))
print("-" * 50)
print('FLOPS =' + str(flops / 1000 ** 3) + 'G')
print('Params =' + str(params / 1000 ** 2) + 'M')
```

结果为：

```
FLOPS =4.1318973446
Params =23.710883M
{'Total': '23.710883M', 'Trainable': '23.710883M'}
```

## Pytorch中查看模型结构、模型参数的函数

### model.state\_dict()

这个函数应该是我们最为熟知的，通过model.state\_dict()返一个字典，然后将预训练模型的权重载入到模型结构中去。因为它本身就是一个字典，所以可以直接修改模型各层的参数，在参数剪枝上，使用起来特别方便。

```
net.state_dict
<bound method Module.state_dict of ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
)
```

## model.modules()

这个函数会返回一个生成器，是一个可迭代的变量。通过model.modules()可以迭代地遍历模型的所有子层。

```
[ResNet(
  (conv1): C...as=True)
  ), Conv2d(3, 64, kernel...ias=False), BatchNorm2d(64, eps=...tats=True), ReLU(inplace=True), MaxPool2d(kernel_siz...ode=False), Sequential(
    (0): B...rue)
  ),
  ), Bottleneck(
    (conv1...rue)
  )]
```

## model.children()

model.children()只会遍历模型的子层，而不会迭代地进行遍历所有子层，而model.modules()能够迭代地遍历模型的所有子层。

```
> [x for x in net.children()]
[Conv2d(3, 64, kernel...ias=False), BatchNorm2d(64, eps=...tats=True), ReLU(inplace=True), MaxPool2d(kernel_siz...ode=False), Sequential(
  (0): B...rue)
),
 Sequential(
  (0): B...rue)
),
 Sequential(
  (0): B...rue)
),
 Sequential(
  (0): B...rue)
),
 AdaptiveAvgPool2d(ou...ze=(1, 1)), Linear(in_features=2...bias=True)]
```

这里做个对比：

```
> 127: Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
> 128: BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 129: ReLU(inplace=True)
> 130: Sequential(
> 131: Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
> 132: BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 133: Bottleneck(
> 134: Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
> 135: BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 136: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
> 137: BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 138: Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
> 139: BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 140: ReLU(inplace=True)
> 141: Bottleneck(
> 142: Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
> 143: BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 144: Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
> 145: BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 146: Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
> 147: BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 148: ReLU(inplace=True)
> 149: AdaptiveAvgPool2d(output_size=(1, 1))
> 150: Linear(in_features=2048, out_features=99, bias=True)
len(): 151
```

```
> 0: Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
> 1: BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
> 2: ReLU(inplace=True)
> 3: MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
> 4: Sequential(
> 5: Sequential(
> 6: Sequential(
> 7: Sequential(
> 8: AdaptiveAvgPool2d(output_size=(1, 1))
> 9: Linear(in_features=2048, out_features=99, bias=True)
len(): 10
```

可以看出，使用model.modules()会把所有的层都遍历出来，而model.children()只会遍历第一层子层。

## model.parameters()

model.parameters()则是迭代地返回模型的所有参数。

```
> 159: Parameter containing:
> 160: Parameter containing:
> special variables
> function variables
> T: tensor([-1.5530e-02, -1.1580e-02,  1.0413e-02,  8.8534e-03,  4.1082e-03,
> data: tensor([-1.5530e-02, -1.1580e-02,  1.0413e-02,  8.8534e-03,  4.1082e-03,
> device: device(type='cuda', index=0)
> dtype: torch.float32
> grad: None
> grad_fn: None
```

## model.named\_modules() model.named\_parameters()

## model.named\_children()

和

顾名思义，它们就是带有名字的model.modules()、model.children()、model.parameters()。model.named\_modules()、model.named\_children()这两个函数不仅会返回模型的子层，还会带上这些层的名字。

```
> 000: ('', ResNet(  
> 001: ('conv1', Conv2d(3, 64, kernel...ias=False))  
> 002: ('bn1', BatchNorm2d(64, eps=...tats=True))  
> 003: ('relu', ReLU(inplace=True))  
> 004: ('maxpool', MaxPool2d(kernel_siz...ode=False))  
> 005: ('layer1', Sequential(  
> 006: ('layer1.0', Bottleneck(  
> 007: ('layer1.0.conv1', Conv2d(64, 64, kerne...ias=False))  
> 008: ('layer1.0.bn1', BatchNorm2d(64, eps=...tats=True))  
> 009: ('layer1.0.conv2', Conv2d(64, 64, kerne...ias=False))  
> 010: ('layer1.0.bn2', BatchNorm2d(64, eps=...tats=True))  
> 011: ('layer1.0.conv3', Conv2d(64, 256, kern...ias=False))  
> 012: ('layer1.0.bn3', BatchNorm2d(256, eps...tats=True))  
> 013: ('layer1.0.relu', ReLU(inplace=True))  
> 014: ('layer1.0.downsample', Sequential(  
> 015: ('layer1.0.downsample.0', Conv2d(64, 256, kern...ias=False))  
> 016: ('layer1.0.downsample.1', BatchNorm2d(256, eps...tats=True))  
> 017: ('layer1.1', Bottleneck(  
> 018: ('layer1.1.conv1', Conv2d(256, 64, kern...ias=False))  
> 019: ('layer1.1.bn1', BatchNorm2d(64, eps=...tats=True))  
> 020: ('layer1.1.conv2', Conv2d(64, 64, kerne...ias=False))  
> 021: ('layer1.1.bn2', BatchNorm2d(64, eps=...tats=True))  
> 022: ('layer1.1.conv3', Conv2d(64, 256, kern...ias=False))  
> 023: ('layer1.1.bn3', BatchNorm2d(256, eps...tats=True))  
> 024: ('layer1.1.relu', ReLU(inplace=True))  
> 025: ('layer1.2', Bottleneck(  
> 026: ('layer1.2.conv1', Conv2d(256, 64, kern...ias=False))  
> 027: ('layer1.2.bn1', BatchNorm2d(64, eps=...tats=True))
```

```
> 0: ('conv1', Conv2d(3, 64, kernel...ias=False))  
> 1: ('bn1', BatchNorm2d(64, eps=...tats=True))  
> 2: ('relu', ReLU(inplace=True))  
> 3: ('maxpool', MaxPool2d(kernel_siz...ode=False))  
> 4: ('layer1', Sequential(  
> 5: ('layer2', Sequential(  
> 6: ('layer3', Sequential(  
> 7: ('layer4', Sequential(  
> 8: ('avgpool', AdaptiveAvgPool2d(ou...ze=(1, 1)))  
> 9: ('fc', Linear(in_features=2...bias=True))
```

model.named\_parameters()就是迭代地返回带有名字的参数,会给每个参数加上带有.weight或者.bias的名字来区分是权重还是偏置。

```
[('conv1.weight', Parameter containing...grad=True)), ('bn1.weight', Parameter containing...grad=True)), ('bn1.bias', Parameter containing...grad=True)), ('layer1.0.bn1.weight', Parameter containing...grad=True)), ('layer1.0.bn1.bias', Parameter containing...grad=True)), ('layer1.0.conv2.weight', Parameter containing...grad=True)), ('layer1.0.bn2.weight', Parameter containing...grad=True)), ('layer1.0.bn2.bias', Parameter containing...grad=True)), ('layer1.0.conv3.weight', Parameter containing...grad=True)), ('layer1.0.bn3.weight', Parameter containing...grad=True)), ('layer1.0.bn3.bias', Parameter containing...grad=True)), ('layer1.0.downsample.0.weight', Parameter containing...grad=True)), ('layer1.0.downsample.0.bias', Parameter containing...grad=True)), ...]
```

## 参数量

在卷积神经网络中，一个卷积核的参数 =  $k \times k \times C_{in} + 1$ ，其中1代表偏置。

一个卷积层的参数 = (一个卷积核的参数)  $\times$  卷积核的数目 =  $(k \times k \times C_{in} + 1) \times C_{out} = k \times k \times C_{in} \times C_{out} + C_{out}$ 。

## FLOPS和FLOPs

- FLOPS：S是大写，代表floating point operations per second的缩写，指的是每秒浮点运算次数，可以理解为计算速度，是一个衡量硬件性能的指标。
- FLOPs：s是小写，代表floating point operations的缩写，其中s代表复数，指的是浮点运算数，可以理解为计算量，用来衡量模型的复杂度。
- 在大部分模型的加速和压缩对比加速效果中，所用的指标都是FLOPS，这个指标主要衡量的就是乘法和加法指令的数量。
- **但是它们几个之间并不是正向的关系，FLOPS并不能完全能够衡量模型的速度，以及模型的参数量少也不代表FLOPs低，并且理论的FLOPs低也不代表实际的推理速度快，因为还受计算平台、模型并行度等等的约束。**

## 浮点运算量和参数量的区别

- 浮点运算量是模型在实际推理过程时，加减乘除这些运算过程中的计算次数，描述的是计算力；
- 参数量指的是模型的大小，和输入的图片大小无关，描述的是所需要的内存；

## GFLOP

GFLOP其实是一个单位， $1\text{GLOPs}=10\text{亿次浮点运算}$ ，是学术界中论文里比较流行的单位。

## 高效设计网络的准则

不管是评估参数量还是计算量，目的都是为了评估这个网络是否具有高效性，在ShuffleNetV2论文中，提出了四条高效设计网络的准则。

### 1.当卷积层的输入与输出的通道数相等时，内存访问时间MAC最小

假设一个 $1 \times 1$ 卷积层的输入特征图通道数是 $c_1$ ，输入特征图尺寸是 $h$ 和 $w$ ，输出特征通道数是 $c_2$ ，那么这样一个 $1 \times 1$ 卷积层的FLOPs就是： $B=c_1 \times c_2 \times h \times w \times 1 \times 1$ 。

		GPU (Batches/sec.)			ARM (Images/sec.)			
c1:c2	(c1,c2) for $\times 1$	$\times 1$	$\times 2$	$\times 4$	(c1,c2) for $\times 1$	$\times 1$	$\times 2$	$\times 4$
1:1	(128,128)	1480	723	232	(32,32)	76.2	21.7	5.3
1:2	(90,180)	1296	586	206	(22,44)	72.9	20.5	5.1
1:6	(52,312)	876	489	189	(13,78)	69.1	17.9	4.6
1:12	(36,432)	748	392	163	(9,108)	57.6	15.1	4.4

Table 1: Validation experiment for **Guideline 1**. Four different ratios of number of input/output channels ( $c_1$  and  $c_2$ ) are tested, while the total FLOPs under the four ratios is fixed by varying the number of channels. Input image size is  $56 \times 56$ .

因为是 $1 \times 1$ 卷积，所以它的输入特征和输出特征尺寸是一样的，这里用 $h$ 和 $w$ 表示，其中 $h \times w \times c_1$ 表示输入特征所需的存储空间， $h \times w \times c_2$ 表示输出特征所需的存储空间， $c_1 \times c_2$ 表示卷积核所需的存储空间。

所以最终 $MAC = h \times w (c_1 + c_2) + c_1 \times c_2$

根据均值不等式可以得出：

$$MAC \geq 2\sqrt{hwB} + \frac{B}{hw}.$$

把 $MAC$ 和 $B$ 带入式子可得，当 $c_1=c_2$ ，也就是当输入特征的通道数和输出特征的通道数相等时，在固定FLOPs时， $MAC$ 可以达到最小值。

## 2. 当组卷积中组group的数目增大时， $MAC$ 也会变大

带group操作的 $1 \times 1$ 卷积的FLOPs为 $B = h \times w \times c_1 \times c_2 / g$ ， $g$ 代表group数量。这是因为每个卷积核都只和 $c_1 / g$ 个输入通道的特征做卷积运算，所以在式子中多除以了一个 $g$ 。

		GPU (Batches/sec.)			CPU (Images/sec.)			
g	c for $\times 1$	$\times 1$	$\times 2$	$\times 4$	c for $\times 1$	$\times 1$	$\times 2$	$\times 4$
1	128	2451	1289	437	64	40.0	10.2	2.3
2	180	1725	873	341	90	35.0	9.5	2.2
4	256	1026	644	338	128	32.9	8.7	2.1
8	360	634	445	230	180	27.8	7.5	1.8

Table 2: Validation experiment for **Guideline 2**. Four values of group number  $g$  are tested, while the total FLOPs under the four values is fixed by varying the total channel number  $c$ . Input image size is  $56 \times 56$ .

同样 $MAC$ 为 $h \times w \times (c_1 + c_2) + c_1 \times c_2 / g$ ，因此 $MAC$ 和 $B$ 之间的关系为 $MAC = h \times w \times (c_1 + c_2) + c_1 \times c_2 / g = h \times w \times c_1 + B \times g / c_1 + B / (h \times w)$

这样可以看出，在固定 $B$ 时，组卷积的 $g$ 变大， $MAC$ 也会变大，因此，模型也会变慢。

## 3. 网络碎片化程度越高，速度越慢，即分支越多，模型越慢。

因为分支越多对并行计算越不利，在GPU上影响比较明显，而在ARM上的影响相对来说会小一些。

	GPU (Batches/sec.)			CPU (Images/sec.)		
	c=128	c=256	c=512	c=64	c=128	c=256
1-fragment	2446	1274	434	40.2	10.1	2.3
2-fragment-series	1790	909	336	38.6	10.1	2.2
4-fragment-series	752	745	349	38.4	10.1	2.3
2-fragment-parallel	1537	803	320	33.4	9.1	2.2
4-fragment-parallel	691	572	292	35.0	8.4	2.1

Table 3: Validation experiment for **Guideline 3**.  $c$  denotes the number of channels for *1-fragment*. The channel number in other fragmented structures is adjusted so that the FLOPs is the same as *1-fragment*. Input image size is  $56 \times 56$ .

其中fragment为网络的支路数量。

#### 4.element-wise的影响不容小觑

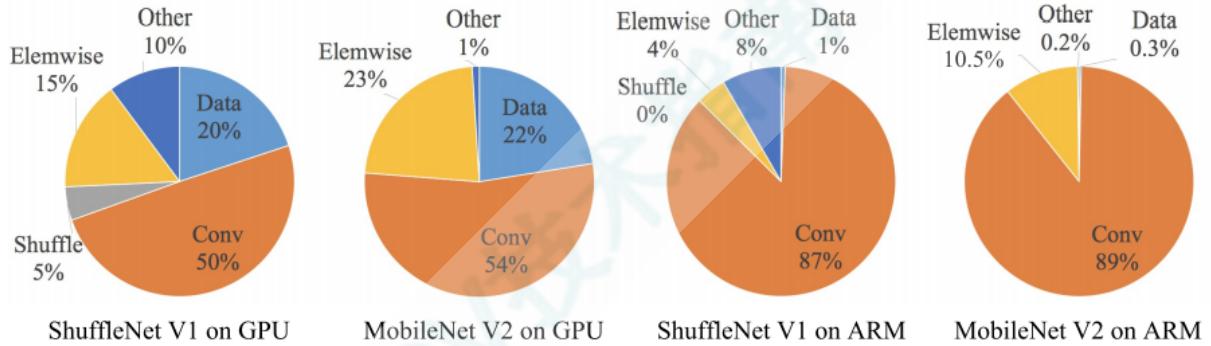


Fig. 2: Run time decomposition on two representative state-of-the-art network architectures, *ShuffleNet v1* [15] ( $1\times$ ,  $g = 3$ ) and *MobileNet v2* [14] ( $1\times$ ).

FLOPS主要表示的是卷积层的时间消耗，而element-wise的操作虽然不怎么增加FLOPS，但element-wise如ReLU、shortcut等操作带来的影响不可忽视，这些操作所带来的时间消耗远比在FLOPS上体现的要高，因此要尽量减少element-wise操作。

		GPU (Batches/sec.)			CPU (Images/sec.)		
ReLU	short-cut	c=32	c=64	c=128	c=32	c=64	c=128
yes	yes	2427	2066	1436	56.7	16.9	5.0
yes	no	2647	2256	1735	61.9	18.8	5.2
no	yes	2672	2121	1458	57.3	18.2	5.1
no	no	2842	2376	1782	66.3	20.2	5.4

Table 4: Validation experiment for **Guideline 4**. The ReLU and shortcut operations are removed from the “bottleneck” unit [4], separately.  $c$  is the number of channels in unit. The unit is stacked repeatedly for 10 times to benchmark the speed.

# 文末

通过查看这些参数量、计算量这些数值，就可以对模型的大小、运行速度、占用资源有个大致的了解。对于参数量、计算量的统计，更多是作为是日常工具，而对于这一些日常使用的工具，我认为实践会比理论更加重要。

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## *Visdom*可视化

### 前言

**Visdom**是Facebook在2017年提供的一套为Pytorch提供可视化的工具。在此之前也有一些非常优秀的库比如**tensorboardX**通过调用**tensorboard**为Pytorch提供便捷的可视化操作，但是毕竟不是Pytorch原生支持的可视化工具，同时也需要额外调用**tensorflow**提供的能力，总觉得不美，宇宙第一DL框架还需要寄TF篱下么？这不，就有了Visdom。Visdom的功能非常的强大同时使用起来也很方便，本文会简单介绍一些Visdom的基础概念以及一些通用的使用方法。

### 安装

直接通过pip方式进行安装。

```
pip install visdom
```

### 服务开启

#### python启动服务

```
"""
-port 指定服务端口
-logging_level 指定日志级别，这个默认就为INFO，可以不用修改
"""
python -m visdom.server -port 8097 -logging_level INFO
```

其他一些相关启动服务参数请参考visdom官方github:

<https://github.com/fossasia/visdom>

有几个**比较关键的参数**单独拎出来说明一下：

```
-enable_login 指定参数后查看可视化信息需要账号密码登录  
-bind_local 指定参数后只能通过本地访问服务，也就是说如果你是在远程服务器上启动了visdom服务，指定这个参数就无法从本地访问了，慎用
```

服务启动后可以看到一些Log信息：

```
It's Alive!  
INFO:root:Application Started  
You can navigate to http://localhost:8098
```

这样就可以直接在浏览器端通过【ip:port】的格式访问这个服务了。

## 浏览器访问

通过浏览器访问服务的时候终端也会输出一些Log信息，从这个Log信息我们可以看到visdom采用了tornado web服务器与应用框架为我们提供基于浏览器的可视化信息展示~

```
INFO:tornado.access:200 POST /env/develop (172.16.33.144) 0.58ms  
INFO:tornado.access:200 POST /compare/develop+main (172.16.33.144) 0.65ms  
INFO:tornado.access:200 POST /env/main (172.16.33.144) 0.44ms  
INFO:tornado.access:200 GET / (172.16.33.144) 26.30ms  
INFO:tornado.access:101 GET /socket (172.16.33.144) 1.06ms
```

## 基本概念

Visdom中很多基本概念其实都来自于**图形图像界面交互**中的概念，下面是几个比较重要的概念。

### Enviroment

一个Enviroment代表一个初始化的‘**环境**’，每个环境下可能包含有多个**Window**窗口，可以通过切换Enviroment切换不同的环境，每个环境包含不同的窗口对象，可以自由方便的进行实验管理。

### Window

窗口对象，内部可以包含**图像，文本，视频**等数据。对应到Pytorch训练过程中可以存在多个窗口内容，比如用作日志的文本数据记录窗口，用作Loss曲线绘制的观察窗口等。

### Filter

可以通过使用正则表达式动态地**过滤Enviroment中的窗口**。这在你的Envirometn中存在许多个窗口时会非常有帮助，比如筛选查看训练中某个loss的下降情况等。

# 使用Visdom

Visdom的使用非常简单，很多情况下两步就可以完成数据的可视化：

1. 创建Visdom对象
2. 调用对应方法构建窗口内容

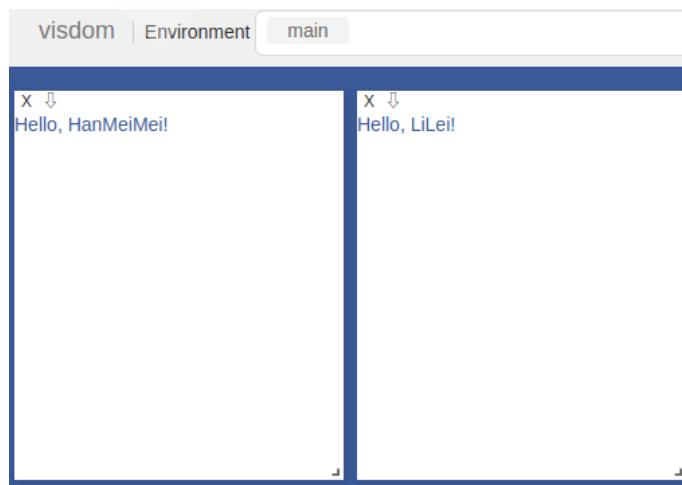
下面从多个不同的场景来介绍visdom的使用方式

## 多Enviroment多窗口管理

```
import visdom
import numpy as np

# env main
vis_main = visdom.Visdom() #env默认为'main'
# 添加一个窗口，窗口名为'one'，窗口内容为文本内容'Hello, HanMeiMei!'
vis_main.text('Hello, HanMeiMei!', win='one')
# 添加另一个窗口，窗口名为'two'，窗口内容为文本内容'Hello, LiLei!'
# append参数设置为True时可以在同一个窗口下继续添加内容而不覆盖之前的内容
vis_main.text('Hello, LiLei!', win='two', append=False)
# env develop
vis_develop = visdom.Visdom(env='develop')
vis_develop.text('Hello, LiLei!')
```

如下图所示可以通过Enviroment来切换不同的显示界面，同一个Enviroment下也可以展示多个窗口内容。同时还可以点击窗口左上角朝下的箭头来将窗口中的内容保存到本地。





## 监听Loss等曲线

在训练过程中需要监听Loss曲线的变化让我们实时掌握训练中的各种情况，比如Loss是否收敛、是否出现过拟合等等。

```

import visdom
import numpy as np

# loss
x, y = 0, 0
vis = visdom.Visdom()
# 首先建立一个窗口
window_train = vis.line(
    X=np.array([x]), #指定X的起始坐标
    Y=np.array([y]), #指定Y的起始坐标
    opts=dict(title='Train Loss'))
window_val = vis.line(
    X=np.array([x]), #指定X的起始坐标
    Y=np.array([y]), #指定Y的起始坐标
    opts=dict(title='Val Loss')) #指定window的title
for step in range(10):
    loss_train = 10 - step + 0.1
    vis.line(
        X=np.array([step]),
        Y=np.array([loss_train]),

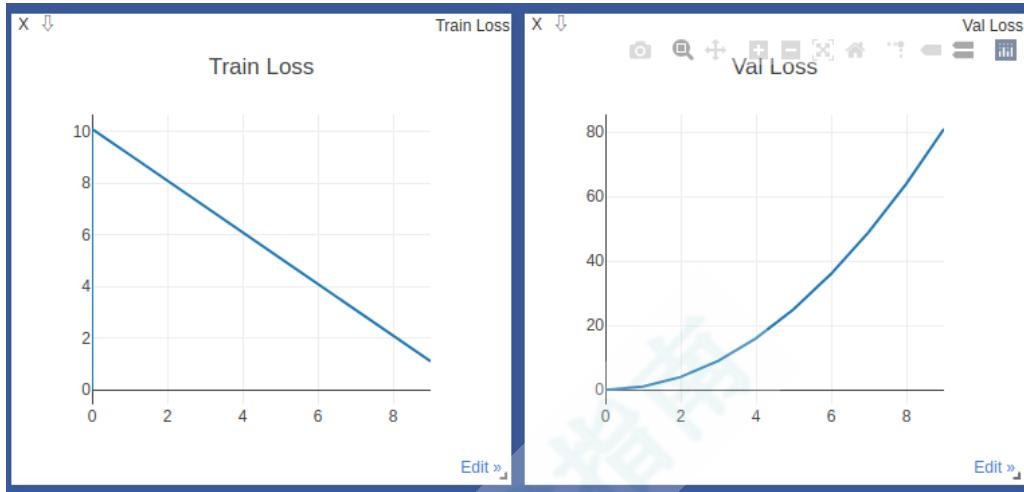
```

```

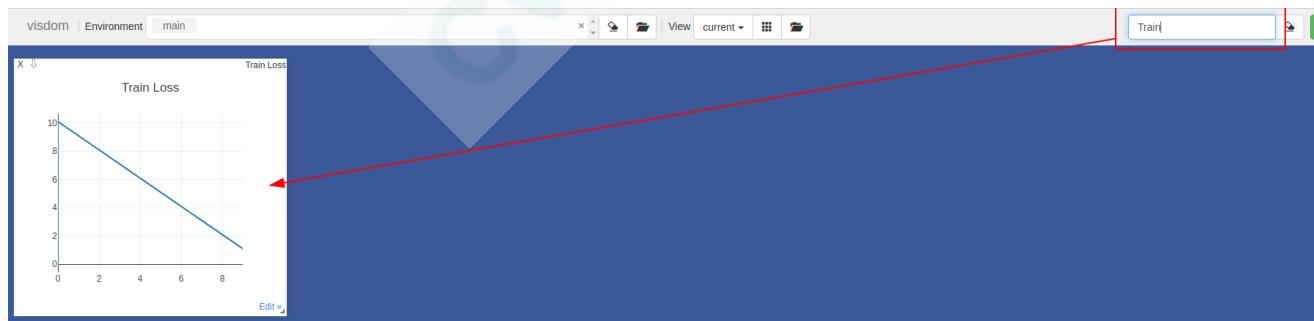
win=window_train, #指定在上面创建好的window内展示
update='append')
loss_val = step * step
vis.line(
X=np.array([step]),
Y=np.array([loss_val]),
win=window_val, #指定在上面创建好的window内展示
update='append')

```

思考一下，从下面这张图的Train Loss和Val Loss曲线我们可以得到什么结论呢？



有时候我们要监听的Loss需要更加细化，比如监听训练Loss时要监听回归的loss，分类的loss等等，这个时候我们想要单独查看某一个窗口中loss变化的情况时就可以用到我们在上面提高的Filter来对窗口展示进行过滤，看下图我们只想要展示Train loss的情况：



## 热力图可视化

有时候在进行实验分析时，我们需要可视化一些热力图或者特征图来对实验效果进行分析，Visdom也提供了非常方便的接口对Tensor数据进行可视化。

```

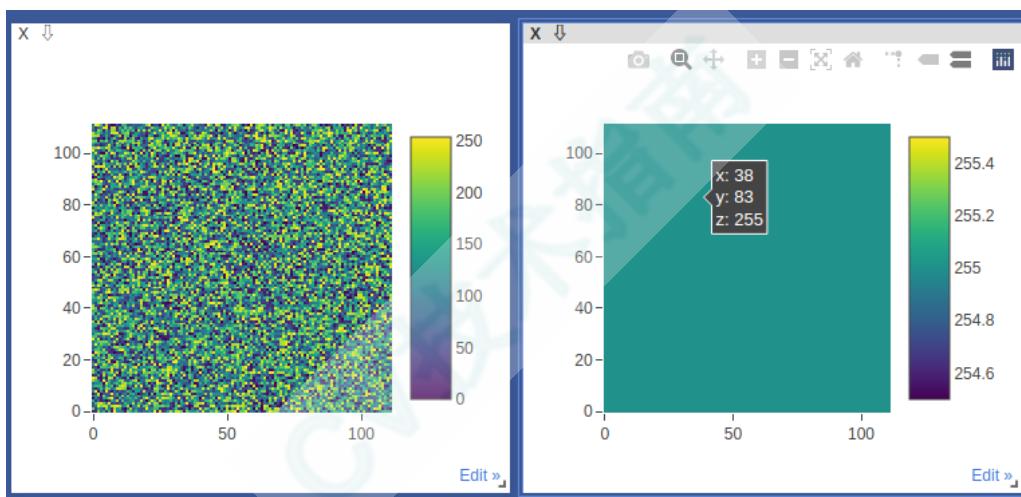
import visdom
import numpy as np
import torch

vis_main = visdom.Visdom()
# heatmap
tensor = torch.randint(0, 255, (112,112))
# heatmap的输入必须是一个二维的Tensor
vis_main.heatmap(tensor, win='three')

tensor_two = torch.ones((112,112)) * 255
vis_main.heatmap(tensor_two, win='four')

```

如下图所示左边是一个随机可视化的二维Tensor，右边是一个值全为255的Tensor。我们想要可视化一个Shape为(C,H,W)的Feature map时只需要对C维进行压缩，然后就可以方便的使用heatmap这个接口来进行可视化了~



## API接口

除了上述介绍的，Visdom还提供了其他各种各样的接口满足大家不同的可视化需求，这里我们不再进行详细介绍，感兴趣的同學可以参考Visdom官方Github。

```

# Basic 接口
vis.image : image 展示图像
vis.images : list of images
vis.text : arbitrary HTML
vis.properties : properties grid
vis.audio : audio
vis.video : videos
vis.svg : SVG object
vis.matplot : matplotlib plot
vis.save : serialize state server-side

# Plotting相关的

```

```
vis.scatter : 2D or 3D scatter plots  
vis.line : line plots  
vis.stem : stem plots  
vis.heatmap : heatmap plots 热力图  
vis.bar : bar graphs  
vis.histogram : histograms 直方图  
vis.boxplot : boxplots  
vis.surf : surface plots  
vis.contour : contour plots  
vis.quiver : quiver plots  
vis.mesh : mesh plots  
vis.dual_axis_lines : double y axis line plots
```

## 写在后面

本文介绍了Visdom的重要基本概念以及常用的可视化接口，对于很多人来说这些常用接口可能就非常够用了，但是Visdom的强大远不止于此。基本上能想到的图形图像接口Visdom都有提供，一些高大上的**论文插图**也可以尝试用Visdom来构建，也可以通过**布局排版**功能做出精美的可视化界面，还有许许多多的功能等待你我进一步探索~

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## 数据增强可视化

对图片进行数据增强，可以有效扩充数据集，减少模型过拟合，常用的数据增强方法如下。

1. 对图片进行一定比例缩放
2. 对图片进行随机位置的截取
3. 对图片进行随机的水平和竖直翻转
4. 对图片进行随机角度的旋转
5. 对图片进行亮度、对比度和颜色的随机变化

这里我们使用PIL库和Pytorch中的torchvision库完成下面的例子，首先确保你已经安装了这些库

```
import torch  
torch.__version__
```

```
'1.10.0'
```

```
import sys
from PIL import Image
from torchvision import transforms as tfs

#打开一张图片
img = Image.open('cat.jpeg')
img
```



## 1.对图片进行一定比例缩放

这里使用`torchvision.transforms.Resize(size, interpolation=2)`对图片进行缩放，第一个参数是缩放的长宽大小，第二个参数是缩放的插值函数，默认为双线性插值方法，详细可参考[文档](#)

```
print('origin image shape: {}'.format(img.size))
new_img = tfs.Resize((100, 200))(img)
print('after scale image shape: {}'.format(new_img.size))
new_img
```

```
origin image shape: (800, 533)
after scale image shape: (200, 100)
```



## 2. 对图片进行随机位置的截取

使用随机位置截取出图片中局部的信息，使得网络结构的输入具有多尺度的特征。  
`torchvision.transforms.CenterCrop(size)`，对图片进行中心截取，size可以是tuple,也可以是Integer，这时切出来是正方形

```
# 中心裁剪出 200 x 200 的区域  
center_img = tfs.CenterCrop(200)(img)  
center_img
```



`torchvision.transforms.RandomCrop(size, padding=0)`，对图片进行随机截取，size可以是tuple,也可以是Integer

```
random_img = tfs.RandomCrop((200, 100))(img)  
random_img
```



`torchvision.transforms.RandomResizedCrop(size, interpolation=2)`, 对图片进行随机截取, 再resize成给定的size大小

```
random_resize_img = tfs.RandomResizedCrop((200, 200))(img)  
random_resize_img
```



### 3.对图片进行随机的水平和竖直翻转

`torchvision.transforms.RandomHorizontalFlip(p=0.5)` 输入参数p为随机水平翻转的概率值, 默认值为0.5

```
# 随机水平翻转  
h_filp = tfs.RandomHorizontalFlip(p=1)(img)  
h_filp
```



`torchvision.transforms.RandomVerticalFlip(p=0.5)` 输入参数p为垂直翻转的概率值,默认值为0.5

```
# 随机竖直翻转  
v_flip = tfs.RandomVerticalFlip()(img)  
v_flip
```



## 4.对图片进行随机角度的旋转

`torchvision.transforms.RandomRotation(degrees[, interpolation,...])` 来实现，其中第一个参数就是随机旋转的角度，比如填入 45，那么每次图片就会在 -45 ~ 45 度之间随机旋转

```
rot_img = tfs.RandomRotation(45)(img)  
rot_img
```



## 5. 对图片进行亮度、对比度和颜色的随机变化

```
torchvision.transforms.ColorJitter(brightness=0, contrast=0, saturation=0, hue=0)
```

brightness (float或 float类型元组(min, max)) – 亮度的偏移幅度。

brightness\_factor从  $[\max(0, 1 - \text{brightness}), 1 + \text{brightness}]$  中随机采样产生。应当是非负数

brightness 设置为 0.5 表示将图像的亮度随机变化为原图像亮度的 50% ( $1-0.5$ ) ~ 150% ( $1+0.5$ )

即  $[\max(0, 1-\text{brightness}), 1 + \text{brightness}]$  就是  $[0.5, 1.5]$

contrast 和 saturation 类似

hue (float或 float类型元组(min, max)) – 色相偏移幅度。

hue\_factor 从  $[-\text{hue}, \text{hue}]$  中随机采样产生，其值应当满足  $0 \leq \text{hue} \leq 0.5$  或  $-0.5 \leq \text{min} \leq \text{max} \leq 0.5$

```
# 亮度
bright_img = tfs.ColorJitter(brightness=0.5)(img) # 随机从 0.5 ~ 1.5 之间亮度变化
bright_img
```

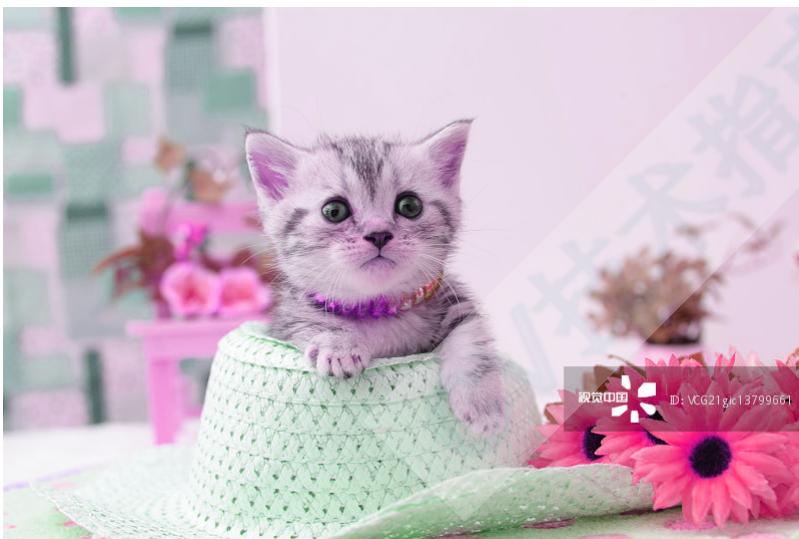


```
# 对比度
contrast_img = tfs.ColorJitter(contrast=0.5)(img) # 随机从 0.5 ~ 1.5 之间亮度变化
contrast_img
```



```
# 颜色
```

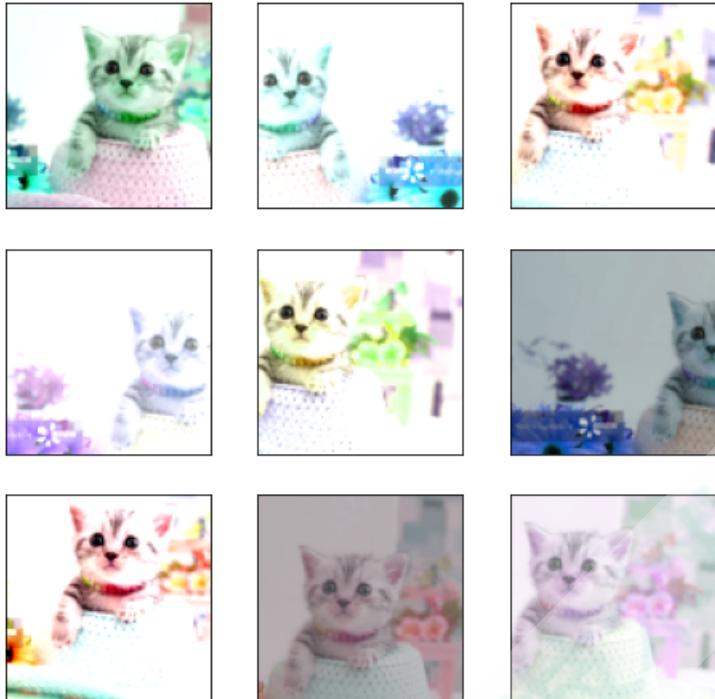
```
color_img = tfs.ColorJitter(hue=0.5)(img) # 随机从 -0.5 ~ 0.5 之间对颜色变化  
color_img
```



## 6.组合使用

```
img_aug = tfs.Compose([  
    tfs.Resize(120),  
    tfs.RandomHorizontalFlip(),  
    tfs.RandomCrop(96),  
    tfs.ColorJitter(brightness=0.5, contrast=0.5, hue=0.5)  
])  
import matplotlib.pyplot as plt  
%matplotlib inline  
nrows = 3  
ncols = 3
```

```
figsize = (8, 8)
_, figs = plt.subplots(nrows, ncols, figsize=figsize)
for i in range(nrows):
    for j in range(ncols):
        figs[i][j].imshow(img_aug(img))
        figs[i][j].axes.get_xaxis().set_visible(False)
        figs[i][j].axes.get_yaxis().set_visible(False)
plt.show()
```



本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## 目标检测数据集格式转换

### 前言

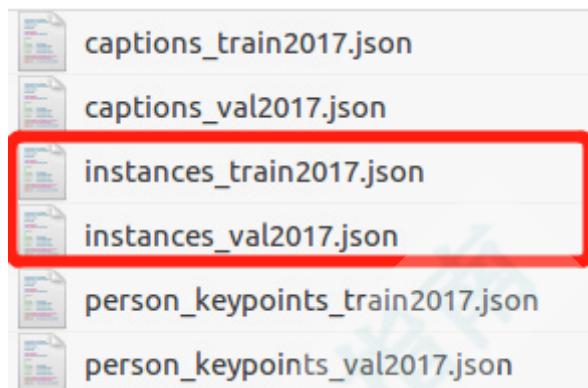
图像目标检测领域有一个非常著名的数据集叫做COCO，基本上现在在目标检测领域发论文，COCO是不可能绕过的Benchmark。因此许多的开源目标检测算法框架都会支持解析COCO数据集格式。通过将其他数据集格式转换成COCO格式可以无痛的使用这些开源框架来训练新的数据集，比如Pascal VOC数据集。

本文首先将介绍COCO和VOC目标检测数据集格式以及VOC转换到COCO格式的核心步骤，最后将自定义一种数据格式利用上述核心步骤将其转换到COCO格式下。只要理解了不同数据集的标注方法，转换数据集其实就是一个非常简单自然的过程，可以拓展到任意方式标注的数据集上。

## 数据集格式介绍

### COCO

其实COCO数据集的标签内容不仅仅涵盖目标检测，还包含了**目标关键点、实例Mask**以及**图片描述**等信息。在这里我们着重介绍COCO的目标检测相关内容。我们以COCO2017为例先看看其标签文件结构：



其中红框框出来的就是以**Json格式组织**的目标检测相关的**标注文件**，其主要由三个部分构成：

1. "info"字段：数据集的**基本信息**描述、版本号、年份等信息。
2. "images"字段：包含了**图片路径**、**宽高信息**、**唯一标志ID**等信息。
3. "annotations"字段：包含了图片中的**Box位置**、**类别**等信息。

其简单示例如下所示：

```
{  
    "info": {  
        "description": "COCO 2017 Dataset",  
        "url": "http://cocodataset.org",  
        "version": "1.0",  
        "year": 2017,  
        "contributor": "COCO Consortium",  
        "date_created": "2017/09/01"  
    },  
    "images": [  
        {  
            "license": 4,  
            "file_name": "000000397133.jpg",  
            "coco_url":  
                "http://images.cocodataset.org/val2017/000000397133.jpg",  
            "height": 425,  
            "width": 600  
        }  
    ]  
}
```

```

    "height": 427,
    "width": 640,
    "date_captured": "2013-11-14 17:02:52",
    "flickr_url": "",
    "id": 397133
},
{
    "license": 1,
    "file_name": "000000037777.jpg",
    "coco_url":
"http://images.cocodataset.org/val2017/000000037777.jpg",
    "height": 230,
    "width": 352,
    "date_captured": "2013-11-14 20:55:31",
    "flickr_url": "",
    "id": 37777
}
],
"annotations": [
{
    "area": 702.1057499999998, //Box的尺寸
    "image_id": 289343, //对应的图像ID
    "bbox": [
        473.07, //左上角点x坐标
        395.93, //左上角点y坐标
        38.65, //Box的宽
        28.67 //Box的高
    ],
    "category_id": 18, //对应的类别
    "id": 1768, //该标签独有ID
    "iscrowd": 0 //0表示非密集场景，1表示密集场景
}
]
}

```

## PASCAL VOC

PASCAL VOC数据集有两个相对重要年份的数据集：PASCAL VOC 2007与PASCAL VOC 2012,每年都会在上一年的基础上增加一些额外的数据或标签。PASCAL VOC数据集也涵盖了分类、检测、分割、动作识别等标签。我们这里着重介绍其检测部分，以PASCAL VOC 2012数据集为例，包含了**20个类别****1W+数据集**，**2W+标注Box**的目标。其标签格式是每一个图片都有一个对应的XML文件作为其标注信息载体，标注信息主要包含如下几方面内容：

1. 图像基本信息：图像名、图像尺寸等
2. **object**字段：目标分类标签、box标签(xmin,ymin,xmax,ymax)等信息

XML主要格式如下：

```
<annotation>
  <folder>VOC2012</folder>
  <filename>2007_000063.jpg</filename>          //标签对应的图片文件
  <source>
    <database>The VOC2007 Database</database>
    <annotation>PASCAL VOC2007</annotation>
    <image>flickr</image>
  </source>
  <size>                                         //图像尺寸
    <width>500</width>
    <height>375</height>
    <depth>3</depth>
  </size>
  <segmented>1</segmented>
  <object>
    <name>dog</name>                         //类别
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>123</xmin>                      //左上角x坐标
      <ymin>115</ymin>                      //左上角y坐标
      <xmax>379</xmax>                      //右下角x坐标
      <ymax>275</ymax>                      //右下角y坐标
    </bndbox>
  </object>
  <object>
    <name>chair</name>
    <pose>Frontal</pose>
    <truncated>1</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>75</xmin>
      <ymin>1</ymin>
      <xmax>428</xmax>
      <ymax>375</ymax>
    </bndbox>
  </object>
</annotation>
```

# 数据集格式转换

在知道了各个数据集格式的基础上做数据集格式的转换就已经是非常简单的任务了，也有很多优秀的开源框架已经帮我们做好了这些事情比如MMDetection中就已经提供好了现成的工具供我们白嫖（bushi），使用了。我们抽取其一些核心部分来一起看看，详细代码请参考 **MMDetection Github:**[https://github.com/open-mmlab/mmdetection/tree/master/tools/dataset\\_converters](https://github.com/open-mmlab/mmdetection/tree/master/tools/dataset_converters)

从不同的数据集转换到COCO下主要也就两个步骤：

1. 解析待转换数据集格式。
2. 用COCO格式重构Json文件。

上述第二步对任意待转换数据集都是一样的，可以抽象为一个函数，输入的是解析好的不同数据集的Box信息等数据。下面我们将以几个不同的数据集为例介绍。

## From VOC to COCO

从VOC数据集到COCO数据集格式转换主要包含如下两个步骤：

1. 解析VOC数据集数据：遍历图片以及对应XML文件，返回一个数组A，数组中的每一个实例包含了图片路径、Box相关标注信息等。
2. 遍历A中的实例信息用COCO的格式表达出来并生成Json文件

其主要由两块核心代码构成，一个是VOC的XML文件解析，一个是Json文件生成。

```
# VOC XML标注文件解析
# XML文件解析已经有下面这个非常方便的Python库供大家使用
import xml.etree.ElementTree as ET

def parse_xml(args):
    xml_path, img_path = args
    tree = ET.parse(xml_path)          # 构建XML文件解析树
    root = tree.getroot()              # 获取XML文件的根节点
    size = root.find('size')           # 获取图像的尺寸
    w = int(size.find('width').text)    # 图像宽高
    h = int(size.find('height').text)
    bboxes = []
    labels = []
    bboxes_ignore = []
    labels_ignore = []
    for obj in root.findall('object'):  # 遍历object字段下所有box信息
        name = obj.find('name').text
        label = label_ids[name]
        difficult = int(obj.find('difficult').text) #这个difficult对应的是
                                                    #COCO中iscrowded
        bnd_box = obj.find('bndbox')
```

```

bbox = [
    int(bnd_box.find('xmin').text),
    int(bnd_box.find('ymin').text),
    int(bnd_box.find('xmax').text),
    int(bnd_box.find('ymax').text)
]
if difficult: # 将difficult属性的Box放入ignore列
表
    bboxes_ignore.append(bbox) # 最后计算AP时这个GT是被忽略的
    labels_ignore.append(label)
else:
    bboxes.append(bbox)
    labels.append(label)
if not bboxes:
    bboxes = np.zeros((0, 4))
    labels = np.zeros((0, ))
else:
    bboxes = np.array(bboxes, ndmin=2) - 1
    labels = np.array(labels)
if not bboxes_ignore:
    bboxes_ignore = np.zeros((0, 4))
    labels_ignore = np.zeros((0, ))
else:
    bboxes_ignore = np.array(bboxes_ignore, ndmin=2) - 1
    labels_ignore = np.array(labels_ignore)
annotation = {
    'filename': img_path,
    'width': w,
    'height': h,
    'ann': {
        'bboxes': bboxes.astype(np.float32),
        'labels': labels.astype(np.int64),
        'bboxes_ignore': bboxes_ignore.astype(np.float32),
        'labels_ignore': labels_ignore.astype(np.int64)
    }
}
return annotation

```

用解析好的annotation重构COCO格式的Json文件：

```

import numpy as np

def voc_classes():
    return [
        'aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car',
'cat',
        'chair', 'cow', 'diningtable', 'dog', 'horse', 'motorbike', 'person',
    ]

```

```
'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor'
]

def cvt_to_coco_json(annotations):
    image_id = 0
    annotation_id = 0
    coco = dict()
    coco['images'] = []
    coco['type'] = 'instance'
    coco['categories'] = []
    coco['annotations'] = []
    image_set = set()

    # coco annotations字段添加标注信息
    def addAnnItem(annotation_id, image_id, category_id, bbox,
difficult_flag):
        annotation_item = dict()
        annotation_item['segmentation'] = []

        # 这里省略了seg部分代码
        seg = []
        annotation_item['segmentation'].append(seg)

        # 转换为COCO对应的x1,y1,w,h格式
        xywh = np.array(
            [bbox[0], bbox[1], bbox[2] - bbox[0], bbox[3] - bbox[1]])
        annotation_item['area'] = int(xywh[2] * xywh[3])
        # 如果difficult标志为1，该目标对应coco中iscrowd字段为1表明为密集目标场景
        if difficult_flag == 1:
            annotation_item['ignore'] = 0
            annotation_item['iscrowd'] = 1
        else:
            annotation_item['ignore'] = 0
            annotation_item['iscrowd'] = 0
        annotation_item['image_id'] = int(image_id)
        annotation_item['bbox'] = xywh.astype(int).tolist()
        annotation_item['category_id'] = int(category_id)
        annotation_item['id'] = int(annotation_id)
        coco['annotations'].append(annotation_item)
        return annotation_id + 1

    #
    for category_id, name in enumerate(voc_classes()):
        category_item = dict()
        category_item['supercategory'] = str('none')
        category_item['id'] = int(category_id)
        category_item['name'] = str(name)
```

```

coco['categories'].append(category_item)

for ann_dict in annotations:
    file_name = ann_dict['filename']
    ann = ann_dict['ann']
    assert file_name not in image_set
    image_item = dict()
    image_item['id'] = int(image_id)
    image_item['file_name'] = str(file_name)
    image_item['height'] = int(ann_dict['height'])
    image_item['width'] = int(ann_dict['width'])
    coco['images'].append(image_item)          # 设置COCO的"images"字段
    image_set.add(file_name)

    # 设置COCO的"annotations"字段
    bboxes = ann['bboxes'][:, :4]           # 获取box和label类别信息
    labels = ann['labels']
    for bbox_id in range(len(bboxes)):
        bbox = bboxes[bbox_id]
        label = labels[bbox_id]
        annotation_id = addAnnItem(
            annotation_id, image_id, label, bbox, difficult_flag=0)

    # ignore的目标表示该GT被忽视
    bboxes_ignore = ann['bboxes_ignore'][:, :4]
    labels_ignore = ann['labels_ignore']
    for bbox_id in range(len(bboxes_ignore)):
        bbox = bboxes_ignore[bbox_id]
        label = labels_ignore[bbox_id]
        annotation_id = addAnnItem(
            annotation_id, image_id, label, bbox, difficult_flag=1)

    image_id += 1

return coco

```

拿到返回的coco对象后只需要调用下列方法就可以将对象序列化成Json文件了。

```

import mmcv
mmcv.dump(coco, out_file) # out_file为输出的json文件名

```

值得注意的一点是上面提到的iscorwd这个字段，这个字段标注为1时，最后统计AP时，该GT与预测框完成匹配后还可以考虑与其他预测框进行匹配，允许多个预测框与其匹配(因为场景是密集的)。

## 自定义格式数据集 to COCO

首先我们自定义一种数据标注格式，我们用txt文件作为标注信息的载体，将txt文件与图像文件通过相同的文件名一一对应。分别将标签文件以及对应图像文件放在Annotations以及JPEGImages文件夹下，同时我们生成JPEGImages图像文件的filelist.txt文件，这个文件每一行对应一个图像文件的全路径：



txt文件格式如下：

```
0 647.0 71.0 21.0 47.0  
0 140.0 384.0 30.0 49.0  
0 229.0 240.0 26.0 69.0  
0 134.0 323.0 21.0 49.0  
0 163.0 305.0 21.0 64.0  
0 1200.0 267.0 19.0 23.0
```

第一列表示类别，从0开始；第二到第五列表示Box信息依次为中心点x方向坐标，中心点y方向坐标，box的宽以及高(cx, cy, w, h)。

我们同样使用前面介绍过的cvt\_to\_coco\_json将固定格式的annotations转换为COCO格式，那么我们只需要编写解析自定义格式数据集生成annotations的代码即可：

```
# box尺寸小于min_size的作为ignore对象  
# file_path为图像路径的filelist.txt文件的全路径  
def parse_info(file_path, min_size):  
    annotations = []  
    invalid_img = 0  
    small_box = 0  
    with open(file_path, 'r') as f:  
        for l in tqdm(f):  
            img_file = l.rstrip()  
            img = cv2.imread(img_file)  
            if img is None:  
                invalid_img += 1  
                continue  
            h,w,_ = img.shape  
            # 获取对应的标签文件  
            ann_file = img_file.replace("JPEGImages", "Annotations").replace  
(\n                ".png", ".txt").replace(".jpg", ".txt")  
            annotation = {'filename' : img_file,  
                          'height' : h,  
                          'width' : w,  
                          'ann' : []}
```

```

        if not osp.exists(ann_file):
            annotations.append(annotation)
            continue
        boxes, labels = [], []
        boxes_ignore, labels_ignore = [], []
        with open(ann_file, 'r') as fr:
            for anno in fr:
                anno_list = anno.rstrip().split(' ')
                cls = int(anno_list[0])
                cx, cy = float(anno_list[1]), float(anno_list[2])
                w, h = float(anno_list[3]), float(anno_list[4])
                # 转换为COCO box表示格式
                x1 = max(0, int(cx-w/2))
                y1 = max(0, int(cy-h/2))
                box = [x1, y1, w, h]
                if w >= min_size and h >= min_size:
                    labels.append(cls)
                    boxes.append(box)
                else:
                    labels_ignore.append(cls)
                    boxes_ignore.append(box)
        boxes = np.zeros((0, 4)) if len(boxes) == 0 else
np.array(boxes)
        labels = np.zeros((0, )) if len(labels) == 0 else
np.array(labels)
        boxes_ignore = np.zeros((0, 4)) if len(boxes_ignore) == 0
else np.array(boxes_ignore)
        labels_ignore = np.zeros((0, )) if len(labels_ignore) == 0
else np.array(labels_ignore)
        annotation['ann']['bbox'] = np.array(boxes)
        annotation['ann']['label'] = np.array(labels)
        annotation['ann']['bbox_ignore'] = np.array(boxes_ignore)
        annotation['ann']['label_ignore'] = np.array(labels_ignore)

    annotations.append(annotation)
    print('INFO: Invalid IMG:{}.'.format(invalid_img))
return annotations

```

## 写在后面

数据集的转换是非常有必要的，在软件设计中我们希望一套代码尽可能多的为不同情况服务。在这里我们希望训练代码中一套数据集（Dataset）class代码来完成所有目标检测任务训练，而不是针对不同的数据集设计不同的Dataset class代码。而对于目标检测来说，COCO可能就是这个最佳的模板~

欢迎加入QQ交流群：444129970。本文档如若任何错误，请加QQ群联系管理员修改

## (十一) 自定义损失函数、添加或设计模块

本节我们介绍如何自定义损失函数，如何在现有的网络上添加或设计一些模块d。

关于如何添加自定义的损失函数，我们已经在搭建网络那一节已经介绍过了，很多朋友仍然不是很清楚。因此这里就继续再详细地写一下，并举一个例子作为说明。

### 自定义损失函数

损失函数中没有可训练的参数，因此损失函数主要是使用torch.nn.functional这个库来实现。

以TripletLoss为例。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class TripletLoss(nn.Module):
    def __init__(self, t1, t2, beta):
        super(TripletLoss, self).__init__()
        self.t1 = t1
        self.t2 = t2
        self.beta = beta
        return

    def forward(self, anchor, positive, negative):
        matched = torch.pow(F.pairwise_distance(anchor, positive), 2)
        mismatched = torch.pow(F.pairwise_distance(anchor, negative), 2)
        part_1 = torch.clamp(matched - mismatched, min=self.t1)
        part_2 = torch.clamp(matched, min=self.t2)
        dist_hinge = part_1 + self.beta * part_2
        loss = torch.mean(dist_hinge)
        return loss
```

简单介绍一下用法，跟定义网络一样，继承nn.Module，然后完成init和forward函数，init函数中设置一些权重，forward编写计算过程。

可能大家所不会写的是计算的这个过程，这个很难讲，了解torch的一些基本计算函数就好了，例如上面用到了指数，截断，求均值，距离函数等。当你有了损失函数的公式，实现这个计算的代码是一件很简单的事。

## 添加或设计模块

添加或设计模块主要需要注重两个东西，一个是输入输出是什么，一个是每个维度代表什么。

前者是可以添加这个模块的实现基础，后者是这个模块起作用的理论基础。

首先是明确输入输出，在CNN网络中，一般就是  $(B, C, H, W)$  的形式。因此想要在一个基本的backbone上添加一个即插即用的模块，模块的输入输出都是这种形式，这个很好理解，不多解释。

在Transformer中，则需要注意以下它的形式，通常有两种方式生成tokens，有些是通过卷积分块得到的，有些是直接分割得到的，在swin transformer中，它是通过卷积生成的  $(B, C, L)$  的形式，然后做了一个transpose得到  $(B, L, C)$ ，这里的L即相当于通道，C相当于特征。在ViT中使用einop库直接分割得到tokens的方式，则生成的是  $(B, N, D)$  它不需要做transpose，N相当于通道，D相当于特征。

由于transformer网络中间经常做transpose，很容易就把哪个维度是特征哪个维度是通道给搞混了，对于这点来说，个人建议是对于在想要添加模块的位置，先打印形状，根据形状明确每个维度代表的含义。

当明确输入输出后，就只剩下了把模块的计算给实现。

实现这一块还是比较简单的，主要也是了解几个基本的操作。

改变形状：reshape, transpose, permute, view, einops库, flatten等。

通道拼接：cat,.

添加网络层：nn.Conv2d, nn.BN, nn.ReLU, nn.Sigmoid, nn.Linear等。

基本了解这些，再明确每个操作是针对哪个维度的，就很简单了。

以SE为例。

```
class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel // reduction, bias=False),
```

```
        nn.ReLU(inplace=True),
        nn.Linear(channel // reduction, channel, bias=False),
        nn.Sigmoid()
    )

    def forward(self, x):
        b, c, _, _ = x.size()
        y = self.avg_pool(x).view(b, c)
        y = self.fc(y).view(b, c, 1, 1)
        return x * y.expand_as(x)
```

如上所示，先明确x的shape，即B, C, H, W。SE的主要操作是对每个feature map去全局均值，然后再经过两层全连接层，获得每个通道的加权值，最后再与输入x进行加权。这个过程首先是做池化，得到的是四维的张量，然后利用view降维，再全连接，最后又扩展到四维，与x逐元素相乘。

这正符合了上面提到的，先明确每个维度代表什么，再把要做的操作做好，如果操作的维度有变，则利用那几个改变形状的操作调整。

这个其实很难说，因为很简单，若是这样还不太清楚，请自行找几个即插即用的模块，对照着上面提到的几个要点看看。

## (十二)完整pipeline项目

Github: <https://github.com/wxkang157/PytorchPipeline>

本文档来自公众号与知识星球【CV技术指南】联合出品。欢迎分享给个人学习，严禁用于商业行为。

扫码关注公众号CV技术指南，专注于计算机视觉的技术总结、论文解读、招聘信息发布等。



扫码加入CV技术指南知识星球。

CV技术指南旨在打造一个完善的计算机视觉知识体系，为入门的人提供服务，为工作的人提供平台。自星球成立以来，星球内有为星友们提供了许多计算机视觉方面的资料，有理论知识，有部署经验，也有论文解读。其中包括目标检测中Head、Neck的设计系列、YOLO系列、anchor-free系列、小目标检测系列、目标检测中的Label Assignment系列、传统目标检测系列、视频中的目标检测系列、图像分割系列、部署系列、CUDA教程系列、论文解读系列、Pytorch源码解读系列、Pytorch实践教程系列。后续还会继续补充~...



欢迎加入QQ群：444129970。QQ群专注于计算机视觉的算法、技术、学习、工作、求职等方面的交流，群内交流氛围极好，有专门的大佬维护，基本99%的提问都会解答。群文件内有很多电子版资源，可供大家免费下载，也欢迎其他人一起上传新资料。



对于初学者，如果需要进行入门辅导，可以看一下下面的海报。

# 计算机视觉入门 1v1辅导班

CV技术指南

## 课程内容

深度学习、机器学习基础、数字图像处理  
pytorch、目标检测、英文文献阅读  
计算机视觉进阶、日后的技术或科研发展路线

## 课程目标

1. 掌握计算机视觉的知识体系
2. 具备搭建模型、分析模型的能力
3. 具备直接阅读英语论文的能力
4. 形成对计算机视觉的理解能力
5. 掌握计算机视觉的正确学习方法
6. 具备独立做科研或项目的能力

## 课程优势

1. 1v1辅导，任何疑问都可以解答到弄懂为止。
2. 由浅及深、循序渐进的学习路线。
3. 全面的计算机视觉知识体系。
4. 阶段考核、进度督促，保证学习效率效果。
5. 注重传授学习方法、而非内容。
6. 根据个人时间灵活制定学习计划与辅导时长
7. 免费加入知识星球一年，享受所有技术教程

## 咨询与报名

扫描右方二维码，备注“入门辅导”，了解详细情况

