目标检测从理解到实战

1. 机器学习、神经网络、深度学习

什么是机器学习

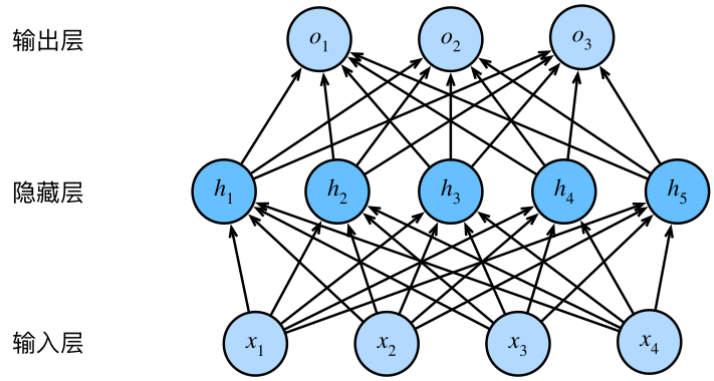
机器学习是一门学科，它致力于研究如何通过计算的手段，利用经验来改善系统自身的性能。在计算机系统中，“经验”通常以“数据”形式存在，因此机器学习所研究的主要内容，是关于在计算机上从数据中产生“模型”的算法。在面对新的数据（例如看到一个没有剖开的西瓜）时，模型会给我们提供相应的判断（是否是好瓜）。

机器学习与神经网络

机器学习包括多种算法，如**支持向量机**、**决策树**、**KNN**等，而**神经网络**只是机器学习算法中的一种。

神经网络与深度学习

神经网络是一种机器学习算法，是由多个“神经元”相互连接形成的网状计算模型，而深度学习模型就是指很深层的神经网络，通常使用误差逆传播（BP）算法进行训练。



2. 函数拟合与神经网络

世界的本质是函数，神经网络做的就是函数拟合。把神经网络当做一个函数$f()$当我们输入一个数据x时，神经网络会预测一个结果y，$\vec{y}= f(\vec{x})$

- 如果我们要对图片进行分类，那x就是图片y就是图片的类别
- 如果我们要判断一个病人是否患了癌症，那么x就是病人的各种信息，y就是病人是否患病
- 如果我们要进行中-英翻译，那么x就是中文，y就是翻译出来的英文
- 如果你的神经网络是GPT，那么x就是你之前跟GPT的对话，y就是GPT的回复

那么如何找到这个$f()$？

答： 每个神经网络都可以看成有n个参数的模型，只要找到正确的参数，就能实现想要的效果。利用大量的标注好的数据$(x_0, y_0), (x_1, y_1), (x_2, y_2)\ldots$，通过最小化损失函数的方式调整这些参数，直到达到满意的效果。而这个最小化损失函数通常是使用梯度下降算法实现的。

3. 一些术语

- 前向传播(forward)：就是指计算$f(x)$，通过输入计算输出
- 反向传播(back propogate)：指通过输出，利用求导的链式法则，从输出层逐层往输入层进行求导，计算各个层的梯度
- 梯度：偏导数构成的向量
- 损失函数(loss function)：要最小化的目标函数，根据任务不同有不同的设计，常见的损失函数包括MSE、CrossEntropyLoss、L1、NLLLoss等
- 数据集(dataset)：一组数据对，包括数据和标签
- 张量(Tensor)：N x N x N x...的数据块，一维数据块叫向量，二维叫矩阵，三维及以上叫张量
- epoch：神经网络的训练过程通常需要多次迭代，一个epoch表示完成一轮训练
- batch：训练过程中，通常会采用批量梯度下降的方式，即一次给GPU输入多张图片计算LOSS，batch大小即一次输入到GPU中的图片数量。

  > 在部署阶段，batch=1

- mAP：即mean Average Precision

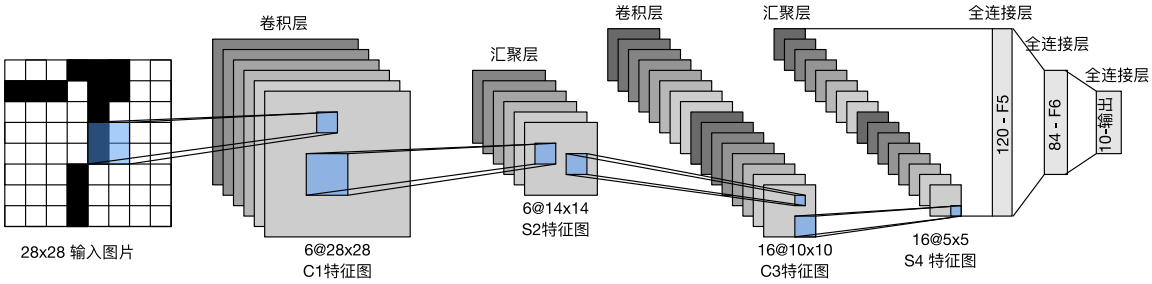
4. Backbones：从LeNet到MobileNet

LeNet-5 最早的CNN

在计算机视觉领域，最常用的神经网络模型是卷积神经网络（Convolutional Neural Network）,其核心是使用卷积层代替全连接层进行特征的提取。

> CNN最近有被Vision Transformer超越的趋势

LeNet-5是深度学习之父，Yann LeCun于1998年提出的最早的卷积神经网络，当时是用来解决手写体数字识别问题，其网络结构图如下所示，从LeNet-5开始，CNN就形成了若干个（卷积、池化）层+（用于特定任务的全连接层）的范式。



下面是LeNet-5对应的PyTorch代码

```
import torch
import torch.nn as nn
```

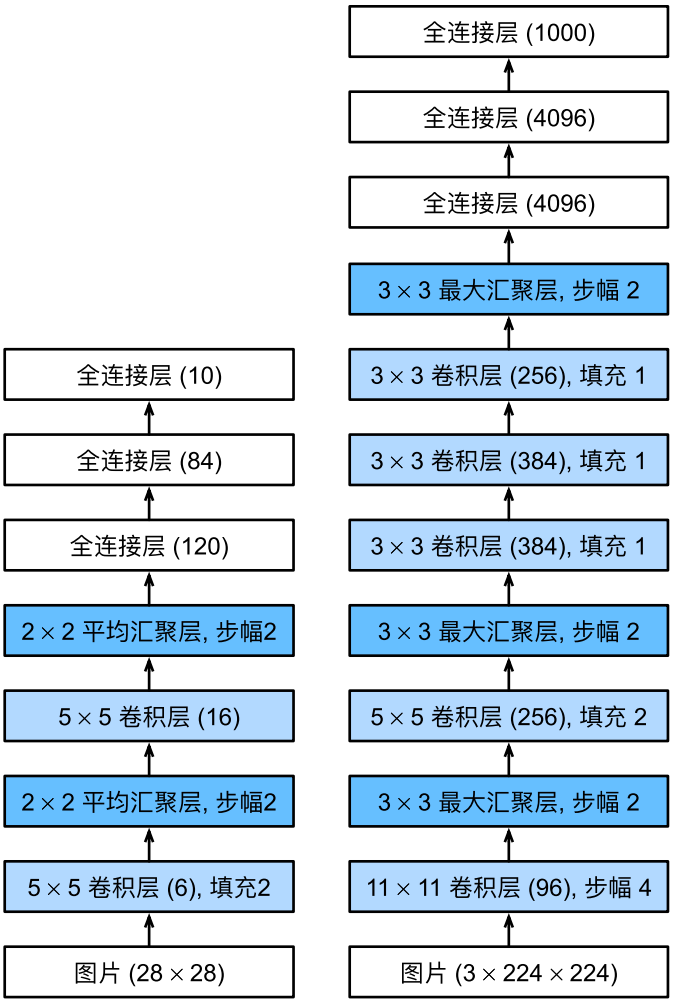
```
class Lenet(nn.Module):
  def __init__(self, num_classes=10):
    super(Lenet, self).__init__()
    #卷积层1
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1,
padding=2)
    #池化(汇聚)层1
    self.pool1 = nn.MaxPool2d(kernel_size=2)
    #卷积层2
    self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1)
    #池化(汇聚)层2
    self.pool2 = nn.MaxPool2d(kernel_size=2)
    #展开
    self.flatten = nn.Flatten()
    #三个全连接层
    self.fc1 = nn.Linear(in_features=16*5*5, out_features=120)
    self.fc2 = nn.Linear(in_features=120, out_features=84)
    self.fc3 = nn.Linear(in_features=84, out_features=num_classes)

    self.act = nn.ReLU(inplace=True)
    self.softmax = nn.Softmax(dim=1)

  def forward(self, x):
    y = self.act(self.conv1(x))
    y = self.pool1(y)
    y = self.act(self.conv2(y))
    y = self.pool2(y)
    y = self.flatten(y)
    y = self.act(self.fc1(y))
    y = self.act(self.fc2(y))
    y = self.fc3(y)
    y = self.softmax(y)
    return y
```

AlexNet 改变世界的神经网络

在LeNet-5的年代，深度学习并没有受到机器学习研究者的重视，理由是深度学习需要大量的计算量，参数多，需要大量的数据集供于训练。2012年，Alex Krizhevsky，Ilya Sutskever和Yoshua Bengio提出了AlexNet，并在同年的ImageNet图像分类竞赛上以压倒性的优势夺得了冠军。AlexNet获得成功的关键是

- ImageNet2012提供了大量的已标注数据集，使得大规模的网络有足够的数据进行学习
- 利用CUDA技术，Alex Krizhevsky和Ilya Sutskever实现了可以在GPU硬件上运行的深度卷积神经网络，而神经网络这种基于矩阵运算的模型能够在GPU上快速运算，突破了深度网络训练的瓶颈。

下图为LeNet-5（左）和AlexNet（右）的对比，AlexNet任然保留了N个卷积池化+全连接层的形式

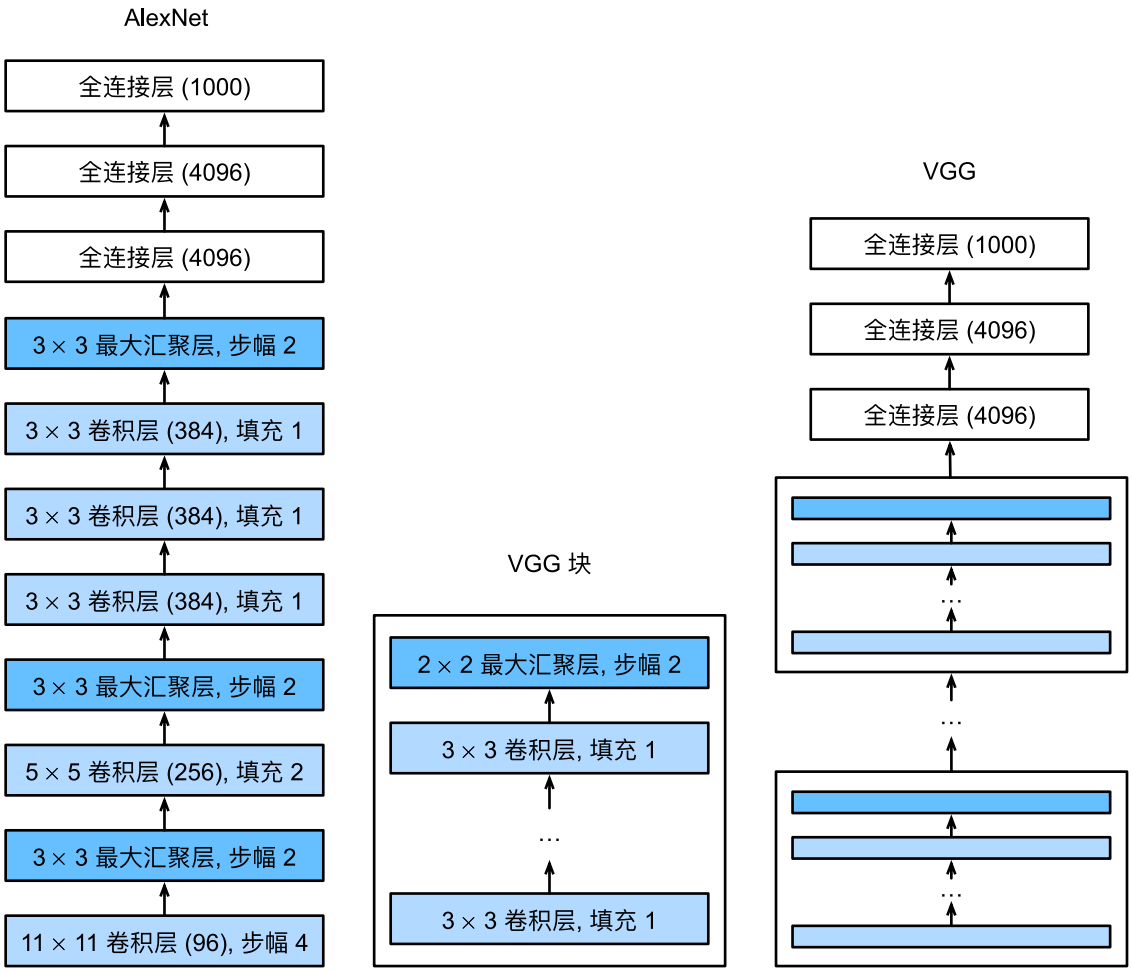


VGG 模块化设计深度卷积网络

就如LeNet和AlexNet，卷积神经网络可以简化为的形式

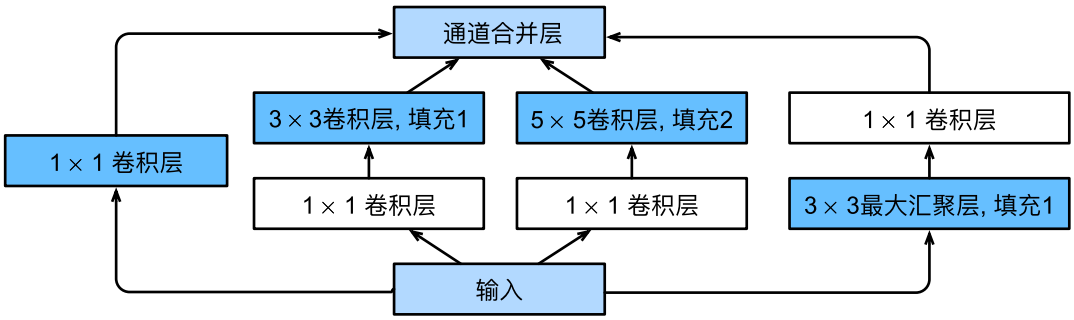
神经网络 $= N * \ (n * Conv + MaxPool) +$ 检测头

VGG网络就是这样，他们将$(n * Conv + MaxPool)$作为一个基本模块（VGG块），通过不同模块的拼接，设计了VGG-16，VGG-19等不同的网络，都取得了不错的效果。

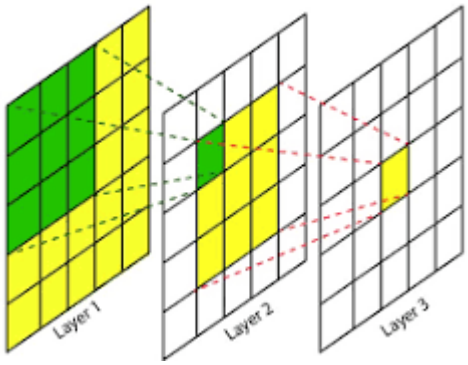


GoogLeNet 不同感受野的融合

GoogLeNet中提出了Inception块，他将不同大小的卷积的结果进行拼接，有效融合了不同感受野的信息。



下图是感受野的示意图，可以看见，第三层一个像素的蕴含着第一层5*5范围内信息。感受野的值越大表示能接触到的原始图像范围就越大，也意味着可能蕴含的信息更为全局，语义层次更高的特征；相反，值越小则表示其所包含的特征越趋向局部和细节。

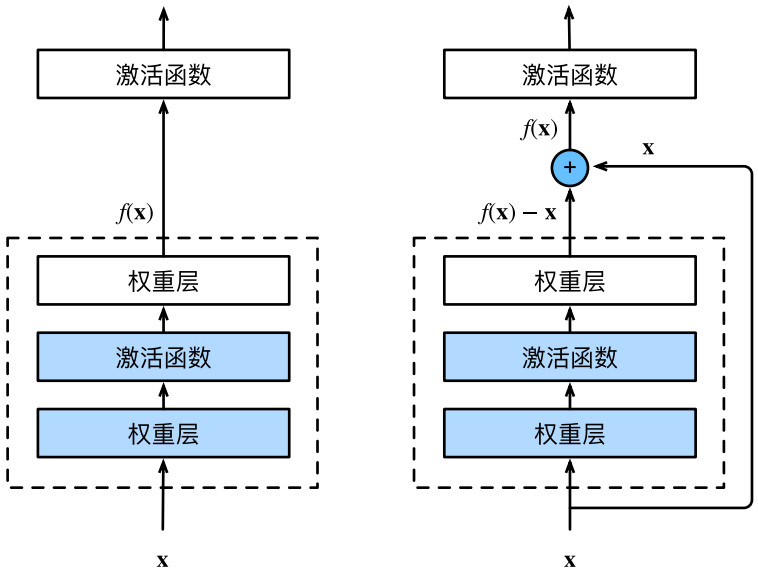


ResNet 残差连接解决梯度消失问题

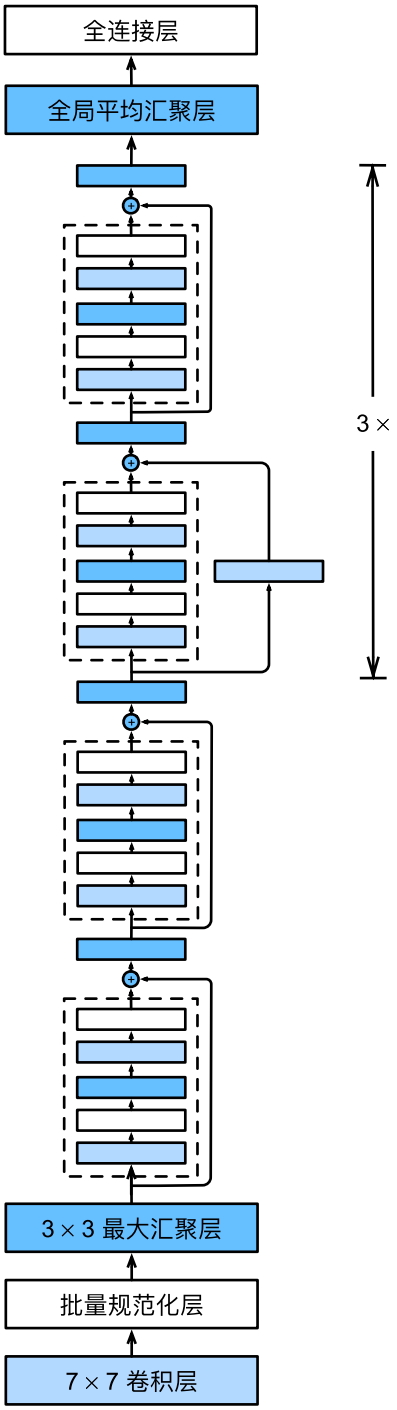
当人们发现深度卷积网络能有效从图片中提取特征时，人们就致力于设计更深更大的神经网络。但是人们发现，深层的神经网络往往难以训练，其中最主要的原因就是梯度消失（某一层的梯度≈0）和梯度爆炸（某一层的梯度≈无穷大）。曾经，人们认为像VGG-19这样的网络已经是网络深度的极限了，这样的网络足以被称为深度卷积神经网络，直到Kaiming He, Shaoqing Ren and Jian Sun带着他们的ResNet以难以置信的1202层、1940万参数（19.4M）炸裂登场。

	# layers	# params	
FitNet [34]	19	2.5M	8.39
Highway [41, 42]	19	2.3M	7.54 (7.72±0.16)
Highway [41, 42]	32	1.25M	8.80
ResNet	20	0.27M	8.75
ResNet	32	0.46M	7.51
ResNet	44	0.66M	7.17
ResNet	56	0.85M	6.97
ResNet	110	1.7M	**6.43** (6.61±0.16)
ResNet	1202	19.4M	7.93

ResNet提出了残差块，将输入直接加到输出上，使得神经网络表达形式变成了$f(x) = g(x) + x$，让神经网络从直接学习$f(x)$变成了学习残差$g(x) = f(x) - x$，事实证明残差$g(x)$比$f(x)$更好学习，有效减缓了梯度消失的问题。如下图，左边是普通的卷积块，右图为残差块。



下图为resnet-18结构图



MobileNet 深度可分离卷积

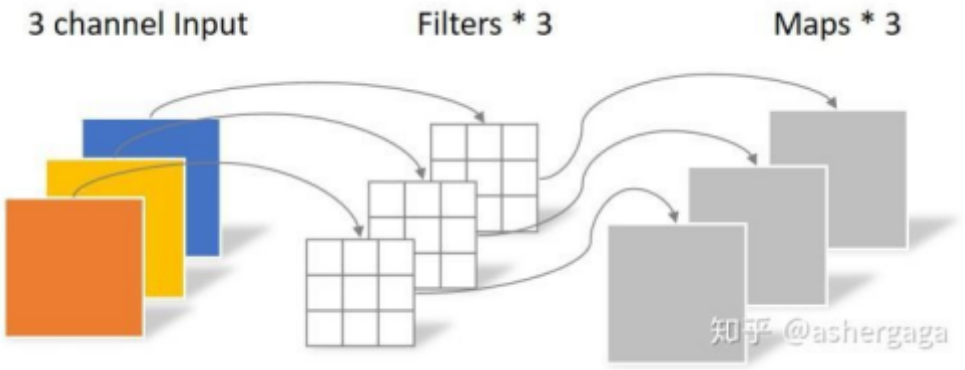
> 我们令`N,C,H,W = feature_map.shape`，N表示批量数，C表示特征图的通道数，H,W分别表示特征图的高和宽

人们发现，增加通道数(C)，也就是特征图的维度，能够更有效地提高网络的性能，但是伴随而生的问题是：**使用常规卷积增加通道数会带来大量的计算**

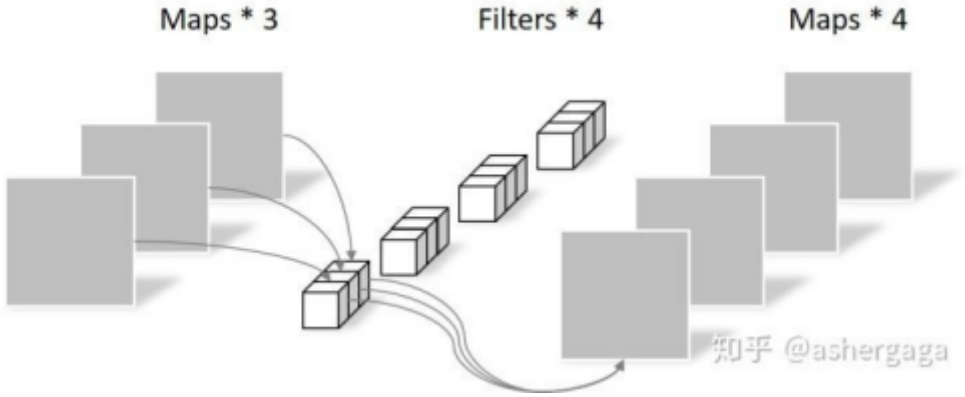
为了解决这个问题，google研究团队提出了MobileNet，创新性地采用深度可分离卷积替换常规卷积，减少计算量。简单概括就是，使用普通卷积提取特征，使用1x1卷积增加通道数。

深度可分离卷积将常规卷积分成了两个部分

- 逐通道卷积（depth-wise conv）：使用与输入通道数相同数量的卷积核进行卷积，输出通道与输入通道数相同
- 逐点卷积（point-wise conv）：即使用多个1x1卷积对逐通道卷积的结果进行升维，增加通道数



上图为逐通道卷积，下图为逐点卷积。



同样将3通道的张量上升为16通道张量的操作中，传统3x3卷积参数量为3x3x3x16 = 432,深度可分离卷积参数量为3x3x3+1x1x3x16 = 75 ，可见深度可分离卷积的参数量更少。

深度可分离卷积证明了，除了在空间上卷积提取特征外（普通卷积），融合不同通道的特征（1x1卷积）也能提高网络的性能，这个idea影响了后来的ShuffleNet、GhostNet的设计。

Heads：从分类到目标检测

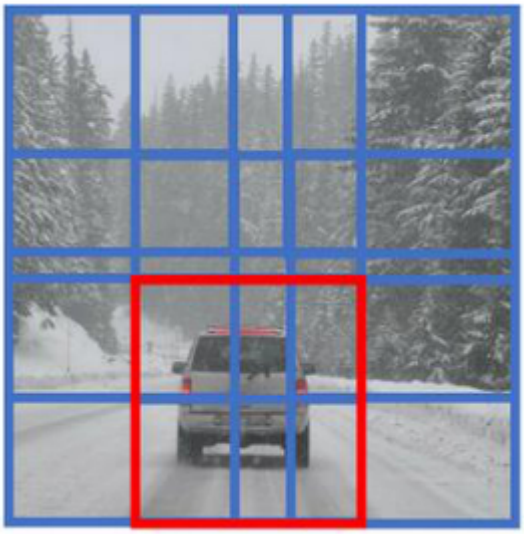
在前面的各种网络设计中，都是在设计前面的卷积层，而卷积层后的全连接层未曾发生过变化。主要是因为，这些网络在提出时，都是在图像分类数据集上进行性能的评估的，而要让网络适配其他的任务，就要修改网络的检测头。

全连接检测头

我们知道全连接网络的输出是一个n维向量。如果我们要进行图像的n分类，我们可以用全连接层作为网络的输出层，输出向量的第i个值表示对类别C_i的概率，所以图像的类别就是概率最大的那个类。

滑动窗口进行目标检测

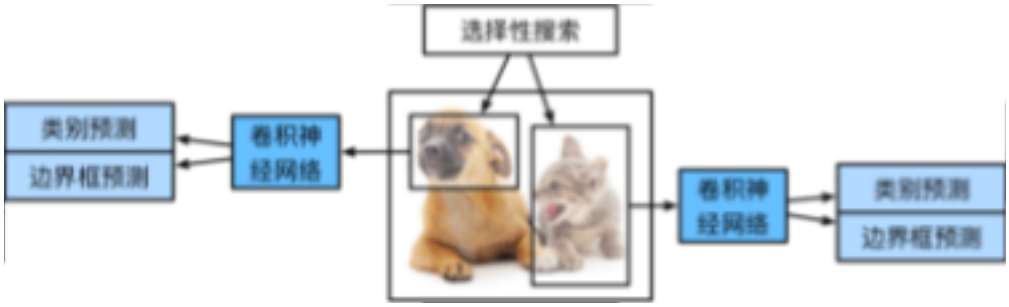
目标检测任务不止需要类别，还需要这个类别在图像中的位置，显然全连接检测头不能再满足我们的需要，在最早的时候，人们使用滑动窗口法进行目标检测。它的原理很简单，使用一个固定的窗口，扫描图像，对窗口内的图像进行分类，如果有目标，生成一个目标框。很显然这样的方法会耗费大量的算力。



两阶段的目标检测

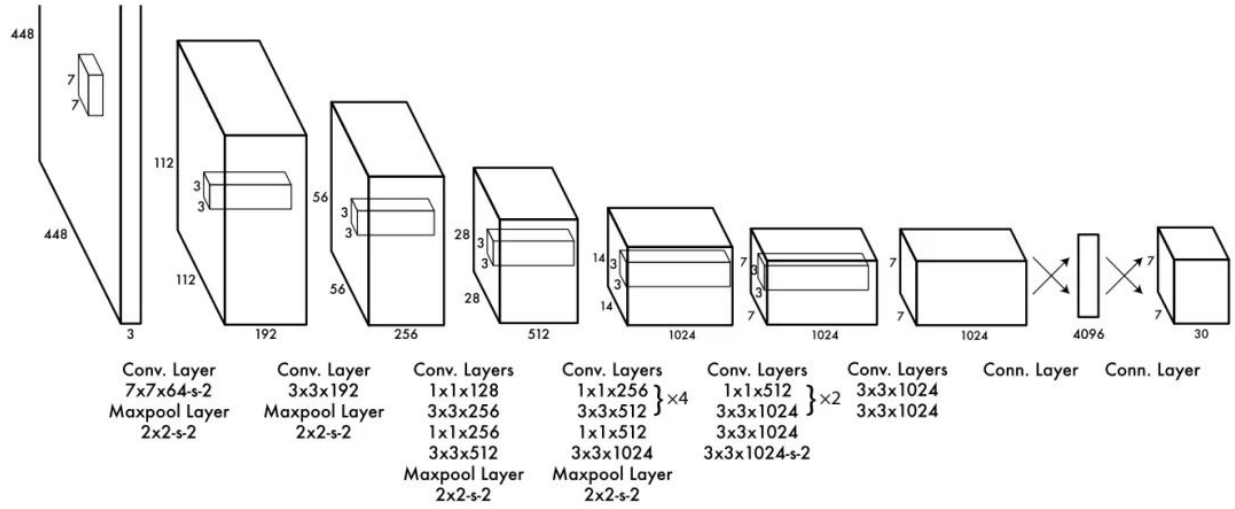
全图的滑动窗口扫描过于耗费算力，我们可以先利用一些方法提取出可能存在目标的区域(proposal)，在对这些区域进行分类。这种先找框再分类的方法被称为两阶段（Two-Stage）目标检测。RCNN系列是其中的代表。

- RCNN：使用基于传统视觉的选择性搜索方法提取proposal
- Faster-RCNN： 使用RPN网络提取proposal



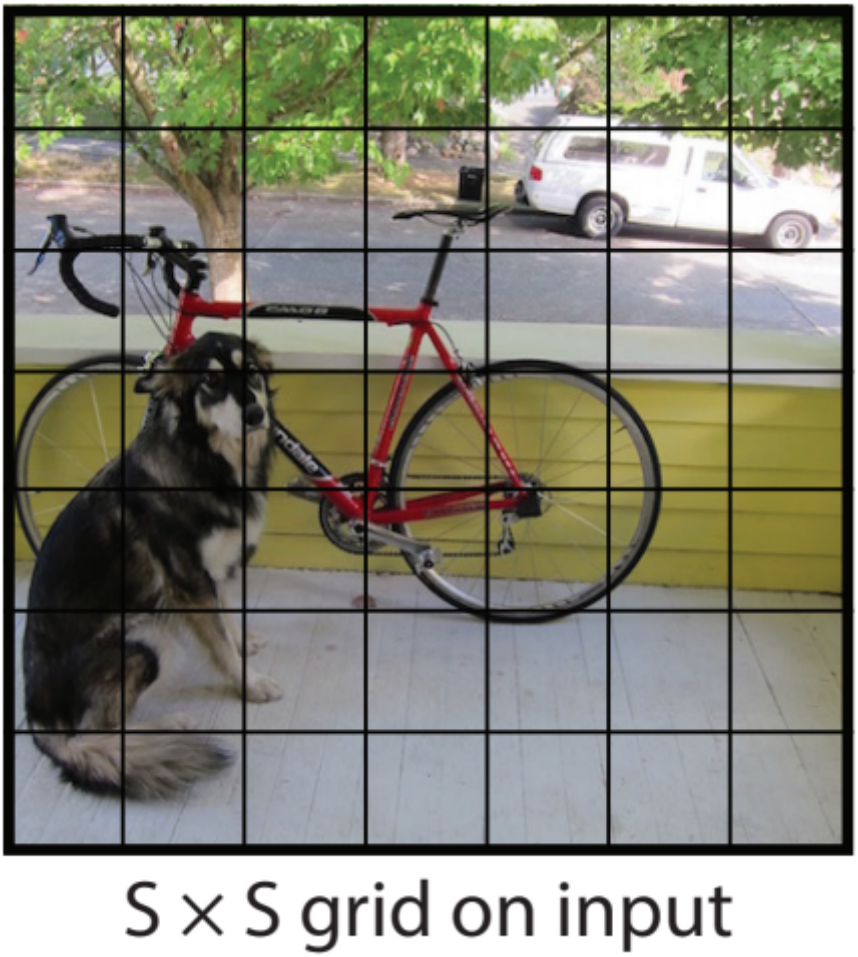
一阶段的目标检测

所谓一阶段，就是直接一步到位同时生成框和类别，YOLO（You Only Look Once）就是一阶段目标检测的代表之作。下图为YOLOv1的网络结构图，可以看见，他输出了一个7x7x30的张量。



这个7x7x30怎么理解呢？

看下图，YOLOv1将图片均匀分割为7x7个格子（grid），每个格子负责预测一个目标。我们可以用obj_{score}代表这个格子存在目标的概率。除了预测有没有目标，我们还要预测这个目标框的形状即x, y, w, h，假设我们需要识别25个类别的目标，则每个格子还要预测25个目标对应的概率$cls_{score}1, cls_{score}2, \ldots, cls_{score}25$。所以一个格子需要输出1+4+25个值，所以整张图像需要输出7x7x30的值。

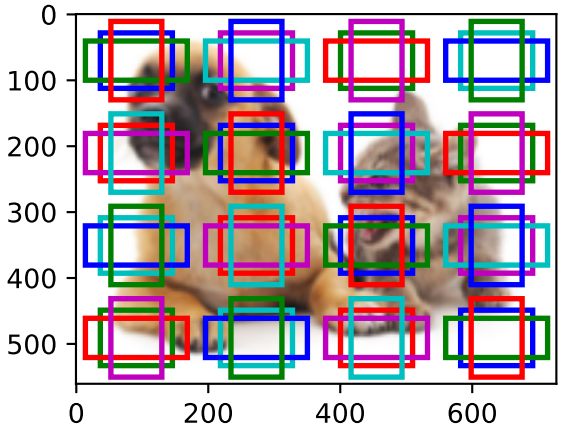


Anchor-free和Anchor-based

YOLOv1是第一个能够实时运行（>24FPS）的目标检测模型，但它有着两个明显的缺点：

- 一张图片只能检测出7x7=49个目标
- 检测框的精度远不如RCNN系列

为了解决这两个问题，YOLO9000（就是YOLOv2）诞生了。YOLO9000采用了Anchor-based的设计方式，就是给每个格子提前设计好三个不同形状的锚框，如下图所示，每个锚框预测一个目标。与YOLOv1直接输出框的大小的不同，YOLOv2输出的是框相对于锚框的偏差。采用锚框的设计让YOLOv2能够同时识别$S*S*3$个目标，并且提高了检测框的精度。



像这样提前设计好锚框的网络，我们成为Anchor-based，而YOLOv1这种不依赖锚框的网络我们称之为Anchor-free。锚框的加入能够提高预测框的精度，然而，有意思的是，从YOLOX开始，现代的CNN又重新拥抱了Anchor-free的设计，并且Anchor-free能够达到比Anchor-based更高的性能，可能是因为backbone的性能越来越好了吧。

Decoupled Head

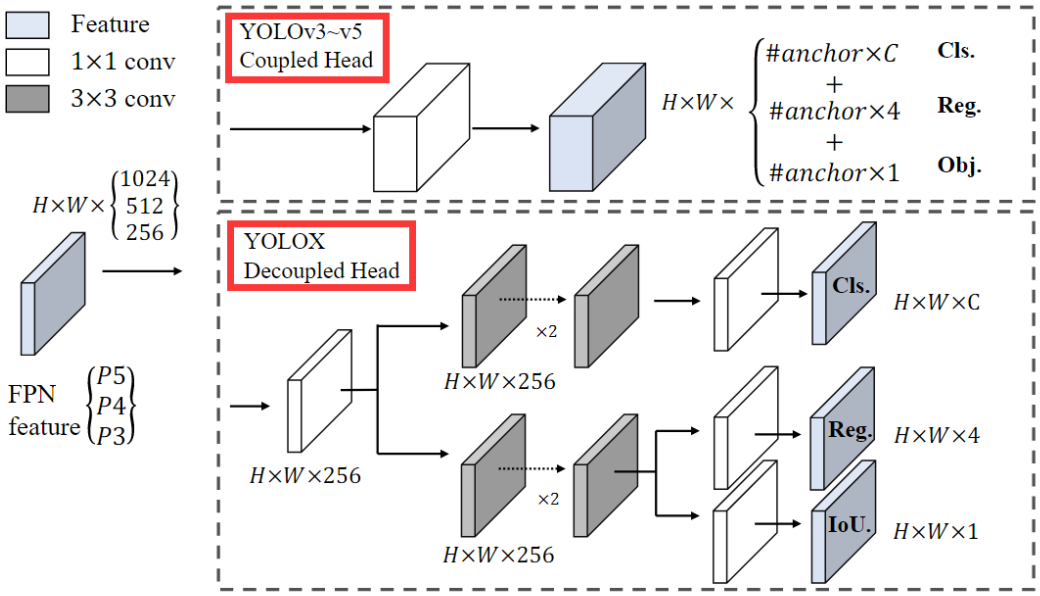
从YOLOv1到YOLOv5，目标检测网络都采用耦合检测头(coupled head)的设计。

观察YOLOv1的网络结构，我们可以知道，不管是目标得分Obj_{score}、检测框位置x,y,w,h，还是分类得分Cls_{score}都是经过同一个卷积层得到的输出，也就是经过相同的特征和特征提取方法。然而不同的输出可能需要不同的特征和不同的参数，这样耦合的设计可能会影响最终的性能。

Jian Sun和它的团队提出了YOLOX，采用了解耦头设计，也就是在backbone后面同时接多个检测头，每个检测头负责其中的一个输出，这样的设计提高了网络的性能。

> 悼念孙剑大神

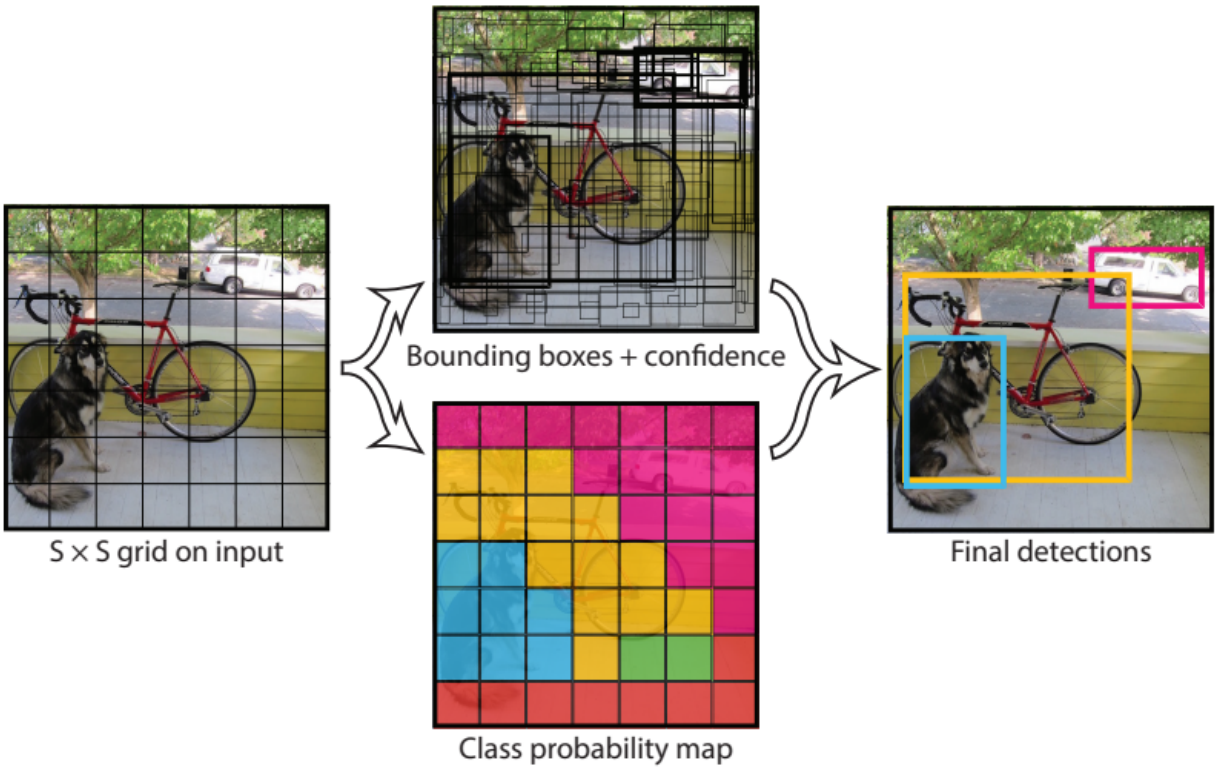
如下图所示，YOLOX的Decoupled Head采用了三个不同的卷积层，分别进行分类（Cls）和检测框预测（Reg.）任务。



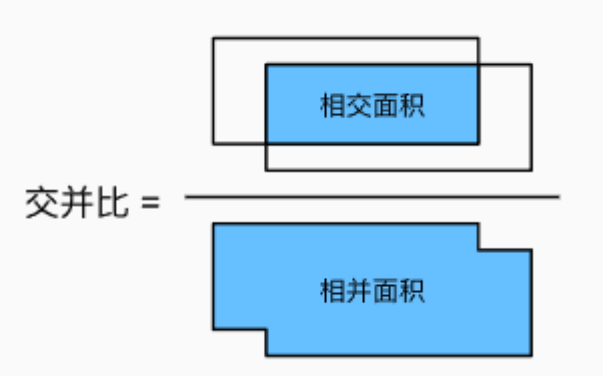
非极大值抑制 NMS算法

在我们前面提到的检测头设计中，不管采用哪种方法，都存在对同一个目标会产生多个检测框的问题。当一个目标很大，同时占据多个grid的时候，不可避免的有多个grid对这个目标产生了预测。NMS算法是一种去掉重叠框，为每个目标保留唯一检测框的算法。其流程如下：

1. 找到Obj_{Score}和Cls_{Score}综合得分最高的那个框，保留这个框A
2. 找到与框A交并比过高的若干个框，丢掉
3. 剩下的框重复1，2步直到没有重复框为止



交并比(IOU)的计算如下：

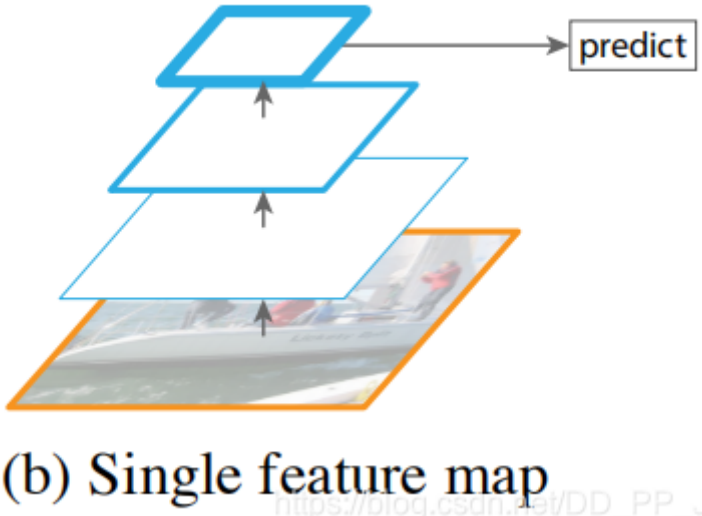


Necks ：从多尺度检测头到特征金字塔

在介绍GoogLeNet和Inception模块时，提到，不同的感受野对应着不同尺度的特征。大的感受野对应全局的特征，小的感受野对应细节特征。在目标检测领域，往往有不同大小的目标，大的目标往往需要全局的特征，小目标则需要一些很细节的特征，也就是说**不同目标的识别需要的感受野不同**。

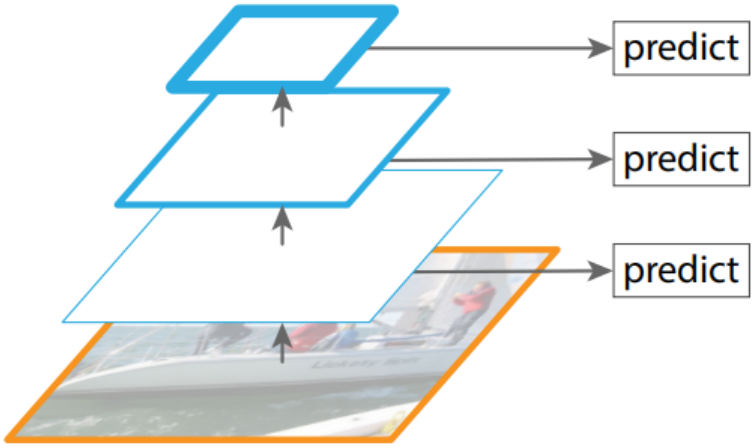
然而，随着卷积神经网络的逐层计算，感受野不可避免的会越来越大，图像的很多细节信息在传递中消失了，导致早期的目标检测网络对**小目标**的识别效果不好。

如下图，早期的CNN只使用最后一次卷积输出的特征图进行预测。



多尺度特征图

为了改善对小目标的识别效果，人们提出了多检测头的设计，将CNN的中间特征图提取出来，连接到一个检测头上进行预测，低层特征图负责小目标的检测，高层特征图负责大目标检测。



(c) Pyramidal feature hierarchy

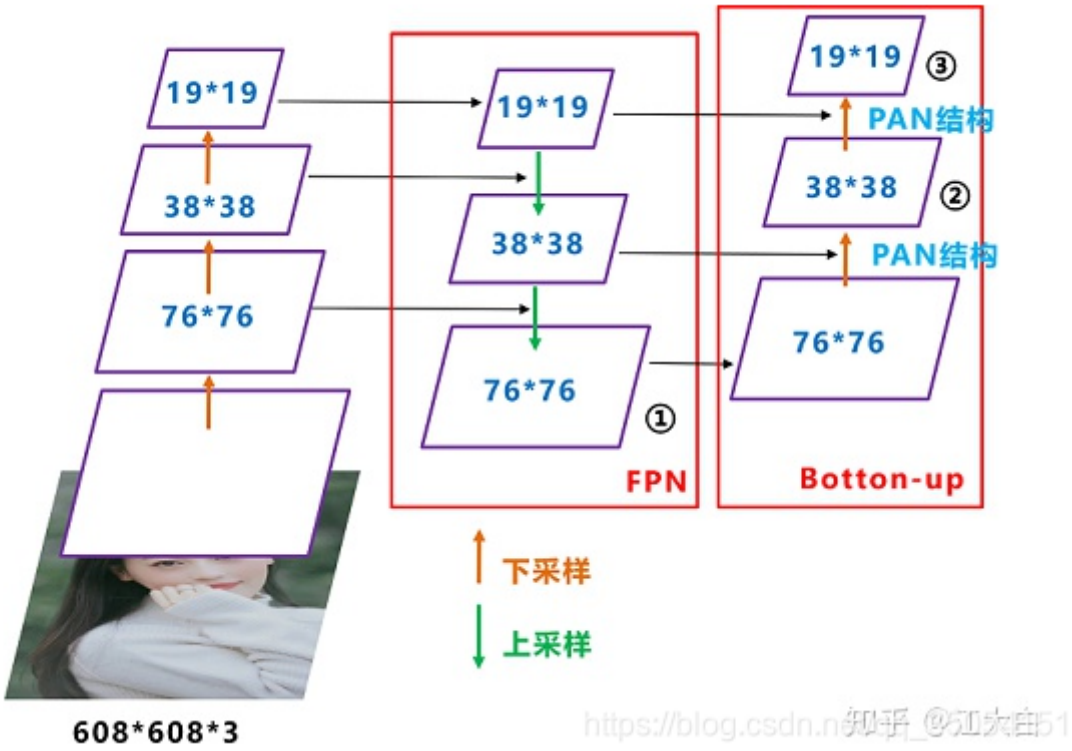
FPN特征金字塔

上面的多检测头设计有个问题，就是低层特征图因为卷积次数较少，缺少全局的信息，导致误识别率上升。为了将高层次的特征图的信息融合进低层次特征图中，人们提出了特征金字塔网络（Feature Pyramidal Network， FPN），将高层次的特征图经过上采样融合到低层特征图上，使低层的特征图获取到了全局的信息，改善识别效果。

(d) Feature Pyramid Network

PAN 加了一条bottom-up路径

既然我们能让高层次细节转移到低层次特征图中，自然而然也能将低层次细节转移到高层次特征图中，于是PAN诞生了。相比FPN，PAN多了一条bottom-up的路径。



结论：CNN=A+B+C

在上面，我们花了大量的时间给大家介绍了卷积神经网络的发展，主干网络从简单的LeNet网络进化到上百层的ResNet，检测头从全连接头进化到YOLOX的Decouled Head，Neck的提出让网络能够适应不同大小的目标检测任务。

当然，这些东西都是好几年前的东西了，随着时间的发展，目标检测模型越来越多，YOLO系列也来到了YOLOV8，但不管是YOLOv5还是v6、v7、v8或者其他目标检测网络，他们都采用了下面的三明治结构

$$ObjectDetectionModel = Backbone + Neck + Head$$

采用不同的Backbone、Neck、Head可以组合出不同的网络，可以说，CNN=A+B+C，比的就是你的A、B、C好不好使

5. YOLOv5实战篇

5.1 认识数据集格式

常见的数据集格式包括COCO格式、YOLO格式以及VOC格式

> COCO和VOC是两个著名的目标检测竞赛，直到今天COCO2017数据集的mAP0.5得分仍然没有超过0.9

- COCO格式：采用json文件存储图像的标签信息
- YOLO格式：采用txt文件存储图像的标签信息
- VOC格式：采用xml文件存储图像的标签信息

这里我们介绍YOLO格式，很简单，每一个图像对应一个同名的txt文件，txt文件的每一行对应一个检测框，每一行有5个数字，分别表示{类别, x, y, w, h}，要注意的是框的信息x, y, w, h是归一化后的值，范围为$[0, 1]$

```
1 0.352 0.410 0.122 0.092
2 0.149 0.801 0.035 0.101
```

YOLOv5训练需要一个train.txt文件和val.txt文件，其中train.txt记录了有多少张图片用于训练，val.txt记录有多少张图片用于验证。如下是train.txt的部分截取，可见train.txt和val.txt的每一行是训练图片的路径，要注意**该图片的同级路径下应该要用一个与图片同名的txt标签文件**

> 我们一般会将数据集按9:1的比例划分为训练集和验证集。

```
/home/zcf/Desktop/FasterPoints/socket/data/379.png
/home/zcf/Desktop/FasterPoints/socket/data/548.png
/home/zcf/Desktop/FasterPoints/socket/data/787.png
/home/zcf/Desktop/FasterPoints/socket/data/868.png
/home/zcf/Desktop/FasterPoints/socket/data/599.png
/home/zcf/Desktop/FasterPoints/socket/data/556.png
/home/zcf/Desktop/FasterPoints/socket/data/803.png
/home/zcf/Desktop/FasterPoints/socket/data/519.png
/home/zcf/Desktop/FasterPoints/socket/data/119.png
```

一般的，数据集文件会有如下结构

```
.
├── train
│   ├── 1.png
│   ├── 1.txt
│   ├── 2.png
│   ├── 2.txt
│
├── val
│   ├── 1001.png
│   ├── 1001.txt
│
├── train.txt
└── val.txt
```

5.2 赛博苦力：打标签

当你准备好图片后（相机拍摄、视频截取等），可以使用一些打标工具进行标注，例如labelImg和labelme，以labelImg为例，其界面如下

使用步骤为：

1. 点击左侧`Open Dir`打开存放有训练图片的文件夹
2. 点击左侧`Change Save Dir`修改标签文件保存路径
3. 左侧`Save`按钮的下一个按钮表示标签的格式，如果不是YOLO格式请切换为YOLO格式
4. `w`键创建一个检测框，用鼠标拖动至正确位置后，修改标签的label为类别名
5. `a`键和`d`键可以切换上一张、下一张图片

5.3 划分数据集

打完标签后，可以用Python脚本将数据集Look 划分为训练集和测试集，并生成train.txt和test.txt，脚本代码可以上网上搜索，这里给个示例，**记得根据实际情况去改代码**，使用方法为

```
python spilit_data.py /your/dataset/path
```

```
# split_data.py
import os
import random
import sys
from shutil import copyfile

from sympy import root

if len(sys.argv) < 2:
    print("no directory specified, please input target directory")
    exit()

root_path = sys.argv[1]

img_path = root_path + '/images'
```

```
label_path = root_path + '/label'
train_path = root_path +'/train'
val_path = root_path + '/val'

if not os.path.exists(root_path):
    print("cannot find such directory: " + root_path)
    exit()

if not os.path.exists(train_path):
    os.makedirs(train_path)

if not os.path.exists(val_path):
    os.makedirs(val_path)

train_percent = 0.9
total_img = os.listdir(img_path)
num = len(total_img)
list = range(num)
train_num = int(num*train_percent)
train_set = random.sample(list,train_num)

print("train size :",train_num)
print("val size: ",num-train_num)

ftrain = open(root_path+'/train.txt','w')
fval = open(root_path+'/val.txt','w')

for i in list:
    name = total_img[i][:]
    prename = total_img[i][:-4]
    if i in train_set:
        ftrain.write(train_path+'/'+name+'\n')
        #copy img
        copyfile(os.path.join(img_path,name),os.path.join(train_path,name))
            #copy label

 copyfile(os.path.join(label_path,prename+'.txt'),os.path.join(train_path,prename+'.
txt'))
    else:
        fval.write(val_path+'/'+name+'\n')
        copyfile(os.path.join(img_path,name),os.path.join(val_path,name))
            #copy label

 copyfile(os.path.join(label_path,prename+'.txt'),os.path.join(val_path,prename+'.tx
t'))

ftrain.close()
fval.close()
```

划分完数据后将数据集复制到datasets文件夹下

5.4 训练YOLOv5

安装yolov5

```
git clone https://github.com/ultralytics/yolov5  # clone
cd yolov5
pip install -r requirements.txt  # install
```

在data文件夹下新建一个mydata.yaml，输入以下内容

```
path: datasets/mydata  # 数据集根路径
train: train.txt  # 训练集路径，这里为../datasets/mydata/train.txt
val: val.txt  # 验证集路径，这里为../datasets/mydata/val.txt
test: test.txt  # 测试集路径，可选

# 类别编号
names:
  0: person
  1: bicycle
```

训练模型，中间的可视化结果以及保存的模型放在runs文件夹下

```
python train.py --data mydata.yaml --epochs 300 --cfg yolov5s.yaml  --batch-size 32
--weights yolov5s.pt
```

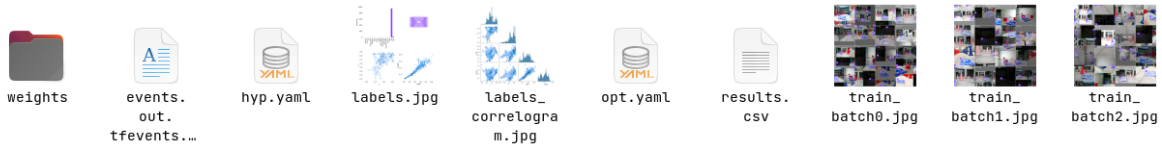
这里有用到几个参数

- epochs：训练的迭代次数，越大则训练时间越长，过大可能导致模型过拟合，过小无法得到最优模型
- batch-size：每个批量输入到GPU中的图片数目，过大容易导致显存不足`CUDA out of memory`，过小导致训练速度过慢
- cfg：YOLOv5是一个模型族，有不同大小的模型设计，从YOLOv5n到YOLOv5x，模型越来越大，精度越来越高，速度也月来越慢
- weights：读取预训练的权重文件

训练过程如图所示

```
  4                -1  2    115712  models.common.C3                        [128, 128, 2]
  5                -1  1    295424  models.common.Conv                      [128, 256, 3, 2]
  6                -1  3    625152  models.common.C3                        [256, 256, 3]
  7                -1  1   1180672  models.common.Conv                      [256, 512, 3, 2]
  8                -1  1   1182720  models.common.C3                        [512, 512, 1]
  9                -1  1    656896  models.common.SPPF                      [512, 512, 5]
 10                -1  1    131584  models.common.Conv                      [512, 256, 1, 1]
 11                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 12          [-1, 6]  1         0  models.common.Concat                    [1]
 13                -1  1    361984  models.common.C3                        [512, 256, 1, False]
 14                -1  1     33024  models.common.Conv                      [256, 128, 1, 1]
 15                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 16          [-1, 4]  1         0  models.common.Concat                    [1]
 17                -1  1     90880  models.common.C3                        [256, 128, 1, False]
 18                -1  1    147712  models.common.Conv                      [128, 128, 3, 2]
 19         [-1, 14]  1         0  models.common.Concat                    [1]
 20                -1  1    296448  models.common.C3                        [256, 256, 1, False]
 21                -1  1    590336  models.common.Conv                      [256, 256, 3, 2]
 22         [-1, 10]  1         0  models.common.Concat                    [1]
 23                -1  1   1182720  models.common.C3                        [512, 512, 1, False]
 24      [17, 20, 23]  1     59334  models.yolo.Detect                      [17, [[10, 13, 16, 30, 33, 23], [30, 61, 62, 45, 59, 119], [116, 90, 156, 198, 373, 326]], [128, 256,
YOLOv5s summary: 214 layers, 7065478 parameters, 7065478 gradients, 16.1 GFLOPs

Transferred 342/349 items from yolov5s.pt
AMP: checks passed ✅
optimizer: SGD(lr=0.01) with parameter groups 57 weight(decay=0.0), 60 weight(decay=0.0005), 60 bias
train: Scanning /home/zcf/Downloads/yolov5-master/datasets/racecar/train.cache... 969 images, 2 backgrounds, 0 corrupt: 100%|██████████| 969/969 [00:00<?, ?it/s]
val: Scanning /home/zcf/Downloads/yolov5-master/datasets/racecar/val.cache... 108 images, 0 backgrounds, 0 corrupt: 100%|██████████| 108/108 [00:00<?, ?it/s]

AutoAnchor: 3.62 anchors/target, 1.000 Best Possible Recall (BPR). Current anchors are a good fit to dataset ✅
Plotting labels to runs/train/exp5/labels.jpg...
Image sizes 640 train, 640 val
Using 8 dataloader workers
Logging results to runs/train/exp5
Starting training for 100 epochs...

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
       0/99      3.51G    0.08135    0.02267    0.03487         11        640: 100%|██████████| 61/61 [00:15<00:00,  3.87it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 4/4 [00:01<00:00,  3.55it/s]
                   all        108        108      0.495      0.917      0.504      0.206

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
       1/99      4.13G    0.07729    0.01503    0.01017         21        640: 100%|██████████| 61/61 [00:16<00:00,  3.79it/s]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 4/4 [00:00<00:00,  4.74it/s]
                   all        108        108      0.389      0.398      0.363      0.081

      Epoch    GPU_mem   box_loss   obj_loss   cls_loss  Instances       Size
       2/99      4.13G     0.0645     0.0154   0.009607         28        640:  15%|▉         | 9/61 [00:02<00:13,  3.78it/s]
```

训练过程中的文件保存在runs文件夹中，保存的pt模型在runs/weighs中



5.4 调用模型推理

模型训练完后，将runs/exp/weights中的模型(best.pt)复制出来，可以调用detect.py进行推理

```
python detect.py --weights best.pt --source 测试图片.jpg
```