

成绩：

**项 目 报 告**

**项目名称： 线上舌苔只能诊断系统**

**课题名称： 基于实例分割的舌苔图像识别**

**指导教师： 李成严**

**组 长： 19-10 简程 1804011005**

**组 员： 19-2 张卫龙 1904010226**

**组 员： 19-2 白弘伟 1904010201**

**组 员： 19-2 姜云建 1904010205**

**专 业： 计算机科学与技术**

**计算机科学与技术学院**

2022 年 3 月 24 日

检测部分：

基于yolov4的目标检测：

[YOLOV4结构解析](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "YOLOV4_48" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

[1、主干特征提取网络Backbone](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "1Backbone_52" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

[2、特征金字塔](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "2_264" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

[3、YoloHead利用获得到的特征进行预测](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "3YoloHead_373" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

[4、预测结果的解码](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "4_466" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

[5、在原图上进行绘制](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "5_715" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

基于yolov4搭建舌体检测网络

基于unet的语义分割：

[unet结构解析](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "YOLOV4_48" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

基于unet搭建舌体分割网络

[1、主干网络介绍](https://blog.csdn.net/weixin_44791964/article/details/111303409" \l "1_24" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

[2、加强特征提取结构](https://blog.csdn.net/weixin_44791964/article/details/111303409" \l "2_149" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

[3、利用特征获得预测结果](https://blog.csdn.net/weixin_44791964/article/details/111303409" \l "3_218" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

利用搭建的unet网络进行舌苔切割

基于Resnet50的分类网络：

[Resnet50结构解析](https://blog.csdn.net/weixin_44791964/article/details/106214657" \l "YOLOV4_48" \t "https://blog.csdn.net/weixin_44791964/article/details/_self)

基于Resnet搭建分类神经网络

利用搭建分类神经网络进行分类

训练部分：

Yolov4训练：

基于神经网络前向传播和后向传播训练模型

Unet训练：

基于keras版本训练模型

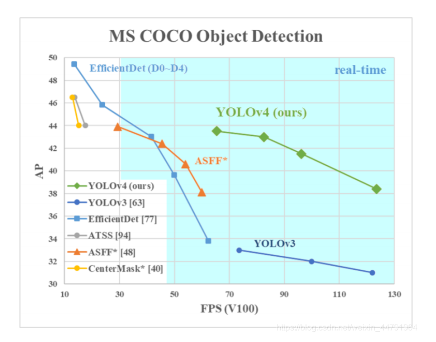
Resnet训练：

基于GradientDescent梯度下降训练模型

舌体检测：

在对比多个深度学习网络后，我们使用yolov4来作为我们的舌体检测网络，因为舌像目标大，特征明显，而大目标的检测一直是属于yolo的强项，且yolo在精度良好的同时检测速度快，只需不到1s就能完成图片解析，故选择yolov4.

**什么是YOLOV4**



YOLOV4是YOLOV3的改进版，在YOLOV3的基础上结合了非常多的小Tricks。

尽管没有目标检测上革命性的改变，但是YOLOV4依然很好的结合了速度与精度。

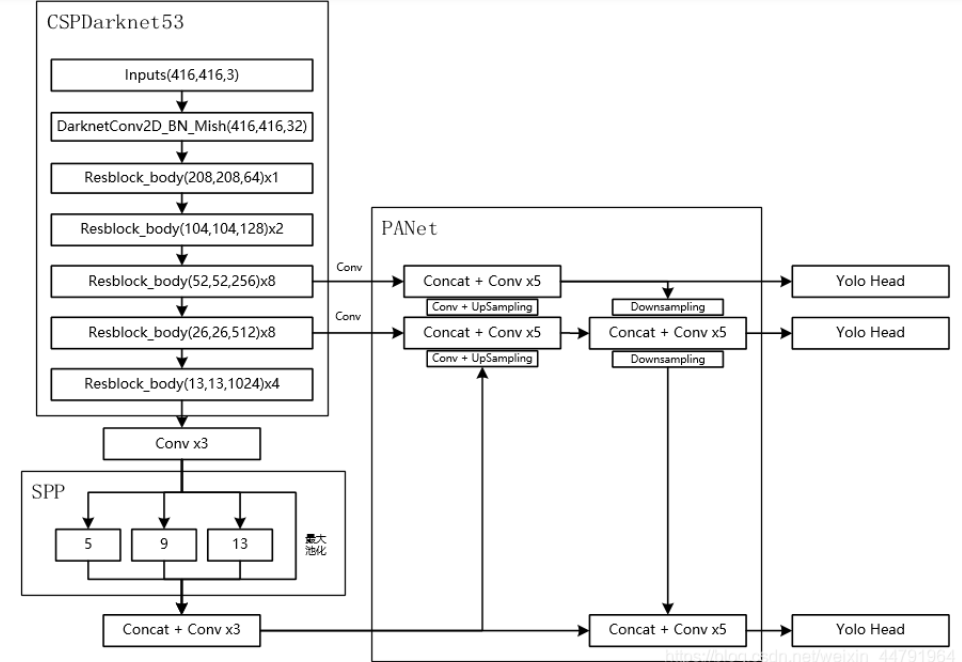
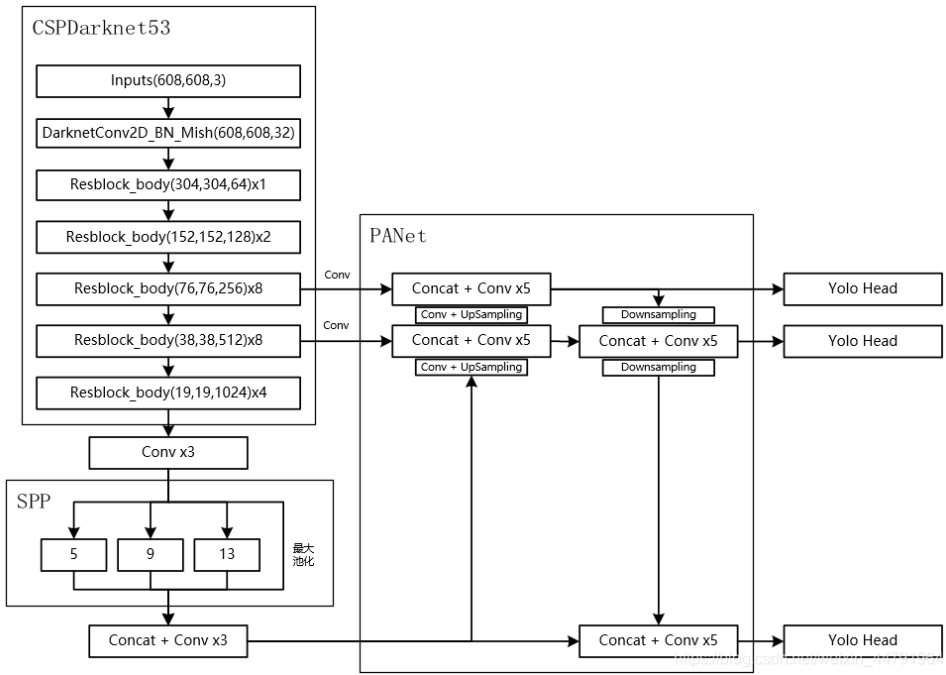
根据上图也可以看出来，YOLOV4在YOLOV3的基础上，在FPS不下降的情况下，mAP达到了44，提高非常明显。

YOLOV4整体上的检测思路和YOLOV3相比相差并不大，都是使用三个特征层进行分类与回归预测。

# **YOLOV4结构解**

在下面我们使用coco数据集为例（含有80个类别）展示yolov4结构

## **1、主干**[特征提取](https://so.csdn.net/so/search?q=%E7%89%B9%E5%BE%81%E6%8F%90%E5%8F%96&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_44791964/article/details/_blank)**网络Backbone**

当输入是416x416时，特征结构如下：  
  
当输入是608x608时，特征结构如下：  


## **2.特征金字塔**

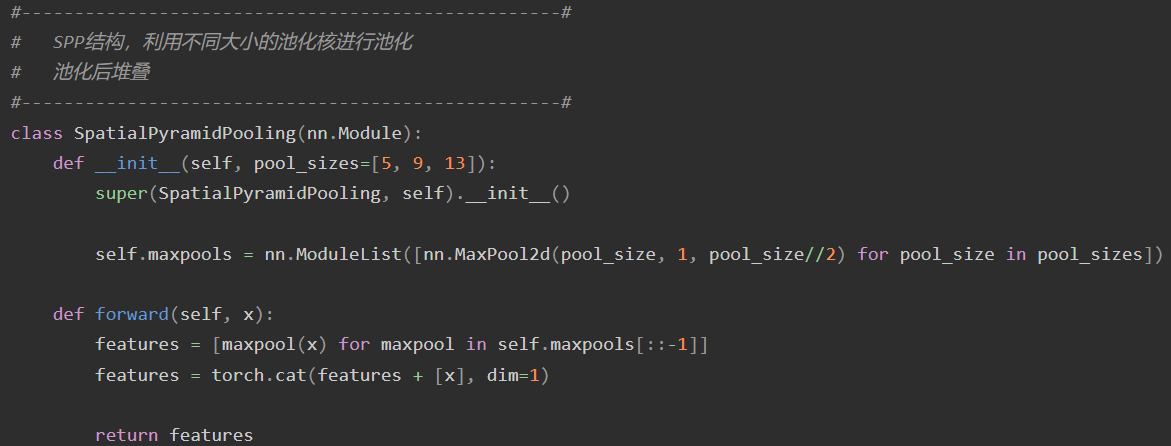
在特征金字塔部分，YOLOV4结合了两种改进:

a).使用了SPP结构。

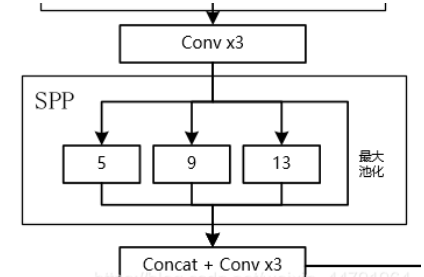
b).使用了PANet结构。

如上图所示，除去CSPDarknet53和Yolo Head的结构外，都是特征金字塔的结构。

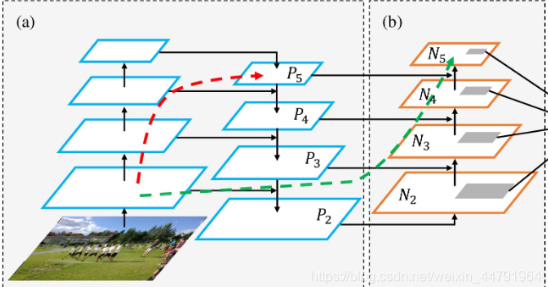
1、SPP结构参杂在对CSPdarknet53的最后一个特征层的卷积里，在对CSPdarknet53的最后一个特征层进行三次DarknetConv2D\_BN\_Leaky卷积后，分别利用四个不同尺度的最大池化进行处理，最大池化的池化核大小分别为13x13、9x9、5x5、1x1（1x1即无处理）



其可以****它能够极大地增加感受野，分离出最显著的上下文特征****。

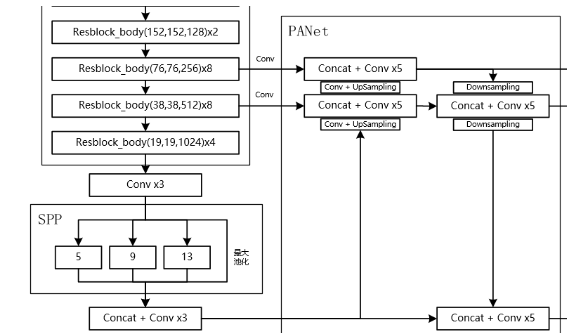


1. PANet是2018年提出的一种实例分割算法，其具体结构由反复提升特征的意思。



上图为原始的PANet的结构，可以看出来其具有****一个非常重要的特点就是特征的反复提取****。  
在（a）里面是传统的特征金字塔结构，在完成特征金字塔从****下到上的特征提取后****，还需要****实现（b）中从上到下的特征提取。****

而在YOLOV4当中，其主要是在****三个有效特征层上使用了PANet结构。****



## **3、YoloHead利用获得到的特征进行预测**

1、在特征利用部分，YoloV4提取多特征层进行目标检测，一共提取三个特征层，分别位于中间层，中下层，底层，三个特征层的shape分别为(76,76,256)、(38,38,512)、(19,19,1024)。

2、输出层的shape分别为(19,19,75)，(38,38,75)，(76,76,75)，最后一个维度为75是因为该图是基于voc数据集的，它的类为20种，YoloV4只有针对每一个特征层存在3个先验框，所以最后维度为3x25；

如果使用的是coco训练集，类则为80种，最后的维度应该为255 = 3x85，三个特征层的shape为(19,19,255)，(38,38,255)，(76,76,255)

而我们的舌体种类只有一个类别---tongue，所以最后的维度是3x1=3，输出的shape为(19,19,3)，(38,38,3)，(76,76,3)

4、预测结果的解码

由第二步我们可以获得三个特征层的预测结果，shape分别为(N,19,19,255)，(N,38,38,255)，(N,76,76,255)的数据，对应每个图分为19x19、38x38、76x76的网格上3个预测框的位置。

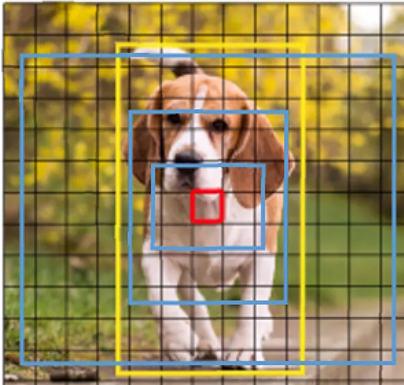
但是这个预测结果并不对应着最终的预测框在图片上的位置，还需要解码才可以完成。

此处要讲一下yolo3的预测原理，yolo3的3个特征层分别将整幅图分为19x19、38x38、76x76的网格，每个网络点负责一个区域的检测。

我们知道特征层的预测结果对应着三个预测框的位置，我们先将其reshape一下，其结果为(N,19,19,3,85)，(N,38,38,3,85)，(N,76,76,3,85)。

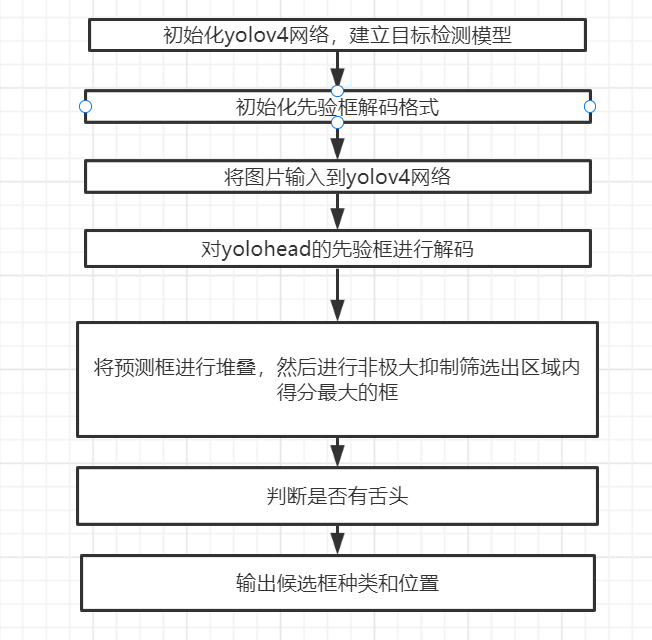
最后一个维度中的85包含了4+1+80，分别代表x\_offset、y\_offset、h和w、置信度、分类结果。

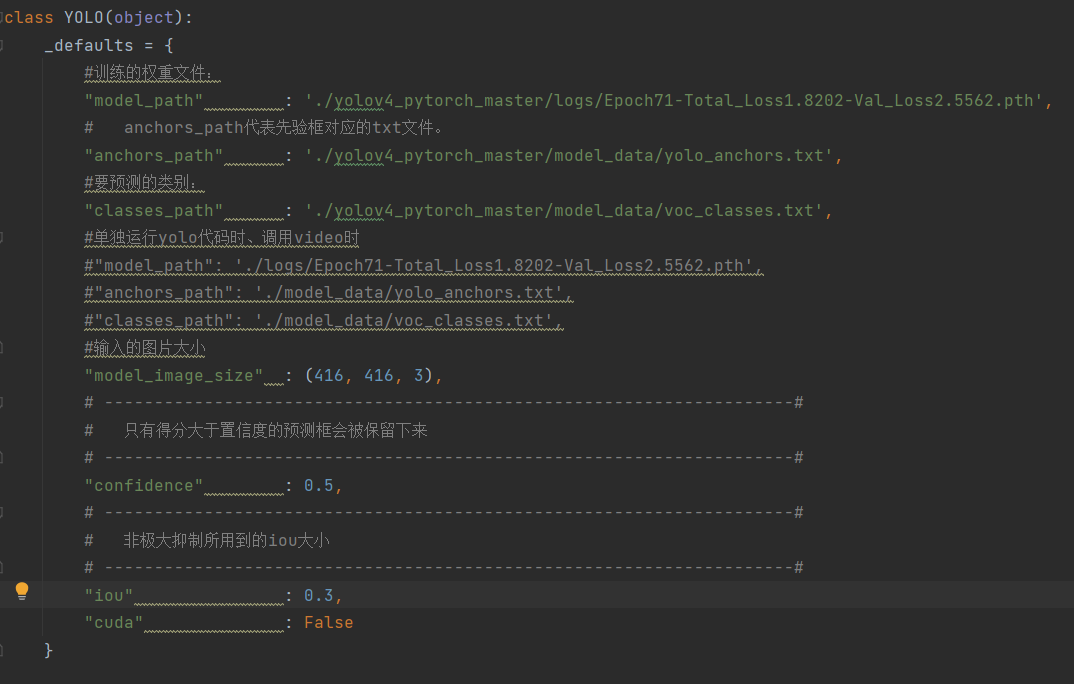
yolo3的解码过程就是将每个网格点加上它对应的x\_offset和y\_offset，加完后的结果就是预测框的中心，然后再利用 先验框和h、w结合 计算出预测框的长和宽。这样就能得到整个预测框的位置了。



当然得到最终的预测结构后还要进行得分排序与非极大抑制筛选

**基于YOLOV4框架搭建舌体检测网络**





第一步：初始yolov4网络

#---------------------------------------------------#  
# 初始化YOLO  
#---------------------------------------------------#  
def \_\_init\_\_(self, \*\*kwargs):  
 self.\_\_dict\_\_.update(self.\_defaults)  
 self.class\_names = self.\_get\_class()  
 self.anchors = self.\_get\_anchors()  
 self.generate()

# ---------------------------------------------------#  
# 获得种类  
# ---------------------------------------------------#  
def \_get\_class(self):  
 classes\_path = os.path.expanduser(self.classes\_path)  
 with open(classes\_path) as f:  
 class\_names = f.readlines()  
 class\_names = [c.strip() for c in class\_names]  
 return class\_names  
  
#---------------------------------------------------#  
# 获得所有的先验框  
#---------------------------------------------------#  
def \_get\_anchors(self):  
 anchors\_path = os.path.expanduser(self.anchors\_path)  
 with open(anchors\_path) as f:  
 anchors = f.readline()  
 anchors = [float(x) for x in anchors.split(',')]  
 return np.array(anchors).reshape([-1, 3, 2])[::-1,:,:]

第二步：生成检测模型并加载权重文件

#---------------------------------------------------#  
# 生成模型  
#---------------------------------------------------#  
def generate(self):  
 #---------------------------------------------------#  
 # 建立yolov4模型  
 # anchors[0]：对应的先验框参数  
 #---------------------------------------------------#  
 self.net = YoloBody(len(self.anchors[0]), len(self.class\_names)).eval()  
 #我们在这里会得到yolohead的三个特征层的输出  
 #---------------------------------------------------#  
 # 载入yolov4模型的权重  
 #---------------------------------------------------#  
 print('Loading weights into state dict...')  
 #是否使用GPU  
 device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
 #下面两行代码是torch提供的加载模型以  
 state\_dict = torch.load(self.model\_path, map\_location=device)  
 #训练权重的函数  
 self.net.load\_state\_dict(state\_dict)

第三步：建立三个特征层的解码格式

后续对图片经过draknet53网络后的三个特征层的解码格式

#---------------------------------------------------#  
# 建立三个特征层的解码格式  
# 解码：预测网络会对对先验框调整，得到最后的预测框  
# 特征层的预测框大小和预测种类  
#---------------------------------------------------#  
self.yolo\_decodes = []  
for i in range(3):  
 self.yolo\_decodes.append(DecodeBox(self.anchors[i], len(self.class\_names), (self.model\_image\_size[1], self.model\_image\_size[0])))

第四步：检测图片

首先对图片预处理，因为yolov4只能处理特定长宽的图片，所以对图片预处理，将图片resize成相应大小

1. 计算输入图片的高和宽  
   2.给图像增加灰条，实现不失真的resize# 因为图片要调整成416\*416大小，如果不给多余的区域添加灰条，也会对添加的区域检测

# 将图片转换成torch形式  
images = torch.from\_numpy(np.asarray(images)) #从numpy.ndarray创建一个张量。  
#---------------------------------------------------------#  
# 将图像输入网络当中进行预测！  
#---------------------------------------------------------#  
outputs = self.net(images) #制一部得到的是yolohead的输出结果  
#yolohead一共得到三个张量，即三个特征层，分别为（N,18,13,13) (1,18,26,26) (1,18,52,52)

# ---------------------------------------------------------#  
# 对先验框解码，并将结果放到output  
# ---------------------------------------------------------#  
output\_list = []  
for i in range(3):  
 output\_list.append(self.yolo\_decodes[i](outputs[i]))  
# 在这一步将会得到三个特征层解码的输出结果  
# 输出结果的形式为#(1,3,52,52,6) (1,3,26,26,6) (1,3,13,13,6) 也就是此时每个特征层上都有三个预测框  
# ---------------------------------------------------------#  
# 将预测框进行堆叠，然后进行非极大抑制  
# 非极大抑制：筛选出区域内得分最大的框  
# 得到预测框最大的种类置信度，预测框的种类，先验框是否包含物体\*预测框的种类若大于0.5，保留  
# 得到预测结果，种类置信度，种类。  
# ---------------------------------------------------------#  
output = torch.cat(output\_list, 1)  
#print("三个特征层解码结果的堆叠：") # 也就是将这九个框堆叠到一张图像上  
#print(output)  
#print(output.size()) # （1,10647，6） 10647=（52\*52+26\*26+13\*13）\*3  
batch\_detections = non\_max\_suppression(output, len(self.class\_names),  
 conf\_thres=self.confidence,  
 nms\_thres=self.iou)

#---------------------------------------------------------#  
# 如果没有检测出物体，返回原图  
#---------------------------------------------------------#  
try:  
 batch\_detections = batch\_detections[0].cpu().numpy()  
except:  
 return image

# ---------------------------------------------------------#  
# 将进行过非极大抑制之后的输出结果分开，以便进行下一步的调整预测框  
# 只留下得分>0.5的预测框，低于0.5的候选框价值不大，丢弃  
# 坐标，置信度，种类  
# ---------------------------------------------------------#  
top\_index = batch\_detections[:, 4] \* batch\_detections[:, 5] > self.confidence  
#print("进行非极大抑制之后的top\_index")  
#print(top\_index)  
top\_conf = batch\_detections[top\_index, 4] \* batch\_detections[top\_index, 5]  
#print("进行非极大抑制之后的top\_conf")  
#print(top\_conf)  
top\_label = np.array(batch\_detections[top\_index, -1], np.int32)  
#print("进行非极大抑制之后的top\_label")  
#print(top\_label)  
  
top\_bboxes = np.array(batch\_detections[top\_index, :4])  
#print("进行非极大抑制之后的top\_bboxes")  
#print(top\_bboxes)  
top\_xmin, top\_ymin, top\_xmax, top\_ymax = np.expand\_dims(top\_bboxes[:,0],-1),np.expand\_dims(top\_bboxes[:,1],-1),np.expand\_dims(top\_bboxes[:,2],-1),np.expand\_dims(top\_bboxes[:,3],-1)

# -----------------------------------------------------------------#  
# 在图像传入网络预测前会进行letterbox\_image给图像周围添加灰条  
# 因此生成的top\_bboxes是相对于有灰条的图像的  
# 我们需要对其进行修改，去除灰条的部分。  
# -----------------------------------------------------------------#  
boxes = yolo\_correct\_boxes(top\_ymin, top\_xmin, top\_ymax, top\_xmax,  
 np.array([self.model\_image\_size[0], self.model\_image\_size[1]]), image\_shape)  
#print("去掉之前加上灰条之后的boxes")  
#print(boxes)  
#这个时候获取到的就是我们最后的预测框，所以在整个的预测过程中，我们用到的基于真实标签框产生的那九个先验框起到的是调整为对应特征层大小的框的作用  
#直白点来讲，就是13\*13的特征层我们用这种坐标来表示这些先验框

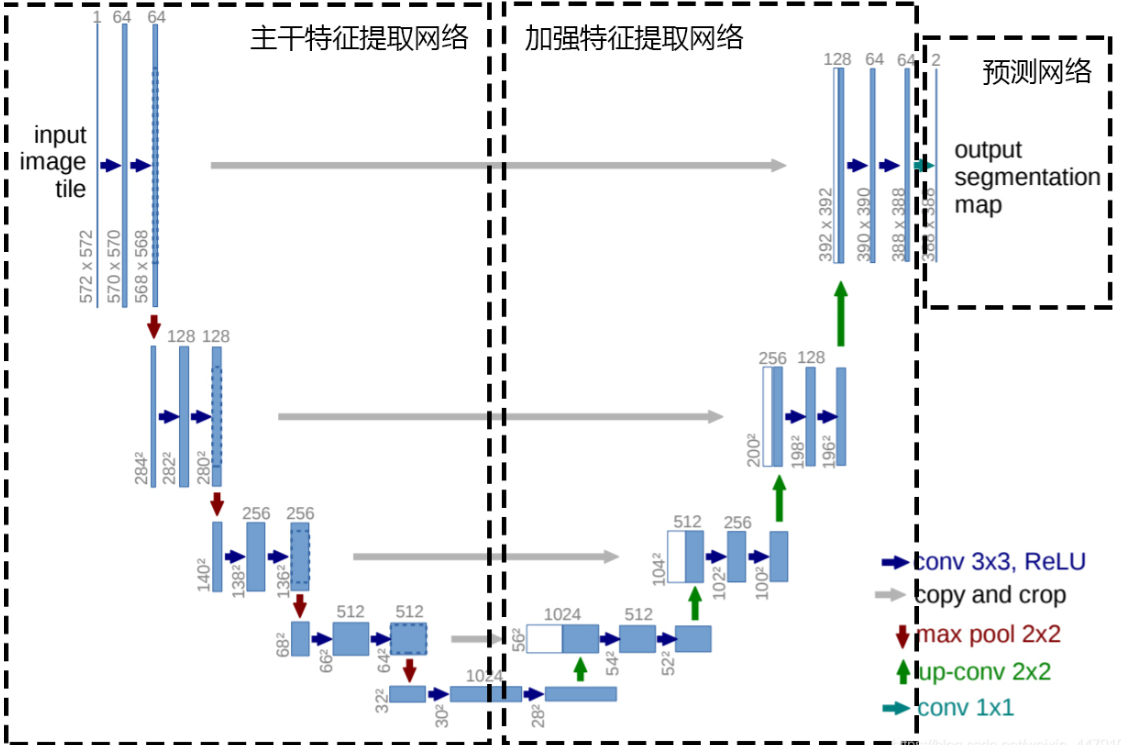
Unet网络结构

Unet可以分为三个部分，如下图所示：

第一部分是主干特征提取部分，我们可以利用主干部分获得一个又一个的特征层，Unet的主干特征提取部分与VGG相似，为卷积和最大池化的堆叠。利用主干特征提取部分我们可以获得五个初步有效特征层，在第二步中，我们会利用这五个有效特征层可以进行特征融合。

第二部分是加强特征提取部分，我们可以利用主干部分获取到的五个初步有效特征层进行上采样，并且进行特征融合，获得一个最终的，融合了所有特征的有效特征层。

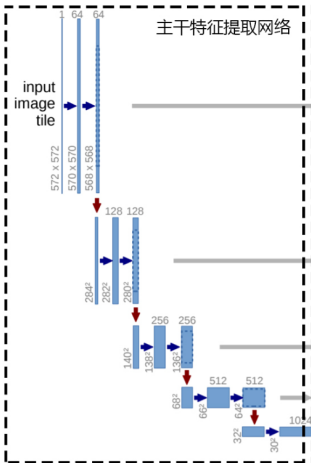
第三部分是预测部分，我们会利用最终获得的最后一个有效特征层对每一个特征点进行分类，相当于对每一个像素点进行分类。



# **Unet实现思路**

## **一、预测部分**

### **1、主干网络介绍**



Unet的主干特征提取部分由卷积+最大池化组成，整体结构与VGG类似。

我们使用的unet框架所采用的主干特征提取网络为VGG16，这样也方便使用imagnet上的预训练权重。

VGG是由Simonyan 和Zisserman在文献《Very Deep Convolutional Networks for Large Scale Image Recognition》中提出卷积神经网络模型

当我们使用VGG16作为主干特征提取网络的时候，我们只会用到两种类型的层，分别是卷积层和最大池化层。

当输入的图像大小为512x512x3的时候，具体执行方式如下：

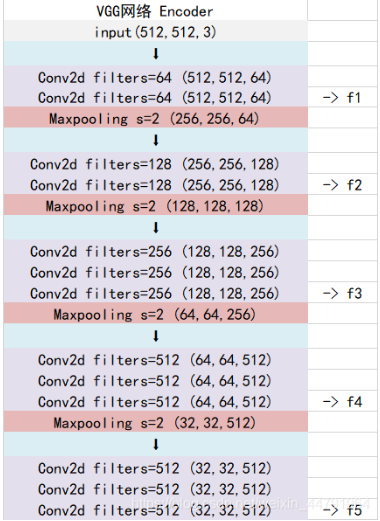
1、conv1：进行两次[3,3]的64通道的卷积，获得一个[512,512,64]的初步有效特征层，再进行2X2最大池化，获得一个[256,256,64]的特征层。

2、conv2：进行两次[3,3]的128通道的卷积，获得一个[256,256,128]的初步有效特征层，再进行2X2最大池化，获得一个[128,128,128]的特征层。

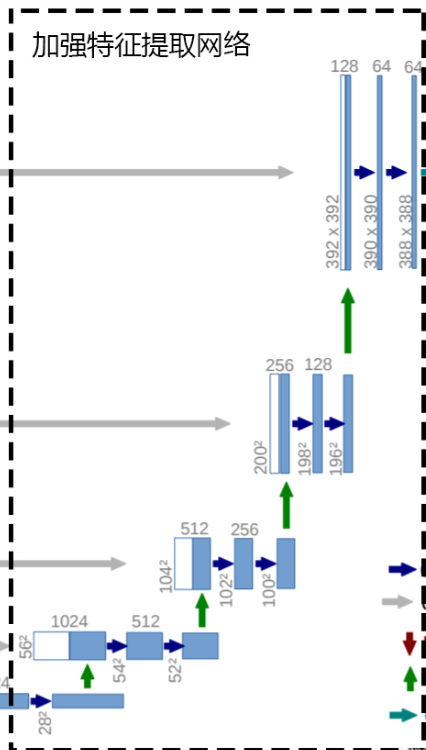
3、conv3：进行三次[3,3]的256通道的卷积，获得一个[128,128,256]的初步有效特征层，再进行2X2最大池化，获得一个[64,64,256]的特征层。

4、conv4：进行三次[3,3]的512通道的卷积，获得一个[64,64,512]的初步有效特征层，再进行2X2最大池化，获得一个[32,32,512]的特征层。

5、conv5：进行三次[3,3]的512通道的卷积，获得一个[32,32,512]的初步有效特征层。



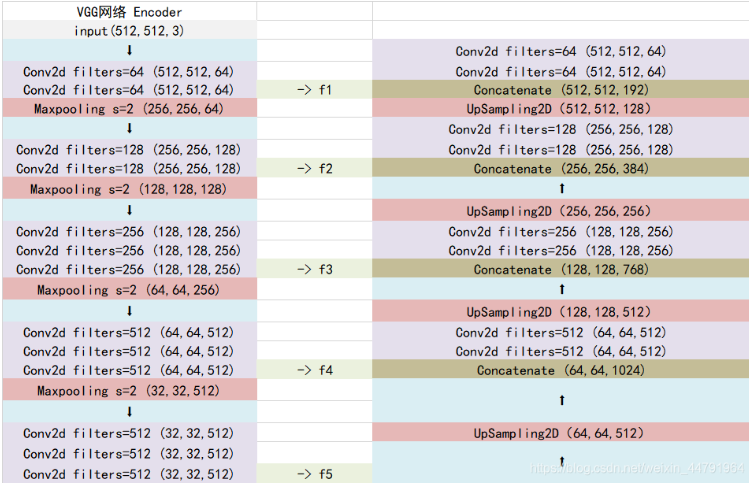
### **2、加强特征提取结构**



利用第一步我们可以获得五个初步的有效特征层，在加强特征提取网络这里，我们会利用这五个初步的有效特征层进行特征融合，特征融合的方式就是对特征层进行上采样并且进行堆叠。

为了方便网络的构建与更好的通用性，我们的Unet和上图的Unet结构有些许不同，在上采样时直接进行两倍上采样再进行特征融合，最终获得的特征层和输入图片的高宽相同。

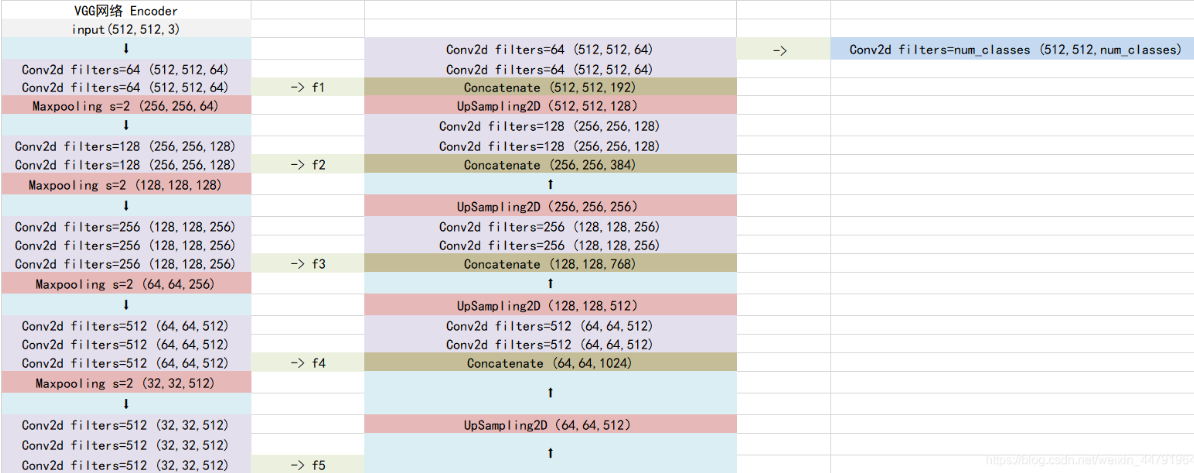
具体示意图如下：



### **3、利用特征获得预测结果**

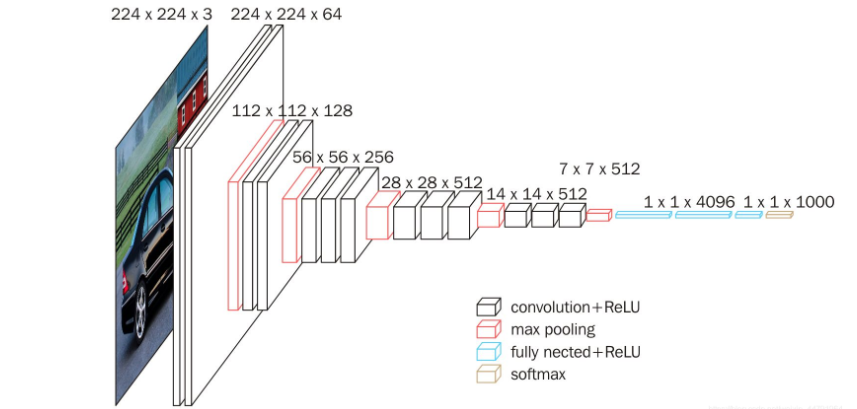
利用1、2步，我们可以获取输入进来的图片的特征，此时，我们需要利用特征获得预测结果。

利用特征获得预测结果的过程为：  
****利用一个1x1卷积进行通道调整，将最终特征层的通道数调整成num\_classes。****



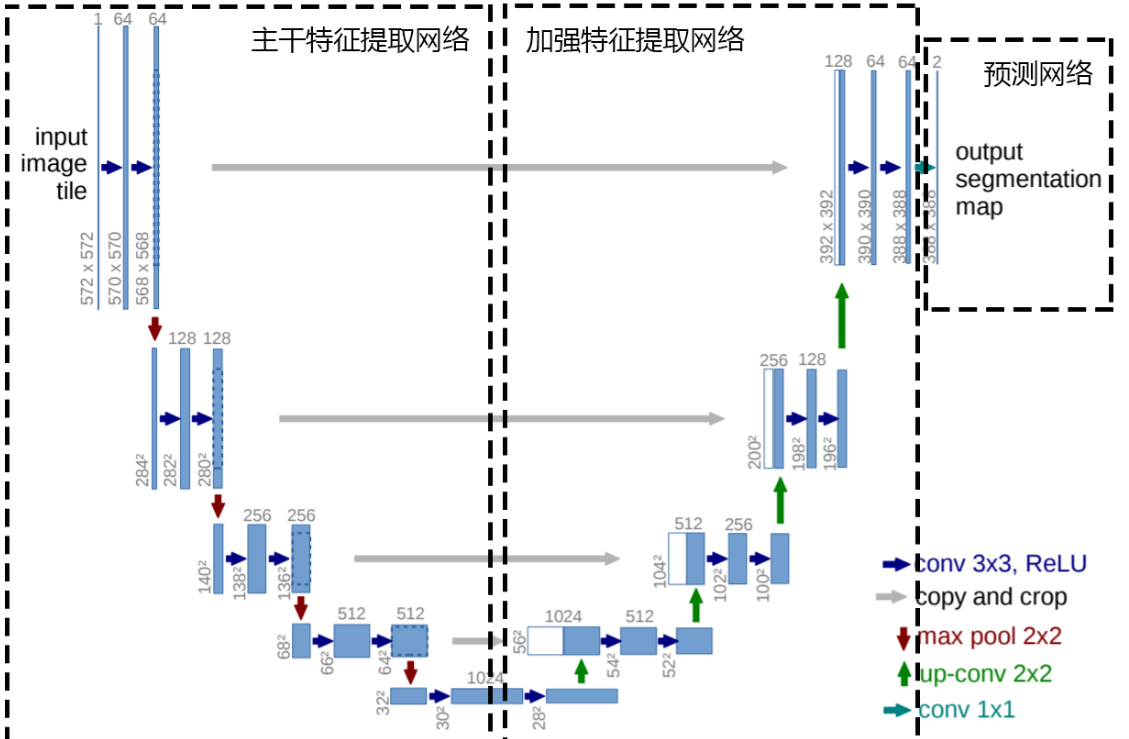
基于unet网络构建图像切割模型

1.首先使用vgg网络来作为主干提取网络

  
对输入的图片进行两次卷积（通道数为64，步长为1）（通道数为128，步长为1）+最大池化（步长为2，压缩图片0.5）+激活函数。重复两次以上步骤，然后再进行三次卷积（通道数为256、512、512，步长为1）+最大池化+激活函数，重复三次。最后对得到的特征进行全连接。利用vgg16提取网络，我们最终可以获得5个有效的特征层。

# -------------------------------#  
# 通过主干提取网络获得五个有效特征层  
# feat1 512,512,64  
# feat2 256,256,128  
# feat3 128,128,256  
# feat4 64,64,512  
# feat5 32,32,512  
# -------------------------------#  
feat1, feat2, feat3, feat4, feat5 = VGG16(inputs)

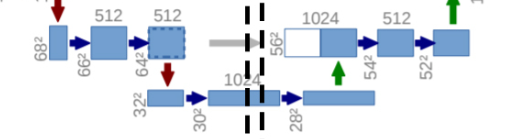
2.构建加强特征提取网络对上诉5个特征层进行特征融合



我们在利用第一步我们可以获得五个初步的有效特征层，在加强特征提取网络这里，我们会利用这五个初步的有效特征层进行特征融合，特征融合的方式就是对特征层进行上采样并且进行堆叠。

为了方便网络的构建与更好的通用性，我们的Unet和上图的Unet结构有些许不同，在上采样时直接进行两倍上采样再进行特征融合（堆叠，不进行crop），最终获得的特征层和输入图片的高宽相同，以便对输入的图片每一个像素点都能分类.

首先对最小的特征层feat5进行上采样



# 32, 32, 512 -> 64, 64, 512

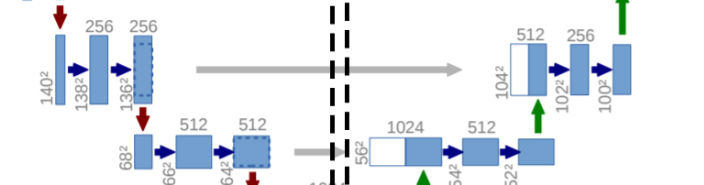
#上采样后的p5\_up宽和高就是feat5的两倍  
P5\_up = UpSampling2D(size=(2, 2))(feat5)

#p5\_up和feat4进行堆叠

#64, 64, 512 + 64, 64, 512 -> 64, 64, 1024  
P4 = Concatenate(axis=3)([feat4, P5\_up])  
# 64, 64, 1024 -> 64, 64, 512

#对堆叠的特征层p4进行两次卷积  
P4 = Conv2D(channels[3], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P4)  
P4 = Conv2D(channels[3], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P4)

继续对上诉得到的p4进行上采样和卷积



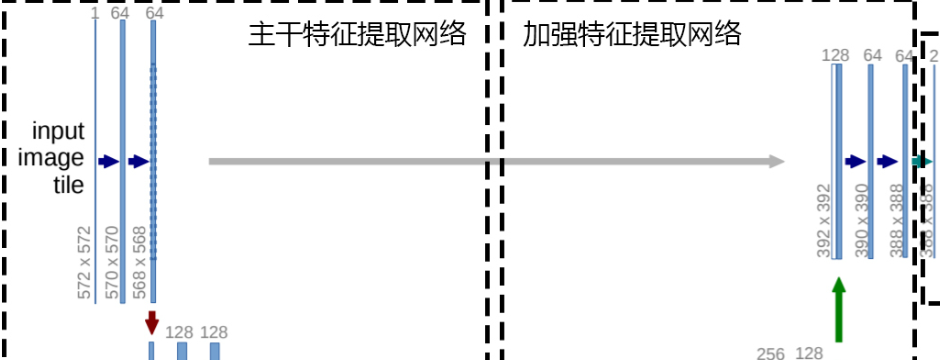
# 64, 64, 512 -> 128, 128, 512  
P4\_up = UpSampling2D(size=(2, 2))(P4)  
# 128, 128, 256 + 128, 128, 512 -> 128, 128, 768

#和feat3堆叠  
P3 = Concatenate(axis=3)([feat3, P4\_up])  
# 128, 128, 768 -> 128, 128, 256  
P3 = Conv2D(channels[2], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P3)  
P3 = Conv2D(channels[2], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P3)

对p3重复上诉步骤

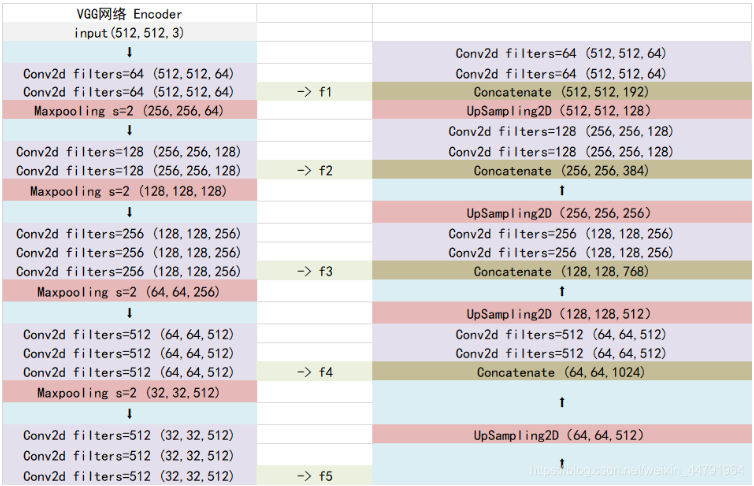


P3\_up = UpSampling2D(size=(2, 2))(P3)  
# 256, 256, 256 + 256, 256, 128 -> 256, 256, 384  
P2 = Concatenate(axis=3)([feat2, P3\_up])  
# 256, 256, 384 -> 256, 256, 128  
P2 = Conv2D(channels[1], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P2)  
P2 = Conv2D(channels[1], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P2)



P2\_up = UpSampling2D(size=(2, 2))(P2)  
# 512, 512, 128 + 512, 512, 64 -> 512, 512, 192  
P1 = Concatenate(axis=3)([feat1, P2\_up])  
# 512, 512, 128 + 512, 512, 64 -> 512, 512, 192  
P1 = Conv2D(channels[0], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P1)  
P1 = Conv2D(channels[0], 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(P1)  
# 512, 512, 64 -> 512, 512, num\_classes  
P1 = Conv2D(num\_classes, 1, activation="softmax")(P1)

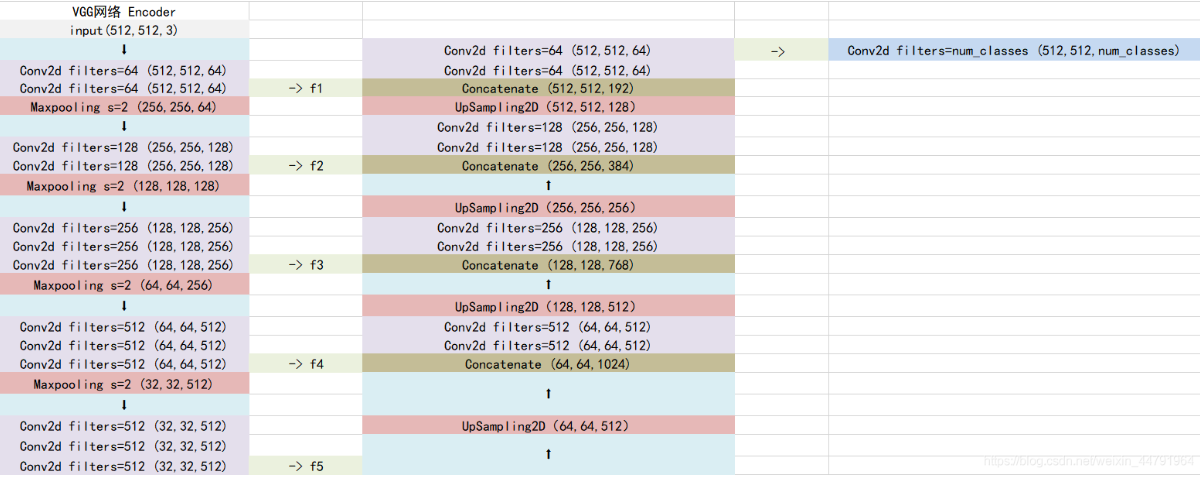
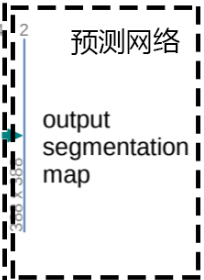
具体示意图如下：



### **3、利用特征获得预测结果**

利用1、2步，我们可以获取输入进来的图片的特征，此时，我们需要利用特征获得预测结果。

利用特征获得预测结果的过程为：  
****利用一个1x1卷积进行通道调整，将最终特征层的通道数调整成num\_classes。****



最终的预测结果shape=[512x512xnum\_classes]相当于对我们输入进来的图片每一个像素点都进行一次分类

# 512, 512, 64 -> 512, 512, num\_classes  
P1 = Conv2D(num\_classes, 1, activation="softmax")(P1)

基于上诉搭建的Unet网络进行舌体分割

首先copy图片，以便将结果绘制在图片上



将图片resize转化成特定大小

计算宽高  
orininal\_h = np.array(image).shape[0]  
orininal\_w = np.array(image).shape[1]  
#letterbox\_image：不失真的resize  
img, nw, nh = self.letterbox\_image(image, (self.model\_image\_size[1], self.model\_image\_size[0]))

将图片输入到网络中进行预测

#对图片进行预测  
pr = self.model.predict(img)[0] # Returns: Numpy array(s) of predictions.  
#选取出每一个像素点所对应最大概率的种类  
pr = pr.argmax(axis=-1).reshape([self.model\_image\_size[0], self.model\_image\_size[1]])  
#去除灰条  
pr = pr[int((self.model\_image\_size[0] - nh) // 2):int((self.model\_image\_size[0] - nh) // 2 + nh),  
 int((self.model\_image\_size[1] - nw) // 2):int((self.model\_image\_size[1] - nw) // 2 + nw)]

根据像素点的类别赋值颜色

#对像素点赋颜色，根据每一个像素点的种类赋颜色  
for c in range(self.num\_classes):  
 seg\_img[:, :, 0] += ((pr[:, :] == c) \* (self.colors[c][0])).astype('uint8')  
 seg\_img[:, :, 1] += ((pr[:, :] == c) \* (self.colors[c][1])).astype('uint8')  
 seg\_img[:, :, 2] += ((pr[:, :] == c) \* (self.colors[c][2])).astype('uint8')  
#对图片进行resize，以便和原图混合  
image = Image.fromarray(np.uint8(seg\_img)).resize((orininal\_w, orininal\_h))

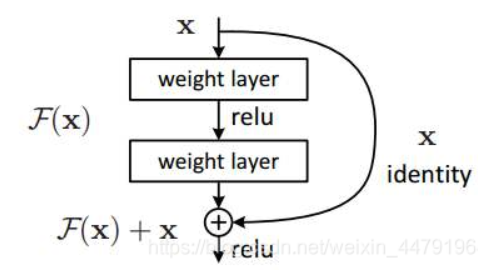


搭建resent50分类网络

## **ResNet50网络介绍**

### **a、什么是**[残差网络](https://so.csdn.net/so/search?q=%E6%AE%8B%E5%B7%AE%E7%BD%91%E7%BB%9C&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_44791964/article/details/_blank)

Residual net(残差网络)：  
****将靠前若干层的某一层数据****输出直接****跳过多层****引入到****后面数据层的输入部分。****  
意味着****后面的特征层的内容会有一部分由其前面的某一层线性贡献。****  
其结构如下：

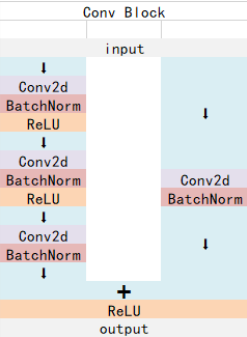


深度残差网络的设计是为了克服由于网络深度加深而产生的学习效率变低与准确率无法有效提升的问题。

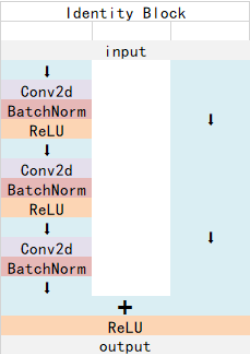
b、什么是ResNet50模型

ResNet50有两个基本的块，分别名为Conv Block和Identity Block，其中Conv Block输入和输出的维度是不一样的，所以不能连续串联，它的作用是改变网络的维度；Identity Block输入维度和输出维度相同，可以串联，它的作用是加深网络的。

Conv Block的结构如下，由图可以看出，Conv Block可以分为两个部分，左边部分为主干部分，存在两次卷积、标准化、激活函数和一次卷积、标准化；右边部分为残差边部分，存在一次卷积、标准化，由于残差边部分存在卷积，所以我们可以利用Conv Block改变输出特征层的宽高和通道数：

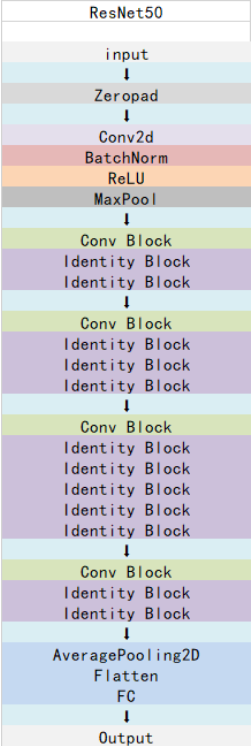


Identity Block的结构如下，由图可以看出，Identity Block可以分为两个部分，左边部分为主干部分，存在两次卷积、标准化、激活函数和一次卷积、标准化；右边部分为残差边部分，直接与输出相接，由于残差边部分不存在卷积，所以Identity Block的输入特征层和输出特征层的shape是相同的，可用于加深网络：



Conv Block和Identity Block都是残差网络结构。

总的网络结构如下：



用Resnet50构建分类神经网络

#构建神经网络  
#GlobalAveragePooling2D是平均池化，它不需要指定pool\_size和strides等参数，操作的实质是将输入特征图的每一个通道求平均得到一个数值。  
#Dropout层在训练期间的每一步中将输入单位随机设置为0，频率为速率，这有助于防止过拟合。未设置为0的输入将按1 /（1-rate）放大，以使所有输入的总和不变。  
#tf.keras.layers.Dense(神经元个数, activation="激活函数", kernel\_regularizer=哪种正则化)  
#tf.keras.layers.Dense(num\_classes)，我们有多少类就输出多少维  
#这里，因为我们希望输出 “输入图片分别属于 0 到 1的概率”，也就是一个 2 维的离散概率分布，所以我们希望这个 2 维向量至少满足两个条件：

该向量中的每个元素均在 IMG_256 之间；

该向量的所有元素之和为 1。

为了使得模型的输出能始终满足这两个条件，我们使用 [Softmax 函数](https://zh.wikipedia.org/wiki/Softmax%E5%87%BD%E6%95%B0) （归一化指数函数， tf.nn.softmax ）对模型的原始输出进行归一化。其形式为 IMG_257 。不仅如此，softmax 函数能够凸显原始向量中最大的值，并抑制远低于最大值的其他分量，这也是该函数被称作 softmax 函数的原因（即平滑化的 argmax 函数）。  
model = tf.keras.Sequential([feature,  
 tf.keras.layers.GlobalAvgPool2D(),  
 tf.keras.layers.Dropout(rate=0.5),  
 tf.keras.layers.Dense(1024, activation="relu"),  
 tf.keras.layers.Dropout(rate=0.5),  
 tf.keras.layers.Dense(num\_classes),  
 tf.keras.layers.Softmax()])

我们利用resnet50分别搭建了舌色分类网络，舌苔分析网络，裂纹舌分类网络，胖瘦舌分类网络。

由于搭建代码都相同，所以用裂纹舌分析来作为例子：

1. 首先读取unet分割后的图片并对每个像素点的像素值缩放至 （0-1），便于后续计算。

# load image  
img\_path = "user\_test/"+str(i)+".jpg"  
assert os.path.exists(img\_path), "file: '{}' dose not exist.".format(img\_path)  
img = Image.open(img\_path)  
# resize image to 224x224  
img = img.resize((im\_width, im\_height))  
plt.imshow(img)  
  
  
# 将像素值缩放至 （0-1）  
\_R\_MEAN = 123.68  
\_G\_MEAN = 116.78  
\_B\_MEAN = 103.94  
img = np.array(img).astype(np.float32)  
img = img - [\_R\_MEAN, \_G\_MEAN, \_B\_MEAN]  
  
# Add the image to a batch where it's the only member.  
img = (np.expand\_dims(img, 0))

1. 基于resnet50残差网络搭建分类神经网络

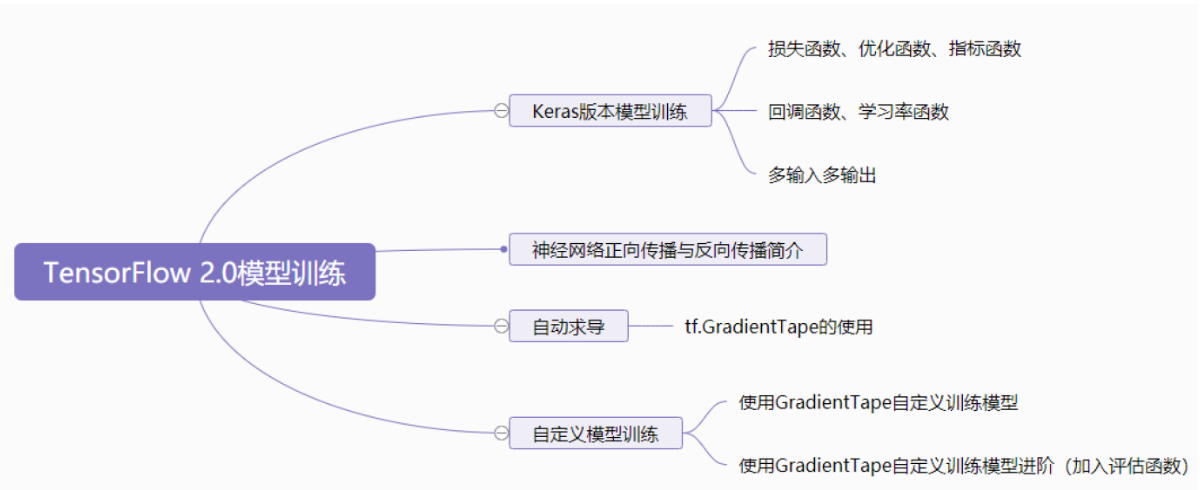
#构建神经网络  
#GlobalAveragePooling2D是平均池化，它不需要指定pool\_size和strides等参数，操作的实质是将输入特征图的每一个通道求平均得到一个数值。  
#Dropout层在训练期间的每一步中将输入单位随机设置为0，频率为速率，这有助于防止过拟合。未设置为0的输入将按1 /（1-rate）放大，以使所有输入的总和不变。  
#tf.keras.layers.Dense(神经元个数, activation="激活函数", kernel\_regularizer=哪种正则化)  
#tf.keras.layers.Dense(num\_classes)，我们有多少类就输出多少维  
#  
model = tf.keras.Sequential([feature,  
 tf.keras.layers.GlobalAvgPool2D(),  
 tf.keras.layers.Dropout(rate=0.5),  
 tf.keras.layers.Dense(1024, activation="relu"),  
 tf.keras.layers.Dropout(rate=0.5),  
 tf.keras.layers.Dense(num\_classes),  
 tf.keras.layers.Softmax()])

我们最终会得到该图片每个类别的概率参数，选取最大概率的类别作为结果返回。

# prediction  
result = np.squeeze(model.predict(img))  
predict\_class = np.argmax(result)

训练部分：

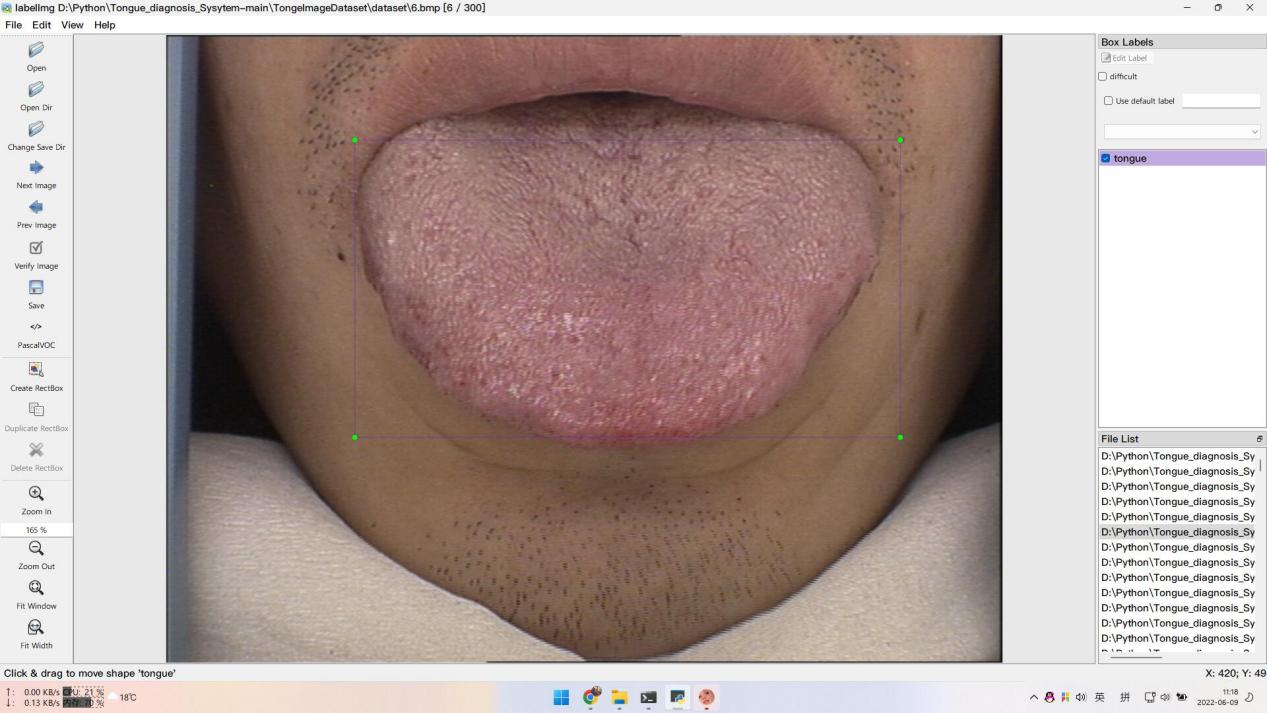
基于tensorflow2.0框架，我们分别对yolov4选择神经网络正向传播与方向传播训练，对unet使用keras版本模型训练，对基于resnet50构建的分类神经网络使用GradientTap自定义训练模型。



Yolov4训练：

1. 训练集制作

使用labelImg制作数据集，人工画出预选框



每张图片生成对应的xml文件：

<annotation>

<folder>dataset</folder>

<filename>1.bmp</filename>

<path>D:\Python\Tongue\_diagnosis\_Sysytem-main\TongeImageDataset\dataset\1.bmp</path>

<source>

<database>Unknown</database>

</source>

<size>

<width>768</width>

<height>576</height>

<depth>3</depth>

</size>

<segmented>0</segmented>

<object>

#分类的类名

<name>tongue</name>

<pose>Unspecified</pose>

<truncated>0</truncated>

<difficult>0</difficult>

#预选框位置

<bndbox>

<xmin>216</xmin>

<ymin>55</ymin>

<xmax>574</xmax>

<ymax>337</ymax>

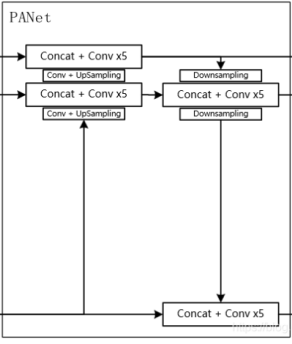
</bndbox>

</object>

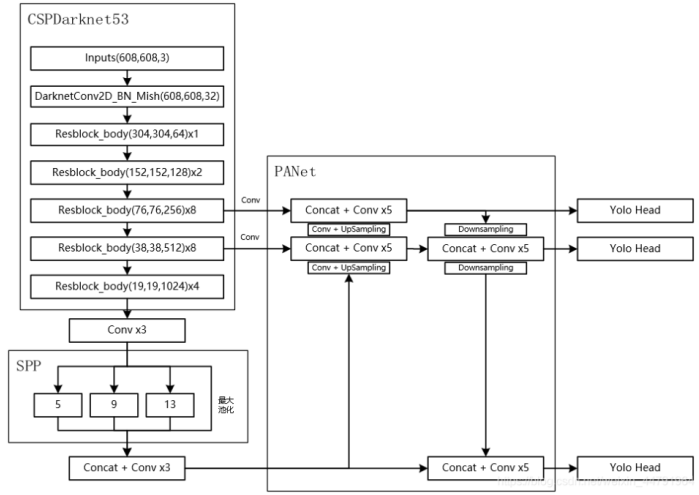
</annotation>

1. 训练

对于整个训练，我们划分成两部分，一部分冻结训练，冻结训练可以加快训练速度，也可以在训练初期防止权值被破坏。此时模型的主干被冻结了，特征提取网络不发生改变，占用的显存较小，仅对网络进行微调。



解冻训练则训练整个网络。



在实际测试中我们发现mosaic的数据增强并不稳定，在增强数据的同时带来一定的数据污染，所以默认不使用mosaic。

在深度学习中，学习率的调整非常重要。

学习率大有如下优点：

1、加快学习速率。

2、帮助跳出局部最优值。

但存在如下缺点：

1、导致模型训练不收敛。

2、单单使用大学习率容易导致模型不精确。

学习率小有如下优点：

1、帮助模型收敛，有助于模型细化。

2、提高模型精度。

但存在如下缺点：

1、无法跳出局部最优值。

2、收敛缓慢。

学习率大和学习率小的功能是几乎相反的。因此我们适当的调整学习率，才可以最大程度的提高训练性能。

在学习率的选择上，我们对比了余旋退火学习率和等间隔调整学习率发现，等间隔调整学习率效果更好。

在优化器的选择上我发现optimizer的weight\_decay起到了反作用，

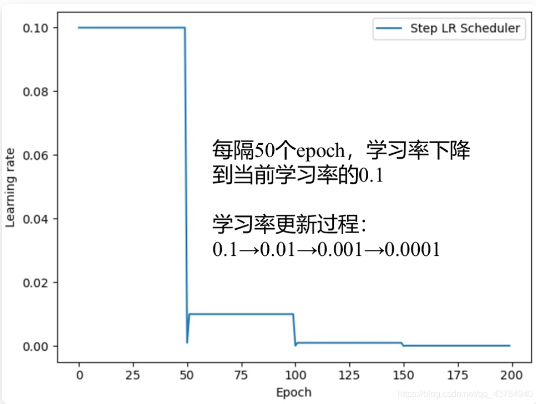
后来查阅资料，应该是数据集和weight\_decay的值设置不当导致的。  
 # 太大了实际会严重干扰第一个Learning Rate阶段的精度。太小了（也就是很多论文的默认设置）会距离收敛最优情形有差距。  
所以去除掉了weight\_decay

训练方法使用神经网络正向传播和反向传播

# ------------------------------------------------------#  
# Yolov4的tricks应用  
# mosaic 马赛克数据增强 True or False   
# 实际测试时mosaic数据增强并不稳定，所以默认为False  
# Cosine\_scheduler 余弦退火学习率 True or False  
# label\_smoothing 标签平滑 0.01以下一般 如0.01、0.005  
# ------------------------------------------------------#  
mosaic = False  
Cosine\_lr = False  
smoooth\_label = 0

#

----------------------------------------------------------------------------#  
 # 我在实际测试时，发现optimizer的weight\_decay起到了反作用，防止过拟合。  
 # 太大了实际会严重干扰第一个Learning Rate阶段的精度。太小了（也就是很多论文的默认设置）会距离收敛最优情形有差距。  
 # 所以去除掉了weight\_decay，大家也可以开起来试试，一般是weight\_decay=5e-4  
   
#parameters()会返回一个生成器（迭代器），生成器每次生成的是Tensor类型的数据．  
 # ----------------------------------------------------------------------------#

  
 optimizer = optim.Adam(net.parameters(), lr)  
 if Cosine\_lr:  
 lr\_scheduler = optim.lr\_scheduler.CosineAnnealingLR(optimizer, T\_max=5, eta\_min=1e-5)  
 else:  
 lr\_scheduler = optim.lr\_scheduler.StepLR(optimizer, step\_size=1, gamma=0.92)

2.加载yolo模型

# ------------------------------------------------------#  
# 创建yolo模型  
# 训练前一定要修改classes\_path和对应的txt文件  
# 我们加载模型参数  
# ------------------------------------------------------#  
# print(len(anchors[0])) 三个先验框  
model = YoloBody(len(anchors[0]), num\_classes)  
# print("model:"+str(model))  
  
# model\_path = "logs/Epoch71-Total\_Loss1.8202-Val\_Loss2.5562.pth"  
model\_path = "model\_data/yolo4\_voc\_weights.pth"  
print('Loading weights into state dict...')  
device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
model\_dict = model.state\_dict() # 状态字典：保存模型中的weight权值和bias偏置值  
# 在PyTorch中，state\_dict是一个从参数名称映射到参数Tesnor的字典对象。  
pretrained\_dict = torch.load(model\_path, map\_location=device) # 加载模型的权重文件  
pretrained\_dict = {k: v for k, v in pretrained\_dict.items() if np.shape(model\_dict[k]) == np.shape(v)}  
model\_dict.update(pretrained\_dict)  
model.load\_state\_dict(model\_dict)  
print('Finished!')

1. 建立损失函数

YOLOLoss：我们的损失实际上包含三个部分，存在物体：**box\_loss**,计算方法有两种，一种使用giou进行计算，另一种使用原始的MSE和BCE进行计算，这个部分我们将融合我们的位置损失权重box\_loss\_scale；**cls\_loss**:使用BCE对分类标签进行计算损失；**conf\_loss**:置信度损失，使用BCE进行计算即可；不存在物体：只有conf\_loss进行计算；最终我们得总体的loss就是上述的各种loss和对应权重之间乘积的加和。

# 建立loss函数，mask：anchors\_mask 用于帮助代码找到对应的先验框，一般不修改。  
yolo\_losses = []  
for i in range(3): # 对于每一个特征层都计算其loss值  
 yolo\_losses.append(YOLOLoss(np.reshape(anchors, [-1, 2]), num\_classes, \  
 (input\_shape[1], input\_shape[0]), smoooth\_label, Cuda, normalize))

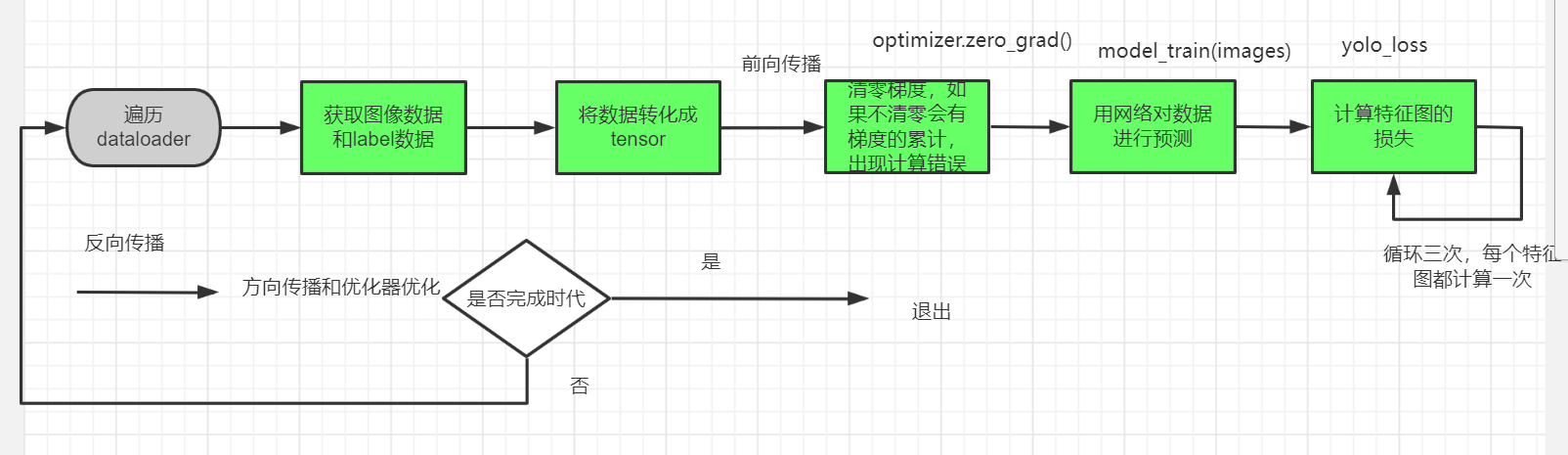
1. 创建训练集和测试集

# 读取txt文件主要是为了构建dataset和dataloader  
val\_split = 0.1  
with open(annotation\_path) as f:  
 lines = f.readlines()  
np.random.seed(10101)  
np.random.shuffle(lines)  
np.random.seed(None)  
num\_val = int(len(lines) \* val\_split)  
num\_train = len(lines) - num\_val

train\_dataset = YoloDataset(lines[:num\_train], (input\_shape[0], input\_shape[1]), mosaic=mosaic,  
 is\_train=True)  
val\_dataset = YoloDataset(lines[num\_train:], (input\_shape[0], input\_shape[1]), mosaic=False, is\_train=False)  
gen = DataLoader(train\_dataset, shuffle=True, batch\_size=Batch\_size, num\_workers=4, pin\_memory=True,  
 drop\_last=True, collate\_fn=yolo\_dataset\_collate)  
gen\_val = DataLoader(val\_dataset, shuffle=True, batch\_size=Batch\_size, num\_workers=4, pin\_memory=True,  
 drop\_last=True, collate\_fn=yolo\_dataset\_collate)

1. 使用基于神经网络正向传播和反向传播的fit\_one\_epoch训练模型

Fit\_one\_epoch流程图



#训练的世代大小  
epoch\_size = max(1, num\_train // Batch\_size)  
epoch\_size\_val = num\_val // Batch\_size  
# ------------------------------------#  
# 冻结一定部分训练  
# 进行冻结训练的话，看下面的循环，我们是针对backbone也就是cspdarknet53的参数，将他们都设置成不需要梯度更新的状态  
# ------------------------------------#  
for param in model.backbone.parameters():  
 param.requires\_grad = False  
## 因为我们的非冻结Epoch就是要训练的轮次，所以直接循环从初始epoch到非冻结epoch  
for epoch in range(Init\_Epoch, Freeze\_Epoch):  
 ## 真正的训练方法(核心)  
 fit\_one\_epoch(net, yolo\_losses, epoch, epoch\_size, epoch\_size\_val, gen, gen\_val, Freeze\_Epoch, Cuda)  
 #每训练一次学习率都调整一次  
 lr\_scheduler.step()

1. 解冻训练

# ------------------------------------#  
# 解冻后训练  
# ------------------------------------#  
for param in model.backbone.parameters():  
 param.requires\_grad = True  
  
for epoch in range(Freeze\_Epoch, Unfreeze\_Epoch):  
 fit\_one\_epoch(net, yolo\_losses, epoch, epoch\_size, epoch\_size\_val, gen, gen\_val, Unfreeze\_Epoch, Cuda)  
 lr\_scheduler.step()

Unet训练：

因为unet网络结构简单，对小数据集支持度好，所以我们使用ReduceLROnPlateau和EarlyStopping配合使用来调整学习率，学习率减半下降，acc三次不下降就下降学习率继续训练

跟yolov4一样，我们同样把unet训练分为冻结训练和解冻训练两个部分。

在损失函数的选取上，我们使用交叉熵来作为损失函数。

查阅资料对比之后我们选择keras版本训练模型中的fir\_generator来训练模型

因为对于小型，简单化的数据集，使用Keras的.fit函数是完全可以接受的。这些数据集通常不是很具有挑战性，不需要任何数据增强。

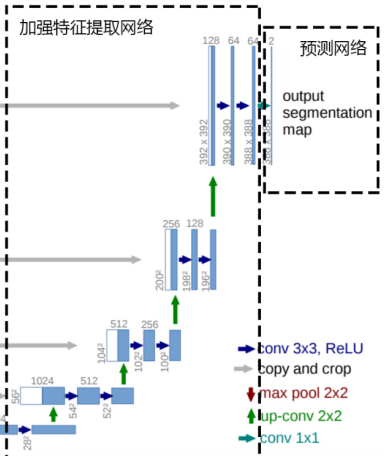
1. 建立unet模型

# 获取model  
model = Unet(inputs\_size,num\_classes)  
model.summary()  
#model\_path = "./model\_data/unet\_medical.h5"  
model\_path = "./logs/ep016-loss0.065-val\_loss0.072.h5"  
model.load\_weights(model\_path, by\_name=True, skip\_mismatch=True)  
  
# 打开数据集的txt，读取所有行  
with open(r"VOCdevkit/VOC2007/ImageSets/Segmentation/train.txt","r") as f:  
 train\_lines = f.readlines()  
  
# 打开数据集的txt  
with open(r"VOCdevkit/VOC2007/ImageSets/Segmentation/val.txt","r") as f:  
 val\_lines = f.readlines()

1. 选择学习率函数

# -------------------------------------------------------------------------------#  
# 训练参数的设置  
# logging表示tensorboard的保存地址  
# checkpoint用于设置权值保存的细节，period用于修改多少epoch保存一次  
# reduce\_lr用于设置学习率下降的方式  
# early\_stopping用于设定早停，val\_loss多次不下降自动结束训练，表示模型基本收敛  
# -------------------------------------------------------------------------------#  
# 保存的方式，1世代保存一次  
checkpoint\_period = ModelCheckpoint(log\_dir + 'ep{epoch:03d}-loss{loss:.3f}-val\_loss{val\_loss:.3f}.h5',  
 monitor='val\_loss', save\_weights\_only=True, save\_best\_only=False, period=1)  
# 学习率下降的方式，acc三次不下降就下降学习率继续训练  
reduce\_lr = ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=3, verbose=1)  
# 是否需要早停，当val\_loss一直不下降的时候意味着模型基本训练完毕，可以停止  
early\_stopping = EarlyStopping(monitor='val\_loss', min\_delta=0, patience=12, verbose=1)

1. 使用fit\_generator训练模型冻结训练加强特征提取网络



LOSS由两部分组成：

1、Cross Entropy Loss。

2、Dice Loss。

Cross Entropy Loss就是普通的交叉熵损失，当语义分割平台利用Softmax对像素点进行分类的时候，进行使用。

Dice loss将语义分割的评价指标作为Loss，Dice系数是一种集合相似度度量函数，通常用于计算两个样本的相似度，取值范围在[0,1]。

计算公式如下：

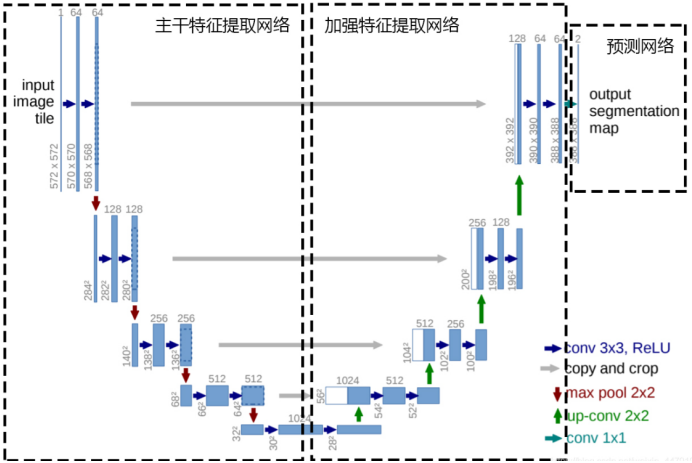


就是预测结果和真实结果的交乘上2，除上预测结果加上真实结果。其值在0-1之间。越大表示预测结果和真实结果重合度越大。所以Dice系数是越大越好。

如果作为LOSS的话是越小越好，所以使得Dice loss = 1 - Dice，就可以将Loss作为语义分割的损失了。

# ------------------------------------------------------#  
# 主干特征提取网络特征通用，冻结训练可以加快训练速度  
# 也可以在训练初期防止权值被破坏。此时模型的主干被冻结了，特征提取网络不发生改变，占用的显存较小，仅对网络进行微调  
# Init\_Epoch为起始世代  
# Freeze\_Epoch为冻结训练的世代  
# UnFreeze\_Epoch总训练世代  
# 提示OOM或者显存不足请调小Batch\_size  
# ------------------------------------------------------#  
if True:  
 lr = 1e-4  
 Init\_Epoch = 0  
 Freeze\_Epoch = 20  
 Batch\_size = 4  
 # 交叉熵loss  
 #当语义分割平台利用Softmax对像素点进行分类的时候，进行使用。  
  
 model.compile(loss = dice\_loss\_with\_CE() if dice\_loss else CE(),  
 optimizer = Adam(lr=lr),  
 metrics = [f\_score()])  
 print('Train on {} samples, val on {} samples, with batch size {}.'.format(len(train\_lines), len(val\_lines), Batch\_size))  
  
 gen = Generator(Batch\_size, train\_lines, inputs\_size, num\_classes).generate()  
 gen\_val = Generator(Batch\_size, val\_lines, inputs\_size, num\_classes).generate(False)  
 # 开始训练  
 #对于小型，简单化的数据集，使用Keras的.fit函数是完全可以接受的。这些数据集通常不是很具有挑战性，不需要任何数据增强。  
  
 model.fit\_generator(gen,  
 steps\_per\_epoch=max(1, len(train\_lines)//Batch\_size),  
 validation\_data=gen\_val,  
 validation\_steps=max(1, len(val\_lines)//Batch\_size),  
 epochs=20,  
 initial\_epoch=0,  
 callbacks=[checkpoint\_period, reduce\_lr,tensorboard])

1. 解冻训练训练整个unet网络



# ----------------------------------------------------#  
# 解冻阶段训练参数  
# 此时模型的主干不被冻结了，特征提取网络会发生改变  
# 占用的显存较大，网络所有的参数都会发生改变  
# ----------------------------------------------------#  
if True:  
 lr = 1e-5  
 Freeze\_Epoch = 20  
 Unfreeze\_Epoch = 50  
 Batch\_size = 4  
 # 交叉熵  
 model.compile(loss = dice\_loss\_with\_CE() if dice\_loss else CE(),  
 optimizer = Adam(lr=lr),  
 metrics = [f\_score()])  
 print('Train on {} samples, val on {} samples, with batch size {}.'.format(len(train\_lines), len(val\_lines), Batch\_size))  
  
 gen = Generator(Batch\_size, train\_lines, inputs\_size, num\_classes).generate()  
 gen\_val = Generator(Batch\_size, val\_lines, inputs\_size, num\_classes).generate(False)  
 # 开始训练  
 model.fit\_generator(gen,  
 steps\_per\_epoch=max(1, len(train\_lines)//Batch\_size),  
 validation\_data=gen\_val,  
 validation\_steps=max(1, len(val\_lines)//Batch\_size),  
 epochs=20,  
 initial\_epoch=20,  
 callbacks=[checkpoint\_period, reduce\_lr,tensorboard])

Resnet训练：

因为我们的分类神经网络是用softmax做标签分类。所以在损失函数的选择上我们使用适用于多分类问题的CategoricalCrossentropy交叉熵损失函数作为loss function， 输出结果的metrics.Mean加权平均值为loss值。metrics.CategoricalAccuracy计算预测与真实值的准确度作为评估指标

训练方法上不同于上面的冻结解冻训练，而是直接使用梯度下降训练方法。

1. 建立损失函数

#这部分是建立损失函数  
#使用适用于多分类问题的CategoricalCrossentropy交叉熵损失函数作为loss function，  
# 输出结果的metrics.Mean加权平均值为loss值。metrics.CategoricalAccuracy计算预测与真实值的准确的来作为评估指标  
loss\_object = tf.keras.losses.CategoricalCrossentropy(from\_logits=False)  
optimizer = tf.keras.optimizers.Adam(learning\_rate=0.0002)  
#计算给定值的（加权）平均值。作为loss值  
train\_loss = tf.keras.metrics.Mean(name='train\_loss')

#计算预测与真实值的准确度。作为评估指标  
train\_accuracy = tf.keras.metrics.CategoricalAccuracy(name='train\_accuracy')  
#计算给定值的（加权）平均值。  
val\_loss = tf.keras.metrics.Mean(name='val\_loss')  
#计算预测与真实值的准确度。  
val\_accuracy = tf.keras.metrics.CategoricalAccuracy(name='val\_accuracy')

1. 训练集的训练step

#训练的step（step和世代类似但不一样，可以这样理解多个step组成一个世代）  
def train\_step(images, labels):  
  
 with tf.GradientTape() as tape:  
 # ouput是预测结果  
 output = model(images, training=True)  
 # 计算真实结果和预测结果的交叉熵损失  
 loss = loss\_object(labels, output)  
 #"""计算梯度，返回的梯度与 sources 同结构"""loss：梯度求导的因变量，model.trainable\_variables梯度求导的自变量  
 gradients = tape.gradient(loss, model.trainable\_variables)  
 #将处理后的梯度更新到模型权重  
 optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))  
 # 计算loss的（加权）平均值。  
 train\_loss(loss)  
 ##计算预测与真实值的准确度。  
 train\_accuracy(labels, output)

测试机的step

#记录每一step的loss值  
def val\_step(images, labels):  
 #ouput是预测结果  
 output = model(images, training=False)  
 #计算真实结果和预测结果的交叉熵损失  
 loss = loss\_object(labels, output)  
 #计算loss的（加权）平均值。  
 val\_loss(loss)  
 ##计算预测与真实值的准确度。  
 val\_accuracy(labels, output)

1. 训练主干函数，由step函数构成

for epoch in range(epochs):  
 #清空梯度  
 train\_loss.reset\_states() # clear history info  
 train\_accuracy.reset\_states() # clear history info  
 val\_loss.reset\_states() # clear history info  
 val\_accuracy.reset\_states() # clear history info  
  
 # JPEGImages  
 #一个时代经历train\_bar个step的训练，这是在训练集上训练  
 train\_bar = tqdm(range(total\_train // batch\_size))  
 for step in train\_bar:  
 images, labels = next(train\_data\_gen)  
 train\_step(images, labels)  
  
 # print JPEGImages process  
 train\_bar.desc = "train epoch[{}/{}] loss:{:.3f}, acc:{:.3f}".format(epoch + 1,  
 epochs,  
 train\_loss.result(),  
 train\_accuracy.result())  
  
 # validate与上面一样，在测试集上训练  
 val\_bar = tqdm(range(total\_val // batch\_size), colour='green')  
 for step in val\_bar:  
 test\_images, test\_labels = next(val\_data\_gen)  
 val\_step(test\_images, test\_labels)  
  
 # print val process  
 val\_bar.desc = "valid epoch[{}/{}] loss:{:.3f}, acc:{:.3f}".format(epoch + 1,  
 epochs,  
 val\_loss.result(), val\_accuracy.result())  
 #如果当前训练结果比之前的更好，保存当前世代的权重文件  
 if val\_loss.result() < best\_test\_loss:  
 best\_test\_loss = val\_loss.result()  
 model.save\_weights("./save\_weights\_crack/resNet\_50.ckpt", save\_format="tf")