# 基于 YOLO_V3 的火焰检测系统

目录

# 1. 研究背景与意义

物体检测一直是计算机视觉领域经久不衰的研究方向。物体检测同样是一个主观的过程，对于人类来说相当简单。就连一个没受过任何训练的孩子通过观察图片中不同的颜色、区域等特征就能轻易定位出目标物体。但计算机收到这些RGB 像素矩阵，不会直接得到目标（如行人、车辆等）的抽象概念，更不必说定位其位置了。再加上目标形态千差万别，目标和背景重合等问题，使得目标检测难上加难。

传统的目标检测算法包括三个阶段，首先生成目标建议框，接着提取每个建议框中的特征，最后根据特征进行分类。以下是这三个阶段的具体过程：

1，生成目标建议框。当输入一张原始图片时，计算机只认识每一个像素点，想要用方框框出目标的位置以及大小，最先想到的方法就是穷举建议框，具体的做法就是用滑动窗口扫描整个图像，还要通过缩放来进行多尺度滑窗。很显然这种方法计算量很大，很多都是重复的计算，并且效率极低。

2，提取每个建议框中的特征。在传统的检测中，常见的 HOG[20]算法对物体边缘使用直方图统计来进行编码，有较好的表达能力。然而传统特征设计需要人工指定，达不到可靠性的要求。

3，分类器的设计。传统的分类器在机器学习领域非常多。具有代表性的 SVM 将分类间隔最大化来获得分类平面的支持向量，在指定特征的数据集上表现良好。

后来人们提出一种新的物体检测方法--YOLO。YOLO 之前的物体检测方法主要是通过区域监测产生大量的可能包含待检测物体的潜在预测框，再用分类器去判断每个边界框里是否包含有物体，以及物体所属类别的 可能性或者 置信度，如 R-CNN,Fast-R-CNN,Faster-R-CNN 等。YOLO 不同于这些物体检测方法，它将物体检测任务当做一个回归问题来处理，使用一个神经网络，直接从一整张图像来预测出边界框的坐标、框中包含物体的置信度和物体的可能性。

我们的课题做了关于图片火焰检测，或者视频火焰检测，或者摄像头实时检测火焰。本课题有十分重要的现实意义，在日常生活中火灾给人们带来的灾难和损失都是难以想象的，我们的工作从搜集不同大小的火焰，例如蜡烛的火焰，打火机的火焰到火灾的火焰，不同尺度的火焰，然后人工进行标记火焰的位置，这个工作量是很大的，但也是必要的，因为只要高质量的输入数据才能让模型有较好的泛华性。最后我们完成的模型检测一张图片仅需要 0.001s~0.004s,就是说在 100fps 以下的 1080P 的视频或者摄像头的画面数据都能准确且实时地识别，能够及时发现火焰,很大程度地减少火灾的发生,能够挽救无数生命和财产损失。有极强的现实意义。

# 2. 功能与指标

1. 能够准检测出别给定图片中的火焰及其数量。
2. 能够准确检测出火焰所在的位置并能够准确框出物体并输出。

3. Flask 制作网页前端，能够有更好的交互性，界面友好。
4. 能够在视频中准确地检测火焰及其数量。
5. 能够在视频中能够准确检测出火焰所在的位置并能够准确框出物体并输出。
6. 要求算法和系统能够足够快，能够实时处理分析视频或者摄像头传来的数据。

# 3. 算法原理

## 3.1 YOLO_v3 网络结构

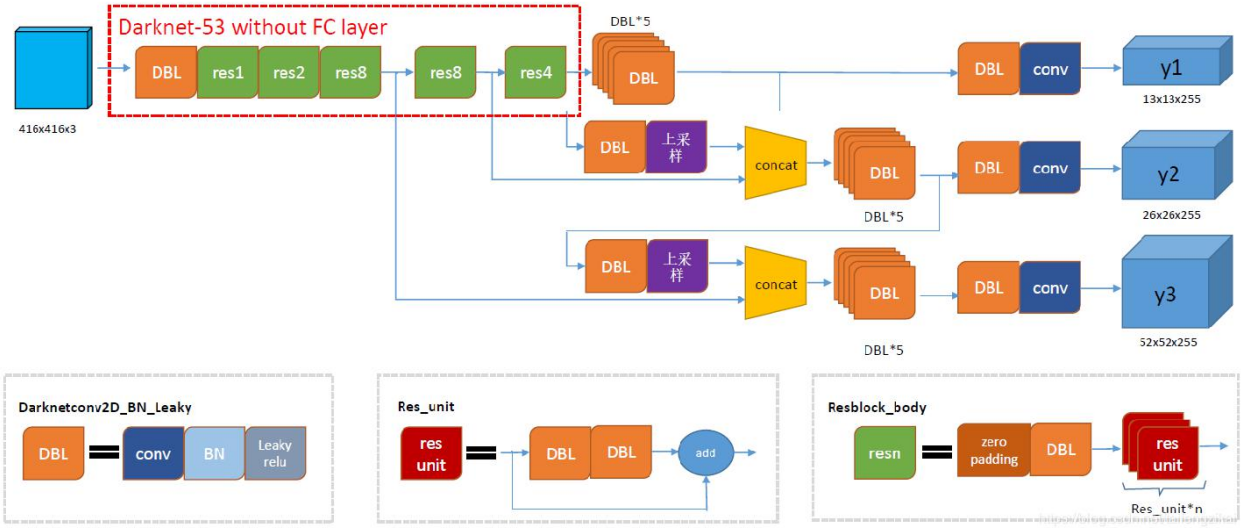



图 1 Yolo v3 结构图

图 1 中的 DBL 是 Yolo v3 的基本组件。yolo3.model 中的 DarknetConv2D_BN_Leaky 函数所定义的那样，Darknet 的卷积层后接 BatchNormalization（BN）和 LeakyReLU。除最后一层卷积层外，在 yolo v3 中 BN 和 LeakyReLU 已经是卷积层不可分离的部分了，共同构成了最小组件。

主干网络中使用了 5 个 resn 结构。n 代表数字，有 res1，res2，…，res8 等等，表示这个 res_block 里含有 n 个 res_unit，这是 Yolo v3 的大组件。从 Yolo v2 的 darknet-19 上升到 Yolo v3 的 darknet-53，前者没有残差结构。Yolo v3 开始借鉴了 ResNet 的残差结构，使用这种结构可以让网络结构更深。对于 res_block 的解释，可以在图 1.1 的右下角直观看到，其基本组件也是 DBL。

在预测支路上有张量拼接（concat）操作。其实现方法是将 darknet 中间层和中间层后某一层的上采样进行拼接。值得注意的是，张量拼接和 Res_unit 结构的 add 的操作是不一样的，张量拼接会扩充张量的维度，而 add 只是直接相加不会导致张量维度的改变。

## 3.2 Darknet-53 特征提取网络

Yolo v3 中使用了一个 53 层的卷积网络，这个网络由残差单元叠加而成。Joseph Redmon 的实验表明，在分类准确度上与效率的平衡上，Darknet-53 模型比 ResNet-101、 ResNet-152 和 Darknet-19 表现得更好。Yolo v3 并没有那么追求速度，而是在保证实时性(fps>60)的基础上追求更好的表现。

一方面，Darknet-53 网络采用全卷积结构，Yolo v3 前向传播过程中，张量的尺寸变换是通过改变卷积核的步长来实现的。卷积的步长为 2，每次经过卷积之后，图像边长缩小一半。如图 2.1 中所示，Darknet-53 中有 5 次卷积的步长为 2。经过 5 次缩小，特征图缩小为原输入尺寸的 1/32。所以网络输入图片的尺寸为 32 的倍数，取为 416×416。Yolo v2 中对于前向过程中张量尺寸变换，都是通过最大池化来进行，一共有 5 次。而 v3 是通过卷积核增大步长来进行，也是 5 次。

另一方面，Darknet-53 网络引入了 residual 结构。Yolo v2 中还是类似 VGG 那样直筒型的网络结构，层数太多训起来会有梯度问题，所以 Darknet-19 也就 19 层。得益于 ResNet 的 residual 结构，训练深层网络的难度大大减小。因此 Darknet-53 网络做到 53 层，精度提升比较明显。

### 3.3 利用多尺度特征进行对象检测

在 YOLO3 更进一步采用了 3 个不同尺度的特征图来进行对象检测。

结合下图看，卷积网络在 79 层后，经过下方几个黄色的卷积层得到一种尺度的检测结果。相比输入图像，这里用于检测的特征图有 32 倍的下采样。比如输入是 416*416 的话，这里的特征图就是 13*13 了。由于下采样倍数高，这里特征图的感受野比较大，因此适合检测图像

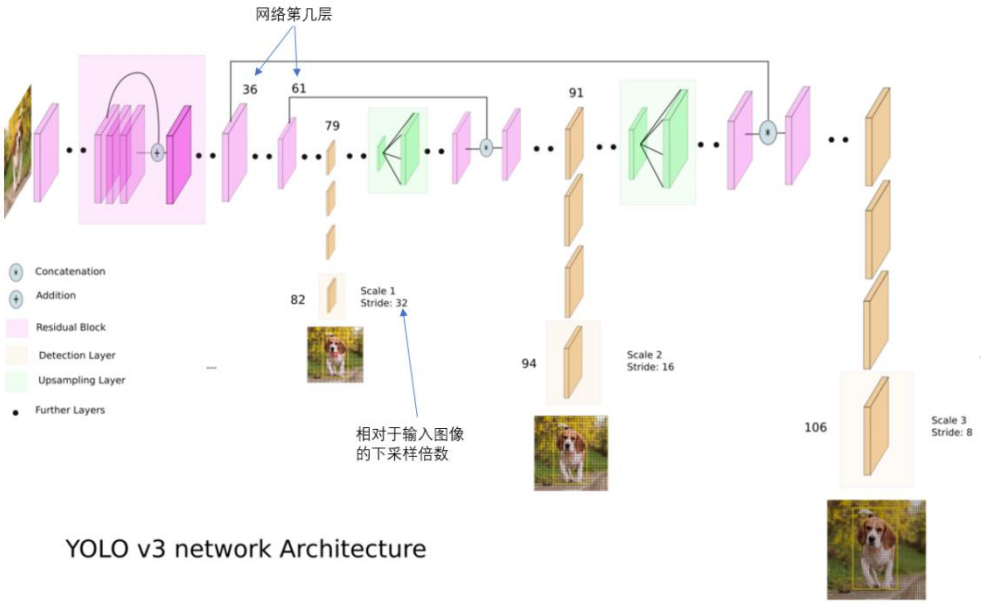



图 2. 多尺度预测结构

为了实现细粒度的检测，第 79 层的特征图又开始作上采样（从 79 层往右开

始上采样卷积），然后与第 61 层特征图融合（Concatenation），这样得到第 91 层较细粒度的特征图，同样经过几个卷积层后得到相对输入图像 16 倍下采样的特征图。它具有中等尺度的感受野，适合检测中等尺度的对象。

最后，第 91 层特征图再次上采样，并与第 36 层特征图融合（Concatenation），最后得到相对输入图像 8 倍下采样的特征图。它的感受野最小，适合检测小尺寸的对象。

## 3.4 多种尺度的先验框

随着输出的特征图的数量和尺度的变化，先验框的尺寸也需要相应的调整。YOLO2 已经开始采用 K-means 聚类得到先验框的尺寸，YOLO3 延续了这种方法，为每种下采样尺度设定 3 种先验框，总共聚类出 9 种尺寸的先验框。在 COCO 数据集这 9 个先验框是：(10x13)，(16x30)，(33x23)，(30x61)，(62x45)，(59x119)，(116x90)，(156x198)，(373x326)。

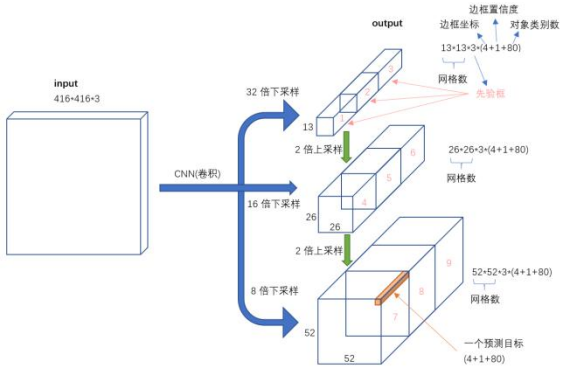
分配上，在最小的 13*13 特征图上（有最大的感受野）应用较大的先验框 (116x90)，(156x198)，(373x326)，适合检测较大的对象。中等的 26*26 特征图上（中等感受野）应用中等的先验框(30x61)，(62x45)，(59x119)，适合检测中等大小的对象。较大的 52*52 特征图上（较小的感受野）应用较小的先验框 (10x13)，(16x30)，(33x23)，适合检测较小的对象。

## 3.5 输入映射到输出

不考虑神经网络结构细节的话，总的来说，对于一个输入图像，YOLO3 将其映射到 3 个尺度的输出张量，代表图像各个位置存在各种对象的概率。

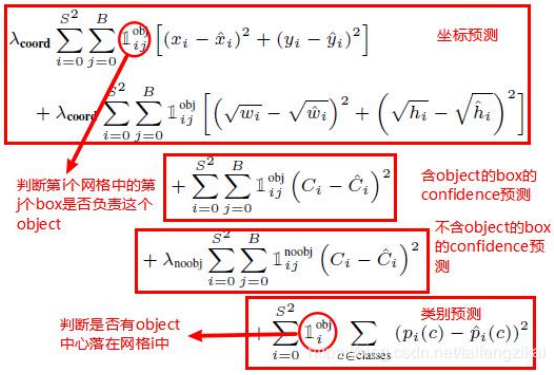
我们看一下 YOLO3 共进行了多少个预测。对于一个 416*416 的输入图像，在每个尺度的特征图的每个网格设置 3 个先验框，总共有 13*13*3 + 26*26*3 + 52*52*3 = 10647 个预测。每一个预测是一个(4+1+80)=85 维向量，这个 85 维向量包含边框坐标（4 个数值），边框置信度（1 个数值），对象类别的概率（对于 COCO 数据集，有 80 种对象（我们的进行迁移学习，所以最后只有一种火焰类别））。

YOLO3 的尝试预测边框数量增加了 10 多倍，而且是在不同分辨率上进行，所以 mAP 以及对小物体的检测效果有一定的提升。




# 3.6 损失函数

在目标检测任务里，有几个关键信息是需要确定的:(x,y),(w,h),class,confidence 。根据关键信息的特点可以分为上述四类，损失函数应该由各自特点确定。最后加到一起就可以组成最终的 loss function 了，也就是一个 loss function 搞定端到端的训练。




Yolo 损失函数

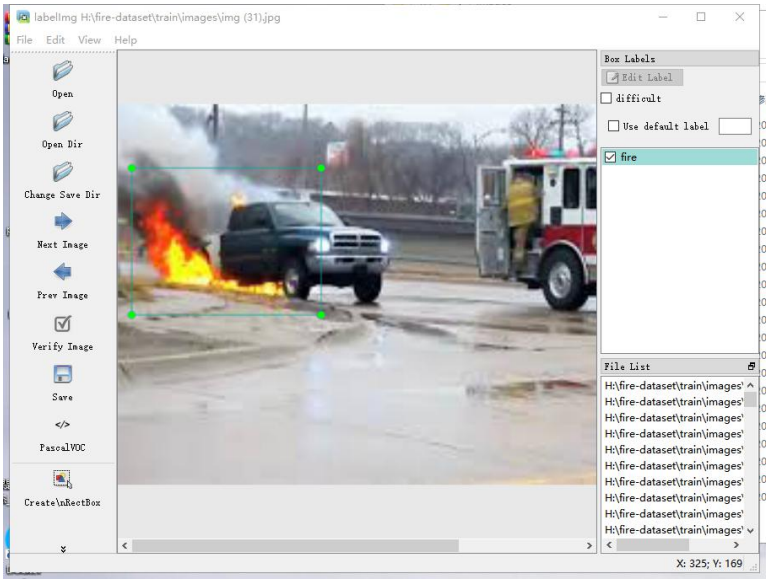
# 3.7 迁移学习

目标识别从头开始训练模型的权重代价是非常昂贵的。所以我们使用了迁移学习，用 yolo3 预训练好的权重，在构建我们自己的网络的时候冻结前面的网络，改变最后几层全连接层使其符合我们自己的火焰类别的识别。

# 4. 实现过程

## 4.1 网络爬取数据和标注数据

我们共搜集和爬取数据 502 张，其中 412 张用于训练，剩余 90 张用于交叉验证模型的好坏。使用的 Scrapy 框架爬取，局限于篇幅不再赘述爬虫过程（也是相当繁琐）。接下来要标记数据，使用的 Labelimg 工具进行标注（Github 上可以下载）




如图所示对每一张图片进行标注，格式为 PascalVOC 格式。标记完所有数据后，每个图片的标记会以 xml 文件保存，其中包括了物体类别（fire），检测框的左上角和右下角的像素坐标。

```
- <object>
    <name>fire</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
  - <bndbox>
      <xmin>82</xmin>
      <ymin>4</ymin>
      <xmax>121</xmax>
      <ymax>36</ymax>
    </bndbox>
  </object>
- <object>
```

经过 voc_label.py 后面处理 xml 文件里的信息能够被提取出来，并存储为模型可以直接使用的 label 文件。

```
def convert_annotation(image_id):
    in_file = open('data/Annotations/%s.xml' % (image_id))
    out_file = open('data/labels/%s.txt' % (image_id), 'w')
    tree = ET.parse(in_file)
    root = tree.getroot()
    size = root.find('size')
    w = int(size.find('width').text)
    h = int(size.find('height').text)
    for obj in root.iter('object'):
        difficult = obj.find('difficult').text
        cls = obj.find('name').text
        if cls not in classes or int(difficult) == 1:
            continue
        cls_id = classes.index(cls)
        xmlbox = obj.find('bndbox')
        b = (float(xmlbox.find('xmin').text), float(xmlbox.find('xmax').text),
             float(xmlbox.find('ymin').text), float(xmlbox.find('ymax').text))
        bb = convert((w, h), b)
        out_file.write(str(cls_id) + " " + " ".join([str(a) for a in bb]) + '\n')
```

```
0 0.71640625 0.5802083333333333 0.16718750000000002 0.21458333333333332
0 0.9 0.33958333333333335 0.1875 0.225
0 0.6109375 0.20416666666666666 0.1875 0.225
0 0.43984375000000003 0.14479166666666665 0.1796875 0.19375
```

Label 文件

Label 文件中分别为（类别 物体中心 x 坐标 物体中心 y 坐标 边界框宽比例 边界框高比例）这正是模型需要的几个数据。至此所有数据标签和数据准备就完成了。

# 4.2 模型的构建

代码主要分为以下几个部分

### 4.2.1 Parse_config.py

-parse_config(config):

解析模型配置文件

```
def parse_config(config):
    '''
    接受一个配置文件
    返回block列表。 每个块是要建立的神经网络中的一个块。 块在列表中表示为字典
    '''
    file = open(config,'r')
    lines = file.read().split('\n')
    #去除空白行
    lines = [x for x in lines if len(x)>0]
    #去除注释
    lines = [x for x in lines if x[0]!='#']
    #去除左右空格
    lines = [x.rstrip().lstrip() for x in lines]
    #记录每一个块
    block = {}
    blocks = []
    #解析lines列表转化为神经网络的blocks块
    for line in lines:
        #标志一个新的块
        if line[0] == '[':
            #如果这个块不为空
            if (len(block)!=0):
                blocks.append(block)
                block = {}
            block["type"] = line[1:-1].rstrip()
        else:
            key,value = line.split('=')
            block[key.rstrip()] = value.lstrip()
    blocks.append(block)
    return blocks
```

-create_modules(blocks)

解析配置文件。并创建 Module_list 字典，字典中包含了每个模块的类别，例如是卷积层还是路由层还是残差网络或者 yolo 层。

```python
#创建模块，此时blocks是包含所有层的字典
def create_modules(blocks):
    net_info = blocks[0]
    module_list = nn.ModuleList()
    prev_filters = 3
    #记录经过卷积层，当前的通道数量
    output_filters = []

    for index,x in enumerate(blocks[1:]):
        module = nn.Sequential()
        #检查block类型
        #为这个block分配新的module
        #添加到module_list

        #如果是卷积层
        if(x['type'] == 'convolutional'):
            #获取该层的信息
            activation = x['activation']
            try:
                batch_normalize = int(x['batch_normalize'])
                bias = False
            except:
                batch_normalize = 0
                bias = True

            filters = (int)(x['filters'])
            padding = (int)(x['pad'])
            kernel_size = (int)(x['size'])
            stride = (int)(x['stride'])

            #如果有填充,保持卷积前后shape保持SAME
```

```python
            #添加卷积层
            conv = nn.Conv2d(prev_filters,filters,kernel_size,stride,pad,bias = bias)
            module.add_module('conv_{0}'.format(index),conv)

            #添加批归一化层
            if batch_normalize:
                bn = nn.BatchNorm2d(filters)
                module.add_module('batch_norm_{0}'.format(index),bn)

            #查看激活函数
            if activation == 'leaky':
                activn = nn.LeakyReLU(0.1,inplace=True)
                module.add_module('leaky_{0}'.format(index),activn)

        #如果是上采样层
        #使用 Bilinear2dUpsampling
        elif (x['type']=='upsample'):
            stride = int(x['stride'])
            upsample = nn.Upsample(scale_factor=2,mode='nearest')
            module.add_module("upsample_{}".format(index), upsample)


        #如果是路由层,这边不太明白
        elif (x['type']=='route'):
            x['layers'] = x['layers'].split(',')
            #开始的索引
            start = int(x['layers'][0])
            #尝试有没有结束索引
            try:
                end = int(x['layers'][1])
            except:
```

```
        #空层就是为了连接用的。初始化为空层，底下再进行操作。
        route = EmptyLayer()
        #添加路由层
        module.add_module("route_{0}".format(index), route)
        if end < 0 :
            filters = output_filters[index+start]+output_filters[index+end]
        else:
            filters = output_filters[index+start]

    #如果是shortcut层。残差网络
    # 空层就是为了连接用的。初始化为空层，底下再进行操作。
    elif (x['type']=='shortcut'):
        shortcut = EmptyLayer()
        module.add_module('shortcut_{0}'.format(index),shortcut)

    #如果是yolo层
    elif (x['type'] == 'yolo'):
        mask = x['mask'].split(',')
        mask = [int(x) for x in mask]
        anchors = x['anchors'].split(',')
        anchors = [int(x) for x in anchors]
        anchors = [(anchors[i],anchors[i+1]) for i in range(0,len(anchors),2)]
        #指定的3个anchors
        anchors = [anchors[i] for i in mask]
        detection = DetectionLayer(anchors)
        module.add_module('Detecion_{}'.format(index),detection)
    module_list.append(module)
    #纪录前一个的通道数
    prev_filters = filters
    output_filters.append(filters)
return (net_info,module_list)
```

4.2.2 DarkNet.py 创建网络，并定义前向传播。返回的是三张不同尺度特征图按照通道连接。返回模型检测的结果向量。

```
class Darknet(nn.Module):
    def __init__(self, cgfile):
        super(Darknet, self).__init__()
        self.blocks = parse_config(cgfile)
        self.net_info, self.module_list = create_modules(self.blocks)
    # 定义前向传播,self.blocks因为的第一个元素self.blocks是一个net不属于正向传递的块。
    def forward(self, x, CUDA):
        modules = self.blocks[1:]
        # 键值对。key为layer的索引，value是特征矩阵 (feature map)
        outputs = {}
        # 写标志为0
        write = 0
        for i, module in enumerate(modules):
            module_type = module['type']
            if module_type == 'convolutional' or module_type == 'upsample':
                # 如果模块是卷积模块或上采样模块，则这就是正向传递的工作方式。
                x = self.module_list[i](x)
            elif module_type == 'route':
                layers = module['layers']
                layers = [int(a) for a in layers]
                if layers[0] > 0:
                    layers[0] = layers[0] - i
                if len(layers) == 1:
                    x = outputs[i + layers[0]]
                else:
                    if (layers[1]):
                        layers[1] = layers[1] - i
                    mp1 = outputs[i + layers[0]]
                    mp2 = outputs[i + layers[1]]
                    # 在深度上连接，及channels连接，要保证长宽一致
                    x = torch.cat((mp1, mp2), 1)
```

```
                x = torch.cat((mp1, mp2), 1)
                # 残差网络
        elif module_type == 'shortcut':
            from_ = int(module['from'])
            x = outputs[i - 1] + outputs[i + from_]
        elif module_type == 'yolo':
            #获得三个anchors值
            anchors = self.module_list[i][0].anchors
            # 获得输入维度
            input_dim = int(self.net_info['height'])
            # 需要检测的物体个数
            num_classes = int(module['classes'])
            # transform
            x = x.data.cuda()
            #x的shape(batch_size,channels,长，宽)
            #shape torch.Size([1, 255, 13, 13])
            #print('prediction.shape',x.shape)
            x = predict_transform(x, input_dim, anchors, num_classes, CUDA)
            # 第一次yolo检测的时候，因为第二张检测图还没生成，还不能concat
            if not write:  # if no collector has been intialised.
                detections = x
                write = 1
            else:
                detections = torch.cat((detections, x), 1)
        outputs[i] = x
    # 返回的是三张特征图的连接
    return detections
```

按照存储顺序加载权重。

```
    # 从weights中加载参数
    bn_biases = torch.from_numpy(weights[ptr:ptr + num_bn_biases])
    ptr += num_bn_biases

    bn_weights = torch.from_numpy(weights[ptr: ptr + num_bn_biases])
    ptr += num_bn_biases

    bn_running_mean = torch.from_numpy(weights[ptr: ptr + num_bn_biases])
    ptr += num_bn_biases

    bn_running_var = torch.from_numpy(weights[ptr: ptr + num_bn_biases])
    ptr += num_bn_biases

    # 把权重reshape成模型需要的参数的形状
    bn_biases = bn_biases.view_as(bn.bias.data)
    bn_weights = bn_weights.view_as(bn.weight.data)
    bn_running_mean = bn_running_mean.view_as(bn.running_mean)
    bn_running_var = bn_running_var.view_as(bn.running_var)

    # 复制参数到模型中去
    bn.bias.data.copy_(bn_biases)
    bn.weight.data.copy_(bn_weights)
    bn.running_mean.copy_(bn_running_mean)
    bn.running_var.copy_(bn_running_var)

else:
    # 如果没加载成功，获得卷积偏差参数的数量
    num_biases = conv.bias.numel()

    # 加载权重
    conv_biases = torch.from_numpy(weights[ptr: ptr + num_biases])
```

## 4.3 模型训练

```
for epoch in range(start_epoch, epochs):  # epoch ------------------------------------------------------
    model.train()
    print(('\n' + '%10s' * 8) % ('Epoch', 'gpu_mem', 'GIoU', 'obj', 'cls', 'total', 'targets', 'img

    # Freeze backbone at epoch 0, unfreeze at epoch 1 (optional)
    freeze_backbone = False
    if freeze_backbone and epoch < 2:
        for name, p in model.named_parameters():
            if int(name.split('.')[1]) < cutoff:  # if layer < 75
                p.requires_grad = False if epoch == 0 else True

    # Update image weights (optional)
    if dataset.image_weights:
        w = model.class_weights.cpu().numpy() * (1 - maps) ** 2  # class weights
        image_weights = labels_to_image_weights(dataset.labels, nc=nc, class_weights=w)
        dataset.indices = random.choices(range(dataset.n), weights=image_weights, k=dataset.n)  # r

    mloss = torch.zeros(4).to(device)  # mean losses
    pbar = tqdm(enumerate(dataloader), total=nb)  # progress bar
    for i, (imgs, targets, paths, _) in pbar:  # batch -------------------------------------------------
        ni = i + nb * epoch  # number integrated batches (since train start)
        imgs = imgs.to(device)
        targets = targets.to(device)

        # Multi-Scale training
        if opt.multi_scale:
            if ni / accumulate % 10 == 0:  #  adjust (67% - 150%) every 10 batches
                img_size = random.randrange(img_sz_min, img_sz_max + 1) * 32
            sf = img_size / max(imgs.shape[2:])  # scale factor
```

```
pred = model(imgs)

# Compute loss
loss, loss_items = compute_loss(pred, targets, model)
if not torch.isfinite(loss):
    print('WARNING: non-finite loss, ending training ', loss_items)
    return results

# Scale loss by nominal batch_size of 64
loss *= batch_size / 64

# Compute gradient
if mixed_precision:
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
else:
    loss.backward()

# Accumulate gradient for x batches before optimizing
if ni % accumulate == 0:
    optimizer.step()
    optimizer.zero_grad()

# Print batch results
mloss = (mloss * i + loss_items) / (i + 1)  # update mean losses
mem = torch.cuda.memory_cached() / 1E9 if torch.cuda.is_available() else 0  # (GB)
s = ('%10s' * 2 + '%10.3g' * 6) % (
    '%g/%g' % (epoch, epochs - 1), '%.3gG' % mem, *mloss, len(targets), img_size)
pbar.set_description(s)
```

模型训练我们使用了 torch 的混合精度训练，相当于实验 5 中 tensorflow 的量化模型，把 float32 转化为 float16，显著减小了 gpu 显存的开销和模型权重文件的大小。我们的指标是 map（均值平均精度，即 MAP）。经过 300 轮迭代，最终 map 能够到达 70%左右，已经是相当不错了。（不同于 accuracy，acuuracy 可能要到 95%及以上就非常好了。）

## 4.4 火焰检测

-arg_parse 进行参数设置。设置需要的文件路径，批量大小，目标的置信度阈值，非极大值抑制的阈值，或者需要检测的类别图片视频或者摄像头。

用 cv2.VideoCapture(arg)能识别不同的类别，如果需要检测视频，arg 为视频位置。如果要实时检测摄像头中的数据，那么 arg=0 为检测摄像头。

```
def arg_parse():
    """
    Parse arguements to the detect module

    """

    parser = argparse.ArgumentParser(description='YOLO v3 检测模块')

    parser.add_argument("--images", dest='images',
                        help="Image / 图片所在的文件夹目录",
                        default="imgs", type=str)
    parser.add_argument("--det", dest='det', help="Image /  检测的结果的图片文件夹目录",
                        default="det", type=str)
    parser.add_argument("--bs", dest="bs", help="批量大小（默认为1）", default=1)
    parser.add_argument("--confidence", dest="confidence", help="目标的置信度阈值", default=0.5)
    parser.add_argument("--nms_thresh", dest="nms_thresh", help="非极大值抑制的阈值", default=0.4)
    parser.add_argument("--cfg", dest='cfgfile', help="配置文件目录",
                        default="cfg/yolov3.cfg", type=str)
    parser.add_argument("--weights", dest='weightsfile', help="Darknet网络的权重",
                        default="cfg/yolov3.weights", type=str)
    parser.add_argument("--reso", dest='reso', help="输入图片的分辨率",
                        default="416", type=str)
    parser.add_argument("--scales", dest="scales", help="用于检测的Scale",
                        default="1,2,3", type=str)

    return parser.parse_args()
```

Detect_images 对设置的文件目录下所有的图片进行检测和画框处理，并把处理的图片存储到目标文件夹下。

其中 def　detect（）函数为核心的检测函数。

```
def detect(save_txt=False, save_img=False):
    img_size = (320, 192) if ONNX_EXPORT else opt.img_size  # (320, 192) or (416, 256) or (608, 352) for (height
    out, source, weights, half, view_img = opt.output, opt.source, opt.weights, opt.half, opt.view_img
    webcam = source == '0' or source.startswith('rtsp') or source.startswith('http') or source.endswith('.txt')

    # Initialize
    device = torch_utils.select_device(device='cpu' if ONNX_EXPORT else opt.device)
    if os.path.exists(out):
        shutil.rmtree(out)  # delete output folder
    os.makedirs(out)  # make new output folder

    # Initialize model
    model = Darknet(opt.cfg, img_size)

    # Load weights
    attempt_download(weights)
    if weights.endswith('.pt'):  # pytorch format
        model.load_state_dict(torch.load(weights, map_location=device)['model'])
    else:  # darknet format
        _ = load_darknet_weights(model, weights)

    # Second-stage classifier
    classify = False
    if classify:
        modelc = torch_utils.load_classifier(name='resnet101', n=2)  # initialize
        modelc.load_state_dict(torch.load('weights/resnet101.pt', map_location=device)['model'])  # load weights
        modelc.to(device).eval()
```

# 4.5 构建基于 Flask 的实时火焰检测网站

## 4.5.1 概述

我们需要通过 Http 协议以较低延迟呈现给浏览器实时监控视频。通过 Http 协议传输视频常见有 HTTP-FLV 方式和 MJPEG 方式。HTTP-FLV 协议将音视频数据封装成 FLV, 然后通过 HTTP 协议的 Chunked-Encoding 传输给客户端，理论上（除去网络延迟）可以做到一个音视频 Tag 的延迟。而该 Project 采用实现起来更为简单且延迟较低，但以增加带宽占用为代价的 MJPEG 流方式传输影像；即将摄像头每帧输入转为 jpeg 编码图像，通过 Http Multipart 字段传输每一帧的 jpeg 图像，浏览器收到后通过<image>标签显示，不断重复获取每一帧并刷新。

## 4.5.2 Http Multipart 字段

MJPEG 流主要依靠 Http Multipart 字段传输，该字段被绝大部分浏览器支持。Multipart 字段由 RFC1341 定义，具有多种不同的类型，针对流媒体，使用 multipart/x-mixed-replace。Multipart 将一系列数据块组合在一个 Http Body 里，并将 HTTP 头部的 Content-Type 字段指定为 multipart；每个 Http Body 包含至少一块数据，块与块间用 boundary 分隔，最后一块末尾为一个 closing boundary，boundary 由程序指定，通常为一定长度的随机字符串，只要不容易与数据内容重复即可。浏览器收到这种 Multipart 类型时，会使用当前块的数据替换先前块的数据，由此画面内容将被逐帧显示出来。只要服务器不主动关闭连接，浏览器会源源不断地获取并显示每一帧图像，由此形成一个连续的视频流。

服务器通过 HTTP Multipart 字段向浏览器传输 MJPEG 流的一个典型 HTTP 响应例子如下（仅列出关键信息）：

HTTP/1/1 200 OK
Content-Type:multipart/x-mixed-replace; boundary=KSoRhZx7t8UKsx8po39G6lz1oLd
--KSoRhZx7t8UKsx8po39G6lz1oLd
Content-Type:image/jpeg
Content-Length:436312
[image1 enconded jpeg data]

--KSoRhZx7t8UKsx8po39G6lz1oLd
Content-Type:image/jpeg
Content-Length:473253
[image2 enconded jpeg data]
...

其中”[image x encoded jpeg data]”表示经过编码后的 jpeg 图片数据，图片间分隔符为程序随机生成的字符串，例如“KSoRhZx7t8UKsx8po39G6lz1oLd”。

## 4.5.3 代码实现

指定 http 协议所需要的“multipart/x-mixed-replace”MIME 类型，当浏览器 uri 为/video_streamer/<cam_index>时产生该 Response。

```
@app.route('/video_streamer/<cam_index>')
def video_streamer(cam_index):
```

```
# A stream where each part replaces the previous part the
multipart/x-mixed-replace content type must be used.
return
Response(jpeg_gen(),mimetype='multipart/x-mixed-replace;
boundary=frame_boundary')
```

利用 Python 提供的 Generator 语法每次生成一个数据帧，包括 Header 和 Payload。

```
def jpeg_gen():
while True:
jpeg=fire_detector.get_jpeg()
# Builds 'jpeg' data with header and payload
yield (b'--frame_boundary\r\n'
b'Content-Type:image/jpeg\r\n\r\n'+jpeg+b'\r\n\r\n')
```

### 4.5.4 性能与优劣

由于 MJPEG 只采用帧内压缩，编解码算法简单，故节约硬件资源[21]。压缩后除了 JPEG 自身的损失，没有帧间编码带来的画质损失。经测试，局域网条件下传输 MJPEG 流用户几乎感觉不到延迟（延迟≤0.2s），如果在公网传输，同一个城域网下延迟一般在 1 秒左右，实时性较好。

缺点主要为 MJPEG 编码方式带来的对视频压缩效率较低，网络传输占用带宽较大等负面影响，720p 中低质量 MJPEG 流带宽占用约为 6.4 Mbps。

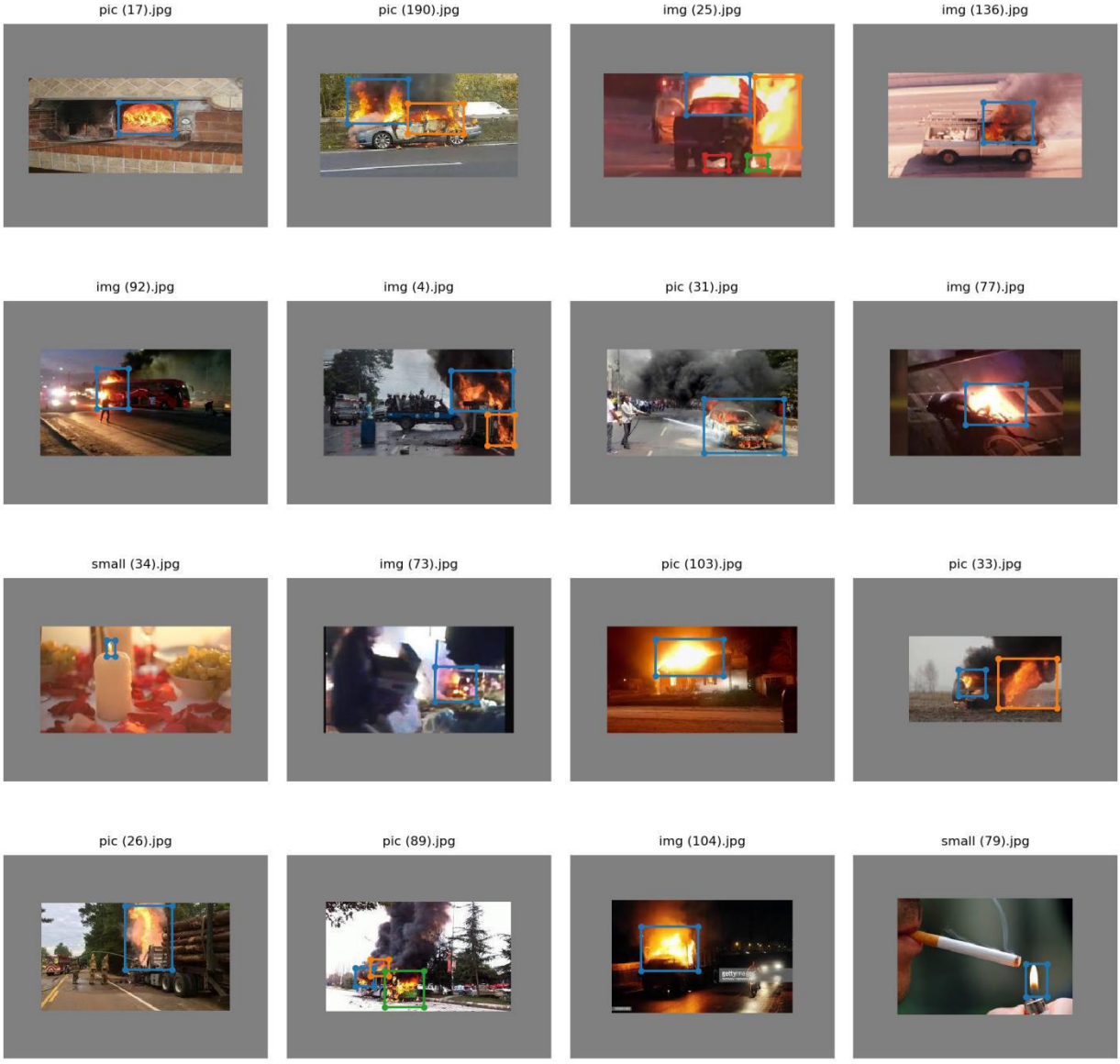
### 4.5.5 Flask 线程与 YOLO 线程交互

YOLO 线程读摄像头帧进行火灾检测并将处理后的帧画面写入一个帧全局变量，Flask 线程处理网络连接并读该全局变量，将识别后的画面实时传输出去；帧全局变量使用 Python 提供的 Condition 和 RLock 进行同步。Flask 在收到 Stream 请求后开启一个新线程来推流，它会阻塞在 Condition 上，当 YOLO 处理完一帧后唤醒它，让其将这一帧画面包装成 HTTP 响应发送给浏览器。

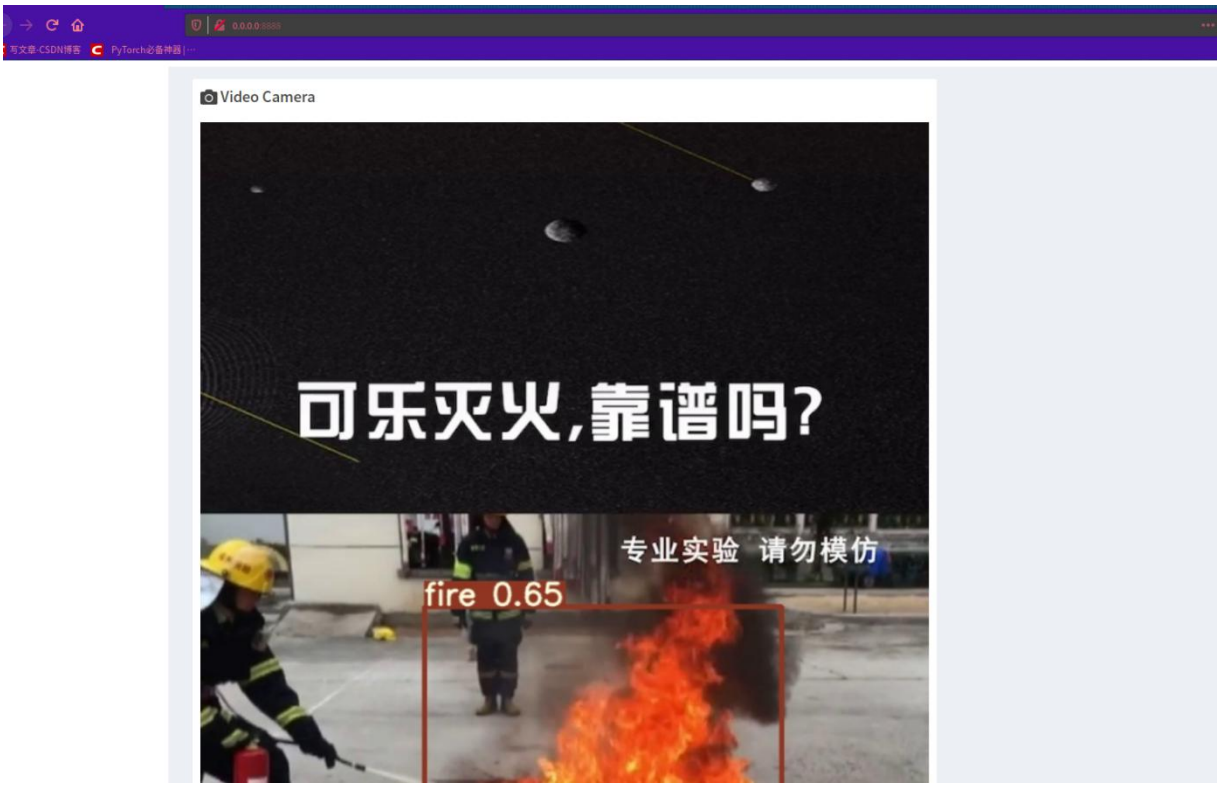
# 5. 结果分析

经过不断优化，模型能够比较完美的检查到目标并能够确定目标的位置。

## 5.1　图片检测效果




## 5.2 Flask 构建的网站及实时检测




## 5.3 视频文件

请见文件夹中 demo.mp4 文件。能够实时并且准确地识别火焰，能够有效预防火灾。

# 6. 参考文献

You Only Look Once: Unified, Real-Time Object Detection; Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi;The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779-788

YOLOv3: An Incremental Improvement; Joseph Redmon, Ali Farhadi; arXiv:1804.02767v1 [cs.CV] 8 Apr 2018

# 7. 分工合作

- YOLO_v3 的网络构建） 李亘杰 周渝茗
- 训练集的搜寻和数据标注 周渝茗
- 算法的细节实现（预测向量的转化，反向传播的调节） 李亘杰 周渝茗
- 模型的冻结，训练，调参 杨雨丰 李亘杰
- 图片目标检测的测试 杨雨丰 周渝茗
- 视频的实时检测 李亘杰 杨雨丰
- Flask 构建实时火焰检测网站 杨雨丰