浙江大学

科 研 训 练 结 题 报 告

项目名称　智能车对环境的感知与识别

姓名学号　曾之宸 3160105111

李辰睿 3160105144

指导老师　陈　剑

年级专业　2016 级 自动化（控制）

所在学院　控制科学与工程学院

结题时间　2019 年 5 月

目 录

智能车对环境的感知与识别

曾之宸　李辰睿

（浙江大学控制科学与工程学院，浙江 杭州　310027）

摘　要：

近年来，随着人工智能技术的发展，人们对智能车的性能及应用提出了更进一步的要求。相比于传统的辅助型智能车，如今的发展趋势更倾向于无人驾驶，即通过传感器等检测车辆周围实时环境条件，并进行即时处理、给出决策控制。这其中，对于外界环境的感知识别是后续其他各步的基础，快速、准确的收集环境信息并进行处理在智能车的应用中起到了关键性作用。如今常采用的环境信息探测方式有多种多样，激光雷达、RGBD 相机、双目相机、红外探测器等等，根据具体的应用场景不同，可以选择的方式也多种多样。考虑到目前识别任务下诸多智能算法相比于传统方法体现出了较好的性能，并且针对图像信息的处理方法也更加多种多样，我们选取了视觉探测器为核心的方式，结合一些较好的网络结构进行改进，应用于智能车的视觉识别目标上。

关键词：图像复原与增强；目标检测算法；视觉测距算法

Abstract：

In recent years, with the development of artificial intelligence, people have put forward further requirements for the performance and application of smart cars. Compared with the traditional auxiliary smart cars, the current development trend is more inclined to unmanned driving, that is, detecting real-time environmental conditions around the smart car through sensors, etc., and performing immediate processing and giving decision control. Among them, the perceptual recognition of the external environment is the basis of other subsequent steps. The rapid and accurate collection of environmental information and processing plays a key role in the application of smart cars. There are many kinds of environmental information detection methods commonly used today, such as laser radar, RGBD camera, binocular camera, infrared detector, etc., depending on the specific application scenario, there are various ways to choose. Considering that many intelligent algorithms under the current recognition task show better performance than traditional methods, and the processing methods for image information are more diverse, we have chosen the visual detector as the core method, combining some better. The network structure is improved and applied to the visual recognition target of the smart car.

Keywords： Image restoration and enhancement; Target detection; Visual ranging

1 绪论

1.1 项目背景

无人驾驶技术是人工智能时代的一大研究热点，智能车的研发涉及机器视觉，行为决策以及控制算法等多个方面。智能车的普及一方面能够将人类从疲劳的驾驶中解放出来；另一方面，智能车不受限于驾驶人员的精神状况，反应速度等个人因素，故理想状况下具有更高的安全性，稳定性及运行效率。但是由于当代城市环境中的复杂道路结构，多变环境背景，众多车辆种类以及频发雾霾天气，使智能车的机器视觉识别面临着许多的挑战。因此需要研发更为准确高效的目标检测和目标跟踪算法，才能更好地满足智能车驾驶的需求。虽然当今国际上智能车驾驶的安全性已有很大的提高，但是 Google 无人车路测发生碰撞，以及近期 Uber 无人车伤人的事件，仍显示智能车的安全性仍未达到上路的标准。本课题旨在在已有的智能车平台上进行智能车的机器视觉算法研究，提高智能车的环境感知能力，以应对错综复杂的城市背景和瞬息万变的交通状况；同时，考虑到智能车高速运行、恶劣天气状况下运行时摄像机拍照的模糊性，也需要相应的算法对模糊照片进行预处理，使检测目标更为明显，为决策提供更为精确的信息。精确高效的目标检测使智能车行驶安全性的保障，希望早日推广智能车的应用，安全性问题不容忽视，目标检测算法的研究迫在眉睫。

1.2 智能车发展现状

美国，德国等发达国家目前已有不少企业将智能车技术推进到地面实测阶段。美国方面，Google 智能车已在美国本土以极低的事故率行驶超过 130 万英里。德国的奔驰、博世等公司已在多种路况下测试智能车，而且这些测试都获得了政府的支持。反观国内的智能车发展现状，相比于国际上前列的国家，我国的智能车技术大多仍停留在实验室阶段。红旗，百度等国内企业虽已取得突破性的成就，但进行了少量路测数据量较少，不足以保证智能车普及应用的安全性。

1.3 文章结构

本文主要以项目开展时间线作为主线，逐步讲解本科研训练项目中涉及的图形预处理、目标检测、基于嵌入式系统的改进及展望。本文第一章为绪论，整体介绍了本项目的项目背景、智能车的发展现状等情况。第二章为视觉信息预处理，对采集到的单个图片帧在进行识别前的去噪、增强过程进行了阐述。第三章为目标检测算法，阐述了用于进行车辆周围环境进行目标识别检测的算法流程及网络结构。第四章为基于嵌入式控制器的改进，针对程序的移植中可能存在的问题进

行了解决方案设计。第五章为总结和展望。

2 视觉信息预处理

2.1 图像预处理需求

车载摄像头采集到周围环境的 RGB 图像信息，在进行进一步的目标检测之前，需要进行复原与增强处理，具体要求为：

① 减少雾霾、雨雪等恶劣环境对图像质量造成的影响

② 消除减少因移动产生的运动模糊

③ 消除因相机自身原因产生的椒盐噪声等

根据这些目标，我们针对不同的要求采取了不同的方法，这里利用 OpenCV 图像增强技术，如利用直方图均衡化图像增强技术提高图像对比度；利用拉普拉斯算子图像增强技术锐化模糊图像；利用对数 log 变换图像增强技术提高昏暗环境下图像清晰度；利用伽马变换图像增强技术校正灰度过高或过低的图像，以提高对比度等。

2.2 算法实现

2.2.1 直方图均衡

① 直方图均衡算法介绍

根据摄像头采集到的视频数据，提取视频不同帧对应的图像，我们对其进行直方图均衡操作。直方图均衡化处理是以累计分布函数变换法作为基础的。该方法能够将像素值分布较为集中，图像对比度较低的图像进行灰度直方图均衡，使其灰度分布范围更广、更为平均，从而提高图像的对比度。

工程上一般采取如下变换函数：

$$s = T(r) = \int_0^r p_r(\omega) d\omega$$

其中 s 为变换后的像素值，r 为变换前的像素值，$p_r(\omega)$ 为变换前图像像素值的概率密度分布函数。对于数字图像，我们通常采用求和的方式替代连续函数中的积分操作。

② 直方图均衡算法 Python 代码实现

```
1. """直方图均衡化 python 代码实现"""
2. from PIL import Image
3. import pylab as pl
```

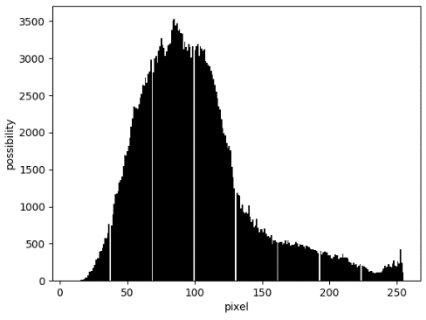
```
import numpy as np


def histeq(im,nbr_bins = 256):  #直方图均衡化子函数
    imhist,bins = np.histogram(im.flatten(),nbr_bins,normed= True)
    cdf = imhist.cumsum()   #获取原图像的灰度值概率分布(完成概率密度积分)
    cdf = 255.0 * cdf / cdf[-1]     #将[0,1]的概率分布函数转换为[0,255]范围内
    im2 = np.interp(im.flatten(),bins[:-1],cdf)     #对于原图像像素 im,依据函数 cdf=f(bins)进行线性插值(映射)
    return im2.reshape(im.shape),cdf


pil_im = Image.open('image.bmp')         #打开原图
pil_im_gray = pil_im.convert('L')        #转化为灰度图像
pil_im_gray.show()          #显示灰度图像

im = np.array(Image.open('image.bmp').convert('L'))
pl.figure()
pl.hist(im.flatten(),256,color='black')  #显示原图像灰度直方图
pl.xlabel('pixel')
pl.ylabel('possibility')

im2,cdf = histeq(im)
pl.figure()
pl.hist(im2.flatten(),256,color='black') #显示直方图均衡后图像灰度直方图
pl.xlabel('pixel')
pl.ylabel('possibility')
pl.show()

im2 = Image.fromarray(np.uint8(im2))
im2.show()
im2.save("imagenew.bmp")
```

Code 2.2.1.1 直方图均衡 Python 代码实现

③ 直方图均衡算法结果展示

Fig 2.2.1.1 原图像

Fig 2.2.1.2 直方图均衡化图像



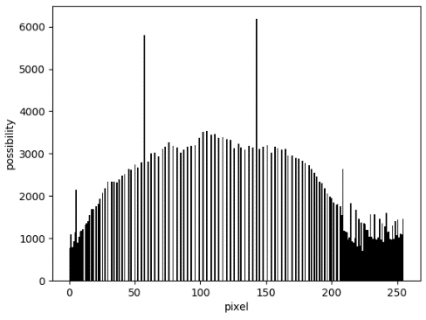


Fig 2.2.1.3 原图像灰度分布直方图

Fig 2.2.1.4 直方图均衡化图像灰度分布

④ 直方图均衡结果分析

比较变化前后的两张图像，我们可以明显地感觉到原图像风格偏暗，处理后的图像中出现了较多的亮色区域(如天空、草地)，从而使得图像对比度整体提高。根据两张灰度分布直方图可以看到，原图像整体风格偏暗，像素值大多集中在(50,130)区间内，图像对比度较低。经过直方图均衡化处理后，可以看到图像的像素值大致均匀地分布在了[0,255]区间内，使图片的像素值范围较大大，图像的对比度增强。

2.2.2 图像滤波去噪

① 均值滤波、中值滤波算法介绍

根据摄像头采集到的视频数据，提取视频不同帧对应的图像，我们对其进行图像滤波操作。对于均值去噪操作，我们分别选取选取 $\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$, $\frac{1}{25}\begin{bmatrix}1&\cdots&1\\\vdots&\ddots&\vdots\\1&\cdots&1\end{bmatrix}_{5\times5}$, $\frac{1}{49}\begin{bmatrix}1&\cdots&1\\\vdots&\ddots&\vdots\\1&\cdots&1\end{bmatrix}_{7\times7}$ 三种大小的卷积核进行卷积操作，进而实现均值去噪操作。对于卷积操作的边界，我们采取填零的方式进行扩充，以保证输出图像与输入图像尺寸相同。

对于中值去噪操作，我们分别选取选取 $\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$, $\begin{bmatrix}1&\cdots&1\\\vdots&\ddots&\vdots\\1&\cdots&1\end{bmatrix}_{5\times5}$, $\begin{bmatrix}1&\cdots&1\\\vdots&\ddots&\vdots\\1&\cdots&1\end{bmatrix}_{7\times7}$ 三三种大小的卷积核进行中值选取，进而实现中值去噪操作。对于卷积操作的边界，我们采取填零的方式进行扩充，以保证输出图像与输入图像尺寸相同。

② 图像滤波去噪算法 Python 代码实现

```
'''''均值去噪 python 代码实现'''
import numpy as np
import os
from skimage import io
import skimage

def load_image_skimage(filename,isFlatten=False):   #读取图像子函数
    isExit=os.path.isfile(filename)
    if isExit==False:
        print("File open error!")
    img=io.imread(filename)  #io.read(filename,img)读取文件
    if isFlatten:
        img_flatten=np.array(np.array(img,dtype=np.uint8).flatten())

        return img_flatten
    else:
        img_arr=np.array(img,dtype=np.uint8)
        return img_arr

def noising():
    image=load_image_skimage('image.jpg')
    image_gaussian=skimage.util.random_noise(image,'gaussian') #添加高斯噪声
    image_salt=skimage.util.random_noise(image,'salt')          #添加盐噪声
    image_pepper=skimage.util.random_noise(image,'pepper')      #添加椒噪声
    skimage.io.imsave('image_gaussian.jpg',image_gaussian)
    skimage.io.imsave('image_salt.jpg',image_salt)
    skimage.io.imsave('image_pepper.jpg',image_pepper)

class denoising(object):     #定义领域平均化去噪类
    def __init__(self):
        self.filter=np.ones((7,7))     #定义卷积核
        self.zp=(self.filter.shape[0]-1)//2   #定义扩充大小
        self.size=self.filter.shape[0]*self.filter.shape[1]  #计算卷积核大小

    def padding(self,dic):    #填补子函数
        self.padding_dic=np.zeros((2*self.zp+dic.shape[0],2*self.zp+dic.shape
[1],dic.shape[2]))
        self.padding_dic[self.zp:(dic.shape[0]+self.zp),self.zp:(dic.shape[1]
+self.zp),:]=dic

    def conv(self,filename):              #卷积子函数
        self.dic=load_image_skimage(filename,isFlatten=False)
        self.padding(self.dic)
```

```
41.         temp_dic=self.padding_dic  #将图像扩展为卷积尺度
42.         self.output=np.zeros((self.dic.shape[0],self.dic.shape[1],self.dic.sh
    ape[2]))  #创建输出矩阵
43.         for k in range(self.dic.shape[2]):
44.             for j in range(self.zp,self.dic.shape[1]+self.zp):
45.                 for i in range(self.zp,self.dic.shape[0]+self.zp):
46.                     temp=temp_dic[(i-self.zp):(i+self.zp+1),(j-
    self.zp):(j+self.zp+1),k]
47.                     self.output[i-self.zp,j-
    self.zp,k]=np.sum(temp*self.filter)/(self.size*255)
48.         return self.output
49.
50. if __name__=='__main__':
51.     func=denoising()
52.     noising()       #给图像分别加入高斯噪声、盐噪声、椒噪声
53.     output=func.conv('image_gaussian.jpg')
54.     skimage.io.imsave('imagenew_gaussian(7).jpg',output)    #高斯噪声图像处理
55.     output=func.conv('image_salt.jpg')
56.     skimage.io.imsave('imagenew_salt(7).jpg',output)        #盐噪声图像处理
57.     output=func.conv('image_pepper.jpg')
```

Code 2.2.2.1 图像均值滤波去噪 Python 代码实现

```
1.  '''''中值去噪 python 代码实现'''
2.  import numpy as np
3.  import os
4.  import skimage
5.  from skimage import io
6.
7.  def load_image_skimage(filename,isFlatten=False):    #读取图像子函数
8.      isExit=os.path.isfile(filename)
9.      if isExit==False:
10.         print("File open error!")
11.     img=io.imread(filename)  #io.read(filename,img)读取文件
12.     if isFlatten:
13.         img_flatten=np.array(np.array(img,dtype=np.uint8).flatten())

14.         return img_flatten
15.     else:
16.         img_arr=np.array(img,dtype=np.uint8)
17.         return img_arr
18. def noising():
19.     image=load_image_skimage('image.jpg')
20.     image_gaussian=skimage.util.random_noise(image,'gaussian')  #添加高斯噪
    声
```

```
    image_salt=skimage.util.random_noise(image,'salt')            #添加盐噪声
    image_pepper=skimage.util.random_noise(image,'pepper')        #添加椒噪声
    skimage.io.imsave('image_gaussian.jpg',image_gaussian)
    skimage.io.imsave('image_salt.jpg',image_salt)
    skimage.io.imsave('image_pepper.jpg',image_pepper)

class denoising(object):      #定义领域平均化去噪类
    def __init__(self):
        self.filter=np.ones((3,3))      #定义卷积核
        self.zp=(self.filter.shape[0]-1)//2    #定义扩充大小
        self.size=self.filter.shape[0]*self.filter.shape[1]   #计算卷积核大小

    def padding(self,dic):     #填补子函数
        self.padding_dic=np.zeros((2*self.zp+dic.shape[0],2*self.zp+dic.shape
[1],dic.shape[2]))
        self.padding_dic[self.zp:(dic.shape[0]+self.zp),self.zp:(dic.shape[1]
+self.zp),:]=dic

    def conv(self,filename):               #卷积子函数
        self.dic=load_image_skimage(filename,isFlatten=False)
        print(self.dic.shape)
        temp_dic=self.dic
        self.output=np.zeros((self.dic.shape[0],self.dic.shape[1],self.dic.sh
ape[2]))  #创建输出矩阵
        for k in range(self.dic.shape[2]):
            for j in range(self.zp,self.dic.shape[1]+self.zp):
                for i in range(self.zp,self.dic.shape[0]+self.zp):
                    temp=temp_dic[(i-self.zp):(i+self.zp+1),(j-
self.zp):(j+self.zp+1),k]
                    self.output[i-self.zp,j-
self.zp,k]=np.median(temp)/255       #选区中值
        return self.output

if __name__=='__main__':
    func=denoising()
    noising()       #给图像分别加入高斯噪声、盐噪声、椒噪声
    output=func.conv('image_gaussian.jpg')
    skimage.io.imsave('imagenew_gaussian.jpg',output)    #高斯噪声图像处理
    output=func.conv('image_salt.jpg')
    skimage.io.imsave('imagenew_salt.jpg',output)        #盐噪声图像处理
    output=func.conv('image_pepper.jpg')
    skimage.io.imsave('imagenew_pepper.jpg',output)      #椒噪声图像处理
```

Code 2.2.2.2 图像中值滤波去噪 Python 代码实现

③ 图像滤波去噪算法结果展示

Fig 2.2.2.1 图像滤波去噪图像

④ 图像滤波去噪结果分析

可以看到，对于两种去噪方式，随着卷积核大小的增大，噪声点的去除程度越高，但相应的图片也会变得更加模糊。更具上述展示出来的结果可以看到，对于较低维度的卷积核，中值去噪能够更好地去除噪声点，而均值去噪相对来说，较低维度的卷积核并不能很好地去噪，噪声点较为明显。这一点能够得到解释，因为图像中的噪声点(特别是盐噪声与椒噪声)，其像素值均有非常明显的变化(例如盐噪声像素值很大，椒噪声像素值很小)，采取中值去噪能够直接忽略掉这些噪声对于原图像的影响；而对于均值去噪则是采用一种平均化的操作将像素中的突变点数值平均化，能够一定程度上削弱噪声点的影响，且在一定范围内卷积核越大，平均效果越显著，削弱效果越明显。

针对均值去噪操作而言，可以看到其对于高斯噪声的处理结果较好，而针对椒盐噪声的处理结果一般。椒盐噪声是单纯的白噪声或黑噪声，使用平均化的处理只能单纯地削弱；而对于高斯噪声，其噪声点的颜色分布较为随机，所以平均化的操作能够一定程度上将不同的颜色间进行抵消，所以处理结果较好。

而针对均值去噪操作，可以看到其针对高斯噪声的处理结果一般，而针对椒盐噪声有很好的抑制效果。椒盐噪声是颜色单一的噪声，其像素值相对于周围点差别较大，所以采用取中值的方式能够很好地将噪声点消除掉；而对于高斯噪声，由于噪声点像素分布较为随机，所以单纯地使用取中值的方式，很有可能出现误操作，导致对高斯噪声的处理效果不尽人意。

2.2.3 图像复原

① 图像复原算法介绍

运动模糊问题实际上也即是一个点扩散函数的求解问题。对于频域表示为 $F(u,v)$ 的原图像，通过计算得到相应的运动模糊点扩散函数 $H(u,v)$，可以得到频域下相应模糊化之后的模糊图像 $G(u,v)=H(u,v)F(u,v)$。对于曝光时间内运动像素长度为 a，运动角度为 θ 的图像，其相应的模糊点扩散函数为 $H(u,v)=\int_0^T e^{-j2\pi[ux_0(t)+vy_0(t)]}dt$，其中 $x_0(t),y_0(t)$ 分别物体在 x 方向与 y 方向的分量。

逆滤波法其在基本思想是根据已知的点扩散函数 $H(u,v)=\int_0^T e^{-j2\pi[ux_0(t)+vy_0(t)]}dt$，根据频域关系式 $G(u,v)=H(u,v)F(u,v)$，我们可以逆向滤波得到 $F(u,v)=\frac{G(u,v)}{H(u,v)}$ 从而获得去模糊化后的图像 $F(u,v)$。之后在通过快速傅里叶反变换即可获得相应的清晰图像。这种方法对于模糊原因已知，点扩散函数已知的图像去模糊化非常有效；但是对于模糊原因不知或是点扩散函数不可知的情况，其难以得到应用。

而维纳滤波器是基于最小均方差的原则对图像进行复原的，其目标是找到未污染图像 $F(u,v)$ 的一个估计，使它们之间的均方误差最小，即 $\min e^2=\min E\{(f-\hat{f})^2\}$ 所以其也被称为最小均方差滤波。维纳滤波器的优点在于其能处理被退化函数退化和噪声污染的图像。在假设噪声和图像不相关，其中一个或另一个有零均值，且估计中的灰度级是退化图像中灰度级的线性函数的条件下，均方误差函数的最小值在频率域由如下表达式给出：

$$\hat{F}(u,v)=\left[\frac{H^*(u,v)S_f(u,v)}{S_f(u,v)|H(u,v)|^2+S_\eta(u,v)}\right]G(u,v)=\left[\frac{H^*(u,v)S_f(u,v)}{|H(u,v)|^2+S_\eta(u,v)/S_f(u,v)}\right]G(u,v)$$
$$=\left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2+S_\eta(u,v)/S_f(u,v)}\right]G(u,v)$$

其中，$H(u,v)$ 为退化函数，$H^*(u,v)$ 为 $H(u,v)$ 的复共轭，$|H(u,v)|^2=H(u,v)H^*(u,v)$，$S_\eta(u,v)=|F(u,v)|^2$=噪声的功率谱，$S_f(u,v)=|F(u,v)|^2$==未退化图像的功率谱，$H(u,v)$ 是退化函数的傅里叶变换，$G(u,v)$ 是退化后图像的傅里叶变换。

从上面的公式可以发现，如没有噪声，即 $S_\eta(u,v)=0$，此时维纳滤波变为直接逆滤波，如有噪声，那么 $S_\eta(u,v)$ 如何估计将成问题，同时 $S_f(u,v)$ 的估计也成问题。在实际应用中假设退化函数已知，如果噪声为高斯白噪声，则 $S_\eta(u,v)$ 为常数，但

$S_f(u,v)$ 通常难以估计。一种近似的解决办法是用一个系数 K 代替 $S_\eta(u,v)/S_f(u,v)$，因此上面的公式变为如下式所示：

$$\hat{F}(u,v)=\left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2+K}\right]G(u,v)$$

实际应用中根据处理的效果选取合适的 K 值。

② 图像复原算法 Python 代码实现

```
"""基于频域逆滤波求解与维纳滤波器求解的图像复原"""
import matplotlib.pyplot as graph
import numpy as np
import scipy.misc
from numpy import fft
import math
import cv2
#定义常量，运动模糊角度,运动模糊距离以及噪声功率
angle=30
dist=50
eps=0.001
a=0.1
#定义图像复原类，其中包括运动模糊图像生成，逆滤波复原以及维纳滤波复原
class restoration(object):
    def __init__(self):
        self.image=cv2.cvtColor(cv2.imread('image.bmp'),cv2.COLOR_BGR2GRAY) #载入原图像
        self.image_size=(self.image.shape[0],self.image.shape[1])
        self.PSF=np.zeros(self.image_size)              #点扩散函数
        self.blurred=np.zeros(self.image_size)          #运动模糊处理后的图像
        self.inverse_result=np.zeros(self.image_size)   #逆变换复原图像
    # 运动模糊,输入原图像,运动角度,以及运动距离， 返回点扩散函数
    def motion(self,angle,dist):
        PSF=np.zeros(self.image_size) #构造点扩散函数矩阵 point_spread_function
        center_position=(self.image_size[0]-1)/2    #图像中心位置
        slope_tan=math.tan(angle*math.pi/180)
        slope_cot=1/slope_tan
        #构造直线运动的点扩散函数,是一条角度等于运动角度,长度等于运动距离的斜直线
        #将点扩散函数所对应像素值设为1,其余点像素值设为0
        if slope_tan<=1:
            for i in range(dist):
                offset=round(i*slope_tan)
                PSF[int(center_position+offset),int(center_position-offset)]=1
        else:
            for i in range(dist):
                offset=round(i*slope_cot)
                PSF[int(center_position-offset),int(center_position+offset)]=1
        self.PSF=PSF/PSF.sum()  #对点扩散函数进行归一化亮度

    #对图片进行运动模糊,输入原图像,点扩散函数以及噪声功率， 返回运动模糊化后的图像
    def blur(self, eps):
        image_fft=fft.fft2(self.image) #原图像傅里叶变换到频域中
        PSF_fft=fft.fft2(self.PSF)+eps
```

```
43.         self.blurred=fft.ifft2(image_fft*PSF_fft)       #频域中点扩散函数与图像相乘，逆变
    换到时域中得运动模糊图像
44.         self.blurred=np.abs(fft.fftshift(self.blurred)) #将图像四角移动至中心
45.     # 对模糊图像采用逆滤波进行复原，输入模糊图像,点扩散函数以及噪声功率，返回复原后的图像
46.
47.     def inverse(self, eps):
48.         image_fft=fft.fft2(self.blurred)    #模糊图像傅里叶变换到频域中
49.         PSF_fft=fft.fft2(self.PSF)+eps
50.         result=fft.ifft2(image_fft/PSF_fft) #频域中模糊图像与点扩散函数相除，逆变化到时域
    中获得复原图像
51.         result=np.abs(fft.fftshift(result)) #将图像搬移至中心位置
52.         self.inverse_result=result
53.
54.    # 对模糊图像采用维纳滤波进行复原，输入模糊图像,点扩散函数,噪声功率以及参数 K(默认为 0.01)
55.     def wiener(self,eps,K=0.01):
56.         image_fft=fft.fft2(self.blurred)                    #模糊图像傅里叶变换到频域中
57.         PSF_fft=fft.fft2(self.PSF)+eps
58.         PSF_fft_1=np.conj(PSF_fft)/(np.abs(PSF_fft)**2+K)#维纳滤波器点扩散函数
59.         result=fft.ifft2(image_fft*PSF_fft_1)               #频域中点扩散函数与模糊图像相
    乘，逆变换到时域中得复原图像
60.         result=np.abs(fft.fftshift(result))                 #将图像搬移至中心
61.         self.wiener_result=result
62.
63.     #程序结果显示子函数
64.     def show(self):
65.         graph.figure(1)
66.         graph.xlabel("Original Image")
67.         graph.gray()
68.         graph.imshow(self.image)      #显示原图像
69.         graph.figure(2)
70.         graph.gray()
71.         self.motion(angle,dist) #获得当前参数下的运动点扩散函数
72.         self.blur(eps)          #运动模糊
73.         self.inverse(eps)       #逆滤波求解
74.         self.wiener(eps)        #维纳滤波器求解
75.         #打印并保存结果图像
76.         graph.subplot(231)
77.         graph.xlabel("Blurred Image")
78.         graph.imshow(self.blurred)
79.         scipy.misc.imsave('Blurred_image.png',self.blurred)
80.         graph.subplot(232)
81.         graph.xlabel("Inverse_deblurred Image")
82.         graph.imshow(self.inverse_result)
83.         scipy.misc.imsave('Inverse_deblurred_image.png',self.inverse_result)
84.         graph.subplot(233)
85.         graph.xlabel("Wiener_deblurred Image(K=0.01)")
86.         graph.imshow(self.wiener_result)
87.         scipy.misc.imsave('Wiener_deblurred_image.png',self.wiener_result)
88.         #给运动模糊图像添加噪声,采用 standard_normal 产生高斯噪声
89.         self.blurred=self.blurred+a*self.blurred.std()*np.random.standard_normal(self.
    blurred.shape)
90.         self.inverse(a+eps)     #噪声逆滤波求解
91.         self.wiener(a+eps)      #维纳滤波器求解
92.         #打印并保存结果图像
93.         graph.subplot(234)
94.         graph.xlabel("Motion+Noisy Blurred Image")
95.         graph.imshow(self.blurred)
96.         scipy.misc.imsave('Motion_Noisy_Blurred_image.png',self.blurred)
```

```
97.         graph.subplot(235)
98.         graph.xlabel("Motion+Noisy Inverse_deblurred Image")
99.         graph.imshow(self.inverse_result)
100.        scipy.misc.imsave('Motion_Noisy_Inverse_deblurred_image.png',self.inverse_resu
    lt)
101.        graph.subplot(236)
102.        graph.xlabel("Motion+Noisy Wiener_deblurred Image(K=0.01)")
103.        graph.imshow(self.wiener_result)
104.        scipy.misc.imsave('Motin_Noisy_Wiener_deblurred_image.png',self.wiener_result)
105.        graph.show()
106.
107.if __name__ == '__main__':
108.    func=restoration()
109.    func.show()
```

Code 2.2.3.1 图像复原 Python 代码实现

③ 图像滤波去噪算法结果展示

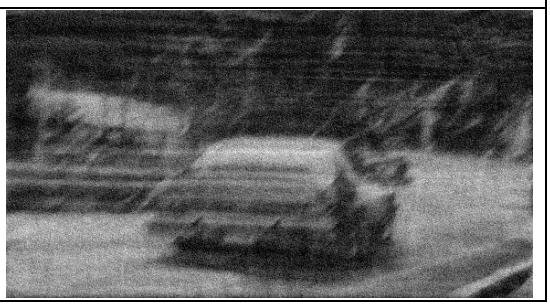
运动模糊化图像	运动模糊+高斯噪声图像
	
逆滤波复原	**逆滤波复原**
	
维纳滤波器复原	**维纳滤波器复原**
	

Fig 2.2.3.1 图像复原图像

④ 图像滤波去噪结果分析

根据代码结果我们可以看到，对于无噪声的运动模糊图像，当我们知道其运

动模糊的点扩散函数时，可以基本无损地复原得到其清晰图像。但是对于有噪声的图像，即便是知道点扩散函数，通过逆滤波变化得到的复原图像虽然没有了速度模糊的重影情况，但是其复原图像上具有明显的噪声点，而且图像的而对比度也明显下降。

再考虑维纳滤波器，维纳滤波器在处理无噪声图像时，其处理效果不及逆滤波法，得到的复原图像仍然具有较为明显的重影现象，但是相比于最初的运动模糊图像，其模糊程度已经得到了较明显的改善。但是对于加入了噪声的运动模糊图像，维纳滤波器具有一定的去噪作用，相比于逆滤波方法，能够输出对比度较为明显的两张图像，并且一定程度上地去除噪声点。

3 目标检测算法

3.1 YOLO v3 目标检测算法简介

YOLO v3(You Only Look Once)是一种十分高效的目标检测算法，其是一种基于深度卷积神经网络的目标检测算法。下面对 YOLO v3 算法做简要介绍。

3.1.1 YOLO v3 基本思想

首先 YOLO v3 算法通过特征提取网络对输入图像进行特征提取，得到一定尺寸的 feature map。接着，将输入图像等分为 13*13 个 grid cell，若 ground truth 中某个目标中心坐标落与某一 grid cell 中，则由该 grid cell 对该目标进行预测。YOLO 算法的一个特有属性是 bounding box，不同的 bounding box 对应不同的可能目标方位；每个 grid cell 均会按照预测固定数量的 bounding box 进行预测，并依据极大值抑制的原则，选择不同 bounding box 中 ground truth 的 IOU 最大值对应的 bounding box 作为该目标的预测值。YOLO v3 的输出值是一个数组，该输出值表征了检测目标的锚框坐标信息，目标类别及相应的检测置信度。

3.1.2 关于 bounding box

关于 bound box 的预测，YOLO v3 算法的 bounding box 的坐标预测方式延续了 YOLO v2 的做法，其主要依据以下公式进行坐标预测：

$$b_x = \sigma(t_x) + c_x; b_y = \sigma(t_y) + c_y$$
$$b_\omega = p_\omega e^{t_\omega}; b_h = p_h e^{t_h}$$

t_x, t_y, t_ω, t_h 即是 YOLO v3 模型的预测输出，c_x, c_y 分别表示 grid cell 的坐标，p_ω, p_h 分别表示预测前 bounding box 的尺寸，b_x, b_y, b_ω, b_h 分别对应预测得到的 bounding

box 的中心的坐标与尺寸。

3.1.3 关于类别预测

YOLO v3 的类别预测主要将 YOLO v2 中的单标签分类改进为多标签分类，因此网络结构上就将原来用于单标签多分类的 softmax 层换成用于多标签多分类的逻辑回归层。原来分类网络中的 softmax 层均假设一张图像或一个对象仅属于一个类别，但在一些复杂场景下，一个目标对象可能具有多重属性，这就涉及到多标签分类技术。多标签分类计数需要用逻辑回归层来对每个类别做二分类。逻辑回归层主要用到 sigmoid 函数，该函数可以将输入约束在 0 到 1 的范围内，因此当一张图像经过特征提取后的某一类输出经过 sigmoid 函数约束后，若置信度大于 0.5，则该目标可以归属为表示属于该类。

3.1.4 关于多 scale 融合预测

在 YOLO v2 算法中，其卷积神经网络中有一个网络称为 passthrough layer，假设最后提取了尺寸为 13*13 的 feature map，则 passthrough layer 的作用即是将前一层尺寸为 26*26 的 feature map 与本层尺寸为 13*13 的 feature map 进行连接；通过该操作加强 YOLO 算法对小目标检测的精确度。该思想在 YOLO v3 中得到了进一步加强，在 YOLO v3 中采用了类似 FPN 的 upsample 与融合做法，并在多个 scale 的 feature map 上做检测，对于小目标的检测效果有明显的提升。

3.2 卷积神经网络结构

YOLO v3 算法所采用的卷积神经网络结构称为 Darknet-53，该网络结构一方面基本采用全卷积，另一方面引入了残差结构。得益于残差结构的思想，深度学习中训练深层网络难度得以大大减小；也是基于此，YOLO v3 算法能够将网络做到 53 层，其相应的目标检测精度也提升比较明显。Darknet-53 的具体结构如下图所示：

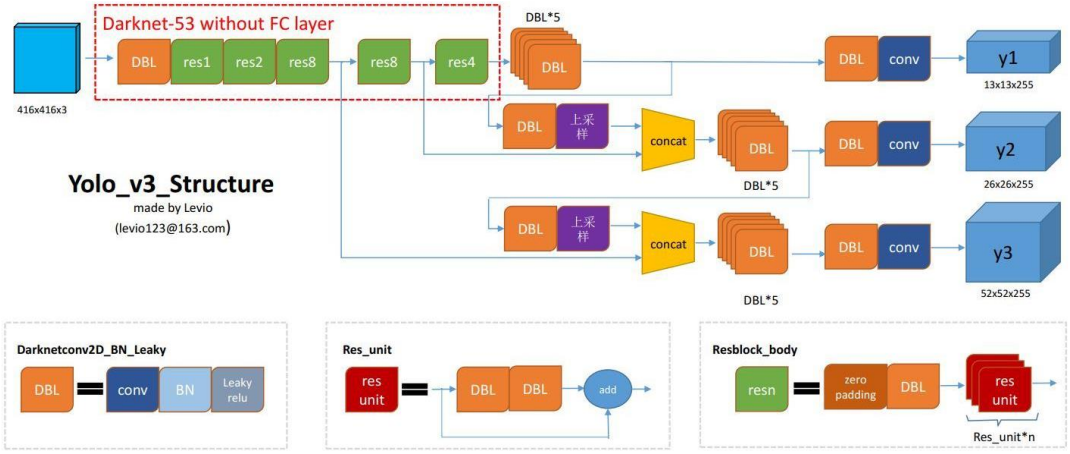


Fig 3.2.1 Darknet-53 网络结构示意图

3.2.1 网络结构：backbone

YOLO v3 网络结构中是不存在池化层和全连接层。在前向传播过程中，张量的尺寸变换是通过改变卷积核的步长来实现的。与 YOLO v2 相同，backbone 都会将输出特征图缩小到输入的 1/32。所以，通常都要求输入图片是 32 的倍数。

相较于 YOLO v2 中通过最大池化的操作来实现张量尺寸变化，YOLO v3 则是通过增大卷积核的步长进行实现。YOLO v3 中进行了 5 次张量尺寸变化的操作，最终将 416x416 的图像输入转化为了 13x13 的张量输出。YOLO v2 算法所采用的 darknet-19 网络结构中不存在残差结构，而 YOLO v3 的 darknet-53 网络结构实现了在保证实时性的基础上对目标检测算法表现的优化。

3.2.2 网络结构：output

YOLO v3 算法的输出为 3 个不同尺度的 feature map，该改进点被称为 predictions across scales。该技术借鉴了 FPN(feature pyramid networks)，采用多尺度来对不同 size 的目标进行检测，越精细的 grid cell 就可以检测出越精细的物体。

YOLO v3 设定的是每个网格单元预测 3 个 bounding box，所以每个 bounding box 需要有(x, y, w, h, confidence)五个基本参数，以及 80 个目标类别的检测概率；之后通过上采样的方法，YOLO v3 实现了多尺度的 feature map。Darknet-53 中将 backbone 中间层的输出与后续网络层的上采样结果进行拼接，并将拼接后的处理结果作为 feature map。

3.3 YOLO v3 Python 代码实现

3.3.1 Python 代码相关依赖库

本目标检测算法的实现主要基于 Tensorflow 的 API，并依托于 OpenCV 平台的相关图像处理与摄像头数据读取函数实现，主要代码如下所示：

```
import argparse
import os
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
import scipy.io
import scipy.misc
import numpy as np
import pandas as pd
import PIL
import tensorflow as tf
from keras import backend as K
from keras.layers import Input, Lambda, Conv2D
from keras.models import load_model, Model
```

```
import cv2
from yad2k.models.keras_yolo import yolo_head, yolo_boxes_to_corners, preprocess_true_boxes, yolo_loss, yolo_body
import yolo_utils
```

Code3.3.1.1 代码相关依赖库

3.3.2 分类阈值过滤

分类阈值过滤部分针对整张图片进行操作，每个 bounding box 对应有 5 个锚框，每个锚框对应 80 个类别。通过分类阈值过滤函数，可以区分有无包含检测物体的锚框，并且返回每个锚框对应的最大可能类别，对应类别评分以及锚框位置尺寸通过阈值来过滤对象和分类的置信度，主要代码如下所示：

```
def yolo_filter_boxes(box_confidence, boxes, box_class_probs, threshold=0.6):

    """
    针对整张图片进行操作，每个像素点对应有 5 个锚框，每个锚框对应 80 个类别。
    经过此函数，可以区分有无包含检测物体的锚框，并且返回每个锚框对应的最大可能类别，对应类
 别评分以及锚框位置尺寸
    通过阈值来过滤对象和分类的置信度。

    参数：
        box_confidence  - tensor 类型，维度为（19,19,5,1）,包含 19x19 单元格中每个单元格预测的 5 个锚框中的所有的锚框的 pc （一些对象的置信概率）。
        boxes - tensor 类型，维度为(19,19,5,4)，包含了所有的锚框的（px,py,ph,pw）。
        box_class_probs - tensor 类型，维度为(19,19,5,80)，包含了所有单元格中所有锚框的所有对象( c1,c2,c3，···，c80 )检测的概率。
        threshold - 实数，阈值，如果分类预测的概率高于它，那么这个分类预测的概率就会被保留。
        box_scores - 锚框得分，维度为(19,19,5,80)

    返回：
        scores - tensor 类型，维度为(None,)，包含了保留了的锚框的分类概率。
        boxes - tensor 类型，维度为(None,4)，包含了保留了的锚框的(b_x，b_y，b_h，b_w)
        classess - tensor 类型，维度为(None,)，包含了保留了的锚框的索引

    注意："None"是因为你不知道所选框的确切数量，因为它取决于阈值。
          比如：如果有 10 个锚框，scores 的实际输出大小将是（10,）
    """

    # 第一步：计算锚框的得分
```

```
    # box_scores:锚框得分(最终函数返回值), box_confidence:指示锚框内是否含有物体, box_class_probs:指示锚框中每个类别对应的概率
    box_scores = box_confidence * box_class_probs

    # 第二步：找到最大值的锚框的索引以及对应的最大值的锚框的分数
    box_classes = K.argmax(box_scores, axis=-1)  # 寻找 box_scores 最后一个轴中最大元素的索引(也即一个锚框中 80 类中得分最高类的索引)
    box_class_scores = K.max(box_scores, axis=-1)  # 寻找 box_scores 最后一个轴中最大元素的得分(也即一个锚框中 80 类中的最高类别得分)

    # 第三步：根据阈值创建掩码
    filtering_mask = (box_class_scores >= threshold)  # 选择在阈值范围规定下的得分, 并将其编码为 True 或 False 以指示框中是否有目标

    # 对 scores, boxes 以及 classes 使用掩码
    # tf.boolean(tensor,mask):将掩码 mask 中为 True 的部分对应的张量 tensor 部分保存, 为 False 的部分对应的张量 tensor 部分舍弃
    scores = tf.boolean_mask(box_class_scores, filtering_mask)  # 获得满足阈值要求的锚框(认为锚框内存在检测物体), 其中最高类别得分
    boxes = tf.boolean_mask(boxes, filtering_mask)  # 获得满足阈值要求的锚框(认为锚框内存在检测物体)对应的尺寸(b_x, b_y, b_h, b_w)
    classes = tf.boolean_mask(box_classes, filtering_mask)  # 获得满足阈值要求的锚框(认为锚框内存在检测物体), 其中最高类别(对应的数字索引)

    return scores, boxes, classes
```

Code3.3.2.1 分类阈值过滤 Python 代码

输入参数：

box_confidence - tensor 类型，维度为（19,19,5,1）,包含 19x19 单元格中每个单元格预测的 5 个锚框中的所有的锚框的 pc （一些对象的置信概率）。

boxes - tensor 类型，维度为(19,19,5,4)，包含了所有的锚框的（px,py,ph,pw）。

box_class_probs - tensor 类型，维度为(19,19,5,80)，包含了所有单元格中所有锚框的所有对象(c1,c2,c3，•••, c80)检测的概率。

threshold - 实数，阈值，如果分类预测的概率高于它，那么这个分类预测的概率就会被保留。

box_scores - 锚框得分，维度为(19,19,5,80)

返回值：

scores - tensor 类型，维度为(None,)，包含了保留了的锚框的分类概率。

boxes - tensor 类型，维度为(None,4)，包含了保留了的锚框的(b_x, b_y, b_h, b_w)

classess - tensor 类型，维度为(None,)，包含了保留了的锚框的索引

3.3.2 非极大值抑制

非极大值抑制部分主要包含 yolo_non_max_suppression(scores, boxes, classes, max_boxes=10, iou_threshold=0.5)与 iou(box1, box2)两个函数，分别实现了非极大值抑制与锚框交并比的计算，主要代码如下所示：

```
def iou(box1, box2):
    """
    针对某两个锚框进行处理，计算出其两者之间的交并比，并返回交并比数值
    实现两个锚框的交并比的计算

    参数：
        box1 - 第一个锚框，元组类型，(x1, y1, x2, y2)
        box2 - 第二个锚框，元组类型，(x1, y1, x2, y2)

    返回：
        iou - 实数，交并比。
    """
    # 计算相交的区域的面积
    xi1 = np.maximum(box1[0], box2[0])  # 靠右的左边框
    yi1 = np.maximum(box1[1], box2[1])  # 靠上的下边框
    xi2 = np.minimum(box1[2], box2[2])  # 靠左的右边框
    yi2 = np.minimum(box1[3], box2[3])  # 靠下的上边框
    inter_area = (xi1 - xi2) * (yi1 - yi2)

    # 计算并集，公式为：Union(A,B) = A + B - Inter(A,B)
    box1_area = (box1[2] - box1[0]) * (box1[3] - box1[1])
    box2_area = (box2[2] - box2[0]) * (box2[3] - box2[1])
    union_area = box1_area + box2_area - inter_area

    # 计算交并比
    iou = inter_area / union_area

    return iou
```

Code3.3.2.1 锚框交并比计算函数 Python 代码

输入参数：

box1 - 第一个锚框，元组类型, (x1, y1, x2, y2)

box2 - 第二个锚框，元组类型, (x1, y1, x2, y2)

返回值：

iou - 实数，交并比

```
def yolo_non_max_suppression(scores, boxes, classes, max_boxes=10, iou_thresh
old=0.5):
    """
    为锚框实现非最大值抑制（ Non-max suppression (NMS)）
```

```
4.
5.      参数：
6.          scores - tensor 类型，维度为(None,)，yolo_filter_boxes()的输出
7.          boxes - tensor 类型，维度为(None,4)，yolo_filter_boxes()的输出，已缩放
    到图像大小（见下文）
8.          classes - tensor 类型，维度为(None,)，yolo_filter_boxes()的输出
9.          max_boxes - 整数，预测的锚框数量的最大值
10.         iou_threshold - 实数，交并比阈值。
11.
12.     返回：
13.         scores - tensor 类型，维度为(,None)，每个锚框的预测的可能值
14.         boxes - tensor 类型，维度为(4,None)，预测的锚框的坐标
15.         classes - tensor 类型，维度为(,None)，每个锚框的预测的分类
16.
17.     注意："None"是明显小于 max_boxes 的，这个函数也会改变 scores、boxes、classes
    的维度，这会为下一步操作提供方便。
18.
19.     """
20.     max_boxes_tensor = K.variable(max_boxes, dtype="int32")  # 用于
    tf.image.non_max_suppression()，缺省时 boxes 个数默认上限为 10
21.     K.get_session().run(tf.variables_initializer([max_boxes_tensor]))  # 初始
    化变量 max_boxes_tensor
22.
23.     # 使用 tf.image.non_max_suppression()来获取与我们保留的框相对应的索引列表
24.     # tf.image.non_max_suppression(boxes,scores,max_output_size,iou_threshold
    ,score_threshold,name),该函数用于实现非最大值抑制
25.     # boxes:锚框位置尺寸信息，scores:锚框对应类别(已完成最大可能类别选择)的得
    分，max_output_size:非最大值抑制最多选择框数，iou_threshold:交并比阈值
26.     # 函数返回值为各个框的索引值，是一个一维数组(索引值从 0 开始)
27.     nms_indices = tf.image.non_max_suppression(boxes, scores, max_boxes, iou_
    threshold)
28.
29.     # 使用 K.gather()来选择保留的锚框
30.     # K.gather(reference, indices)，该函数用于根据索引在张量中找出对应子张量
31.     # reference:被搜寻张量，indices:索引值
32.     scores = K.gather(scores, nms_indices)  # 寻找非最大值抑制后的锚框分数
33.     boxes = K.gather(boxes, nms_indices)  # 寻找非最大值抑制后的锚框位置尺寸
34.     classes = K.gather(classes, nms_indices)  # 寻找非最大值抑制后的锚框类别
35.
36.     return scores, boxes, classes
```

Code 3.3.2.2 锚框非极大值抑制 Python 代码

输入参数：

scores - tensor 类型，维度为(None,), yolo_filter_boxes()的输出

boxes - tensor 类型，维度为(None,4), yolo_filter_boxes()的输出，已缩放到图像大小

classes - tensor 类型，维度为(None,), yolo_filter_boxes()的输出

max_boxes - 整数，预测的锚框数量的最大值

iou_threshold - 实数，交并比阈值

返回值：

scores - tensor 类型，维度为(,None)，每个锚框的预测的可能值

boxes - tensor 类型，维度为(4,None)，预测的锚框的坐标

classes - tensor 类型，维度为(,None)，每个锚框的预测的分类

3.3.3 对所有锚框进行过滤

该函数实现了将 YOLO v3 编码的输出值，即一系列锚框转化为预测框及其对应的置信度、坐标与类别，主要代码如下所示：

```
1.  def yolo_eval(yolo_outputs, image_shape=(720., 1280.),
2.                max_boxes=10, score_threshold=0.6, iou_threshold=0.5):
3.      """
4.      将 YOLO 编码的输出（很多锚框）转换为预测框以及它们的分数，框坐标和类。
5.
6.      参数：
7.          yolo_outputs - 编码模型的输出（对于维度为（608,608,3）的图片），包含 4 个 tensors 类型的变量：
8.                              box_confidence :  tensor 类型，维度为 (None, 19, 19, 5, 1)
9.                              box_xy         :  tensor 类型，维度为 (None, 19, 19, 5, 2)
10.                             box_wh         :  tensor 类型，维度为 (None, 19, 19, 5, 2)
11.                             box_class_probs:  tensor 类型，维度为 (None, 19, 19, 5, 80)
12.         image_shape - tensor 类型，维度为（2,），包含了输入的图像的维度，这里是 (608.,608.)
13.         max_boxes - 整数，预测的锚框数量的最大值
14.         score_threshold - 实数，可能性阈值
15.         iou_threshold - 实数，交并比阈值
16.
17.     返回：
18.         scores - tensor 类型，维度为(,None)，每个锚框的预测的可能值
19.         boxes - tensor 类型，维度为(4,None)，预测的锚框的坐标
20.         classes - tensor 类型，维度为(,None)，每个锚框的预测的分类
21.     """
22.
23.     # 获取 YOLO 模型的输出
```

```
    box_confidence, box_xy, box_wh, box_class_probs = yolo_outputs

    # 中心点转换为边角
    boxes = yolo_boxes_to_corners(box_xy, box_wh)

    # 可信度分值过滤
    scores, boxes, classes = yolo_filter_boxes(box_confidence, boxes, box_class_probs, score_threshold)

    # 缩放锚框，以适应原始图像
    # 本代码中实现的是在(608,608,3)RGB 图像上的 YOLO，而输入输出图像并不一定是(608,608,3)，所以需要缩放锚框以匹配输入输出图像
    boxes = yolo_utils.scale_boxes(boxes, image_shape)

    # 使用非最大值抑制
    scores, boxes, classes = yolo_non_max_suppression(scores, boxes, classes, max_boxes, iou_threshold)

    return scores, boxes, classes
```

Code 3.3.3.1 对所有锚框进行过滤 Python 代码

输入参数：

yolo_outputs - 编码模型的输出（对于维度为（608,608,3）的图片），包含 4 个 tensors 类型的变量：

box_confidence：tensor 类型，维度为(None, 19, 19, 5, 1)

box_xy：tensor 类型，维度为(None, 19, 19, 5, 2)

box_wh：tensor 类型，维度为(None, 19, 19, 5, 2)

box_class_probs：tensor 类型，维度为(None, 19, 19, 5, 80)

image_shape - tensor 类型，维度为（2,），包含了输入的图像的维度，这里是(608.,608.)

max_boxes - 整数，预测的锚框数量的最大值

score_threshold - 实数，可能性阈值

iou_threshold - 实数，交并比阈值

返回值：

scores - tensor 类型，维度为(,None)，每个锚框的预测的可能值

boxes - tensor 类型，维度为(4,None)，预测的锚框的坐标

classes - tensor 类型，维度为(,None)，每个锚框的预测的分类

3.3.4 会话启动及目标检测

通过运行存储在 sess 的计算图，以预测 image_file 的边界框，并打印出预测的图与信息，主要代码如下所示：

```
def predict(sess, image_file, is_show_info=True, is_plot=True):
    """
    运行存储在 sess 的计算图以预测 image_file 的边界框，打印出预测的图与信息。
    参数：
        sess - 包含了 YOLO 计算图的 TensorFlow/Keras 的会话。
        image_file - 存储在 images 文件夹下的图片名称
    返回：
        out_scores - tensor 类型，维度为(None,)，锚框的预测的可能值。
        out_boxes - tensor 类型，维度为(None,4)，包含了锚框位置信息。
        out_classes - tensor 类型，维度为(None,)，锚框的预测的分类索引。
    """
    # 图像预处理
    image, image_data = yolo_utils.preprocess_image(image_file, model_image_size=(608, 608))

    # 运行会话并在 feed_dict 中选择正确的占位符.
    out_scores, out_boxes, out_classes = sess.run([scores, boxes, classes],
                                                  feed_dict={yolo_model.input: image_data, K.learning_phase(): 0})

    # 打印预测信息
    # if is_show_info:
    # print("在" + str(image_file) + "中找到了" + str(len(out_boxes)) + "个锚框。")

    # 指定要绘制的边界框的颜色
    colors = yolo_utils.generate_colors(class_names)
    # 在图中绘制边界框
    yolo_utils.draw_boxes(image, out_scores, out_boxes, out_classes, class_names, colors)

    # 保存已经绘制了边界框的图
    # image.save(os.path.join("out", image_file), quality=100)


    cv2.imshow('Image', np.array(image))

    # 打印出已经绘制了边界框的图
    if is_plot:
        output_image = scipy.misc.imread(os.path.join("out", image_file))
        plt.imshow(output_image)

    return out_scores, out_boxes, out_classes
```

Code 3.3.4.1 会话启动及目标检测

输入参数：

sess - 包含了 YOLO 计算图的 TensorFlow/Keras 的会话。

image_file - 存储在 images 文件夹下的图片名称

返回值：

out_scores - tensor 类型，维度为(None,)，锚框的预测的可能值。

out_boxes - tensor 类型，维度为(None,4)，包含了锚框位置信息。

out_classes - tensor 类型，维度为(None,)，锚框的预测的分类索引。

3.3.5 主程序

主程序通过 OpenCV 库函数实时读取摄像头信息并转化为标准的图片信息，将图像信息传入 YOLO v3 算法进行目标检测，并将目标检测结果转化为视频进行显示，主要代码如下所示：

```
def main()
    '''''定义分类，锚框与图像维度'''
    class_names = yolo_utils.read_classes("model_data/coco_classes.txt")  # 读入类别标签文本
    anchors = yolo_utils.read_anchors("model_data/yolo_anchors.txt")  # 读入锚框尺寸(w,h)，共五种锚框
    image_shape = (720., 1280.)
    yolo_model = load_model("model_data/yolov2.h5")  # 加载训练模型(该模块位于 keras 模块下)
    yolo_outputs = yolo_head(yolo_model.output, anchors, len(class_names))  # 将模型的输出转化为边界框
    scores, boxes, classes = yolo_eval(yolo_outputs, image_shape)  # 过滤锚框
    # 将视频文件转化为图像进行处理
    vc = cv2.VideoCapture("video/road_video_compressed.mp4")
    totalFrameNumber = vc.get(7)  # 获取总视频帧数
    c = 1
    step = 2  # 定义帧数间隔
    if vc.isOpened():
        rval, frame = vc.read()
    else:
        rval = False
        print('File open error!')
    if rval:
        while c < totalFrameNumber:
            rval, frame = vc.read()
            frame = cv2.resize(frame, (1280, 720))  #将图像转化为处理的标准尺寸
            if (c % step == 0):  # every 5 fps write frame to img
                # 计算需要在前面填充几个 0
```

```
                num_fill = int(len("0000") - len(str(1))) + 1
                # 对索引进行填充 f
                # 开始绘制，不打印信息，不绘制图
                out_scores, out_boxes, out_classes = predict(sess, frame, is_
    show_info=False, is_plot=False)
            c = c + 1
            cv2.waitKey(1)
    vc.release()
    cv2.destroyAllWindows()
```

Code 3.3.5.1 主程序

3.4 目标检测效果展示

基于 YOLO v3 算法，我们将第二部分预处理的图像出入到 YOLO v3 目标检测算法中，可以得到相应的智能车视觉信息。此处截取了部分目标检测算法的检测效果，具体视频数据实时处理结果可以参考我们的代码附件。

Fig 3.4.1 YOLO v3 效果展示图一

Fig 3.4.2 YOLO v3 效果展示图二

Fig 3.4.3 YOLO v3 效果展示图三

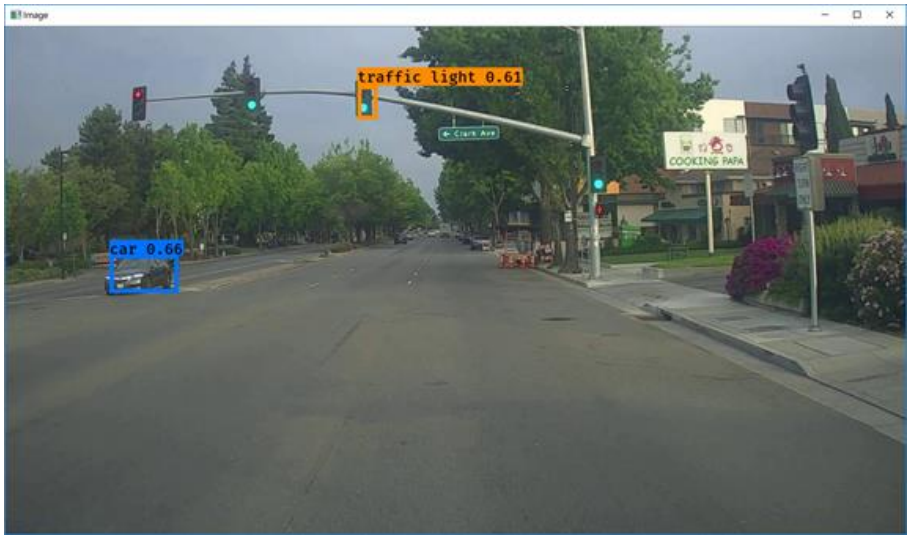


Fig 3.4.4 YOLO v3 效果展示图四

4 基于嵌入式控制器的改进

4.1 实时性

相比于电脑而言，常见的嵌入式处理器由于兼具有控制与运算等多重任务，计算性能上不可避免的有所下降。即使在 PC 端运行，当我们直接输入视频流时，程序的帧数也远低于 25 帧，如果直接移植至单片机或嵌入式处理器平台，程序的实时性无法保证，因此我们采取了关键帧处理方法，即在采集的视频流中进行采样，将采样得到的图像帧作为输入进行处理，抛弃其他帧，从而减少处理器的运算量。

4.1.1 合理性分析

真实环境中物体的运动可以认为是连续变化的，考虑到人眼的视觉暂留效应，对于快速运动的物体，人眼仍能够保留其影像 0.1 秒至 0.4 秒而非立刻消失。因此，当我们输入的图像较长时间用于算法处理时，其连贯性就被破坏，帧数降低。如对于每秒 60 帧视频流而言，第一帧的图像与第六十帧的图像可能仅有微小差别，而控制器的决策仅是在检测到某一现象出现和消失时进行的，中间的信息在大部分情况下可以忽略。因此当我们采集第一帧及相隔的某一帧作为关键帧时，中间缺失的部分可以通过插值等方法进行填补。

间隔帧数的选取也有着要求，对于交互端的控制者来说，其插值效果必须要满足人眼的视觉暂留效果才能够显得连续，而对于嵌入式处理器而言，其插值效果取决于处理器的控制速度、智能车的探测需求等，通常而言，车速越高时，单位时间内采集到的图像变化越显著；而控制速度越慢时，越需要提前进行决策控制，因此，这两种情况下，采样频率、插值参数都有所变化，可以认为存在相应的函数关系。

4.1.2 实现代码

```
totalFrameNumber = vc.get(7)   #获取总视频帧数
c=1
step=2     #定义帧数间隔
if vc.isOpened():
    rval,frame=vc.read()
else:
    rval=False
    print('File open error!')
if rval:
    while c<totalFrameNumber:
        rval,frame=vc.read()
        frame=cv2.resize(frame,(1280,720))  #将图像转化为处理的标准尺寸
        if (c%step == 0):                                #every 5 fps write frame to img
            #计算需要在前面填充几个 0
            num_fill = int( len("0000") - len(str(1))) + 1
            #对索引进行填充 f
            #开始绘制，不打印信息，不绘制图
            out_scores, out_boxes, out_classes = predict(sess,frame,is_show_info=False,is_plot=False)
        c=c+1
        cv2.waitKey(1)
vc.release()
cv2.destroyAllWindows()
```

Code 4.1.2.1 视觉测距 Python 代码实现

这里使用参数 step 定义帧数间隔，每相隔该数量的图片后，再进行一次目标检测，这样来降低处理器的运行负担，提高运算效率。但是 step 的具体值取决于摄像头的采样速度、使用 CPU/GPU 的性能等多种条件，不能一概而论，这里仅以在 PC 端运行时的参数为例。

4.2 距离检测算法

在时间允许的条件下，我们尝试了简单的距离检测，其基于的是在 YOLO v3 算法中已识别获得的锚框，其具体步骤分为标定与测量。

标定：将车载摄像头视为理想针孔相机模型，对于无穷远物，其所成像应当在相机后焦平面之上，因此物大小与物距大小之比与像大小与焦距大小之比相等，考虑到像大小与其在 CMOS/CCD 上所占有像素数量成正比，两者间所差的系数完全可并入焦距中，因此我们只需要对焦距进行标定，并储存有识别物体的实际大小信息，即可通过该方法得到物距离信息。

测量：测量时，首先读入锚框对象，并根据锚框的标签找到真实物具有的长宽尺寸 W，而像的大小可以通过锚框的长宽的像素值 P 确定，根据标定好的焦距 F,即可得到$\mathrm{D} = \frac{\mathrm{W}*\mathrm{F}}{\mathrm{P}}$,从而获得距离信息。

这样的方法存在着不足之处，当物体的姿态改变时，图像中锚框的像素值是会随之改变的，如对于道路上的车辆而言，正面与侧面显然具有不同尺寸，而这种方法无法考虑物体的姿态变化的情况，因此对于动态的目标效果较差，但对于静态的规则目标，如指示牌、路灯、行道线等，从车辆方向进行观察的情况通常是不会变化的，因此可以满足部分需求。

5 总结和展望

本项目主要希望能解决智能车对于环境探测的准确性与实时性问题，相比于其他项目，本项目更加偏向于软件方面，特别是将一些已有的技术整合利用到其中，并做出改进。

虽然在项目实施过程中，软件部分的内容可以进行方便的仿真、测试，但是缺少因硬件条件支持，很多的参数确定无法进行。在我们的程序中，识别任务采用了现有的网络结构，训练集来自于 Kitti，其对于硬件依赖性较小，但预处理模块、图像帧采样模块、距离检测模块等很多参数都取决于相机的内外参、车载处理器的性能甚至具体的行驶环境等，较小的硬件改动就会对参数造成影响。因此，即使在 PC 机上进行过仿真模拟，也很难保证在实际行驶过程中不出现问题。

但即使如此，我们的工作有一些参考意义，它为视觉探测的方法提供了一套

完整流程，对于参数调整等工作仅仅是局部的，对于不同的模块又有着更多的改进空间。如预处理中对于恶劣天气影响的分析，关键帧模块插值的具体实现等，都可以有进一步的探索空间。

总之，这次项目使得组内的成员收获了很多，也非常感谢老师及研究生学长在项目进行过程中对我们的指导帮助。