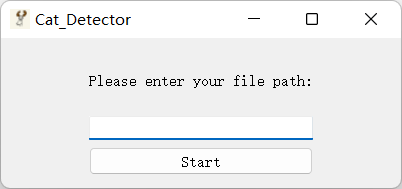
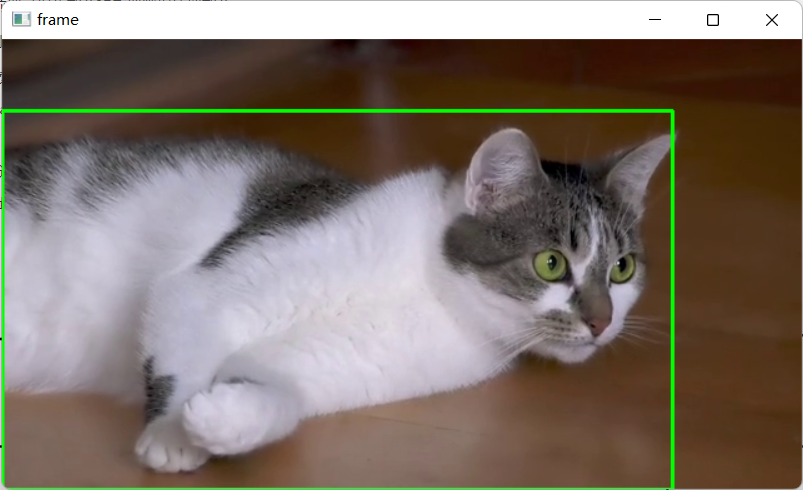
# Cat\_Detector目标检测系统

## 1.系统功能描述

本项目是一个基于R-CNN目标检测算法的实时目标检测系统，该系统可以检测并识别视频中的猫咪，只需在可视化界面中输入需要检测的视频路径并点击“start”即可开始进行目标检测



该系统具有简洁美观的可视化界面，并且操作简单易上手。可将本系统作为组件嵌入到流浪猫搜寻系统、猫咪看护机器人等，可极大的提升这些系统的性能。



## 2.系统设计

整个项目的项目结构如下

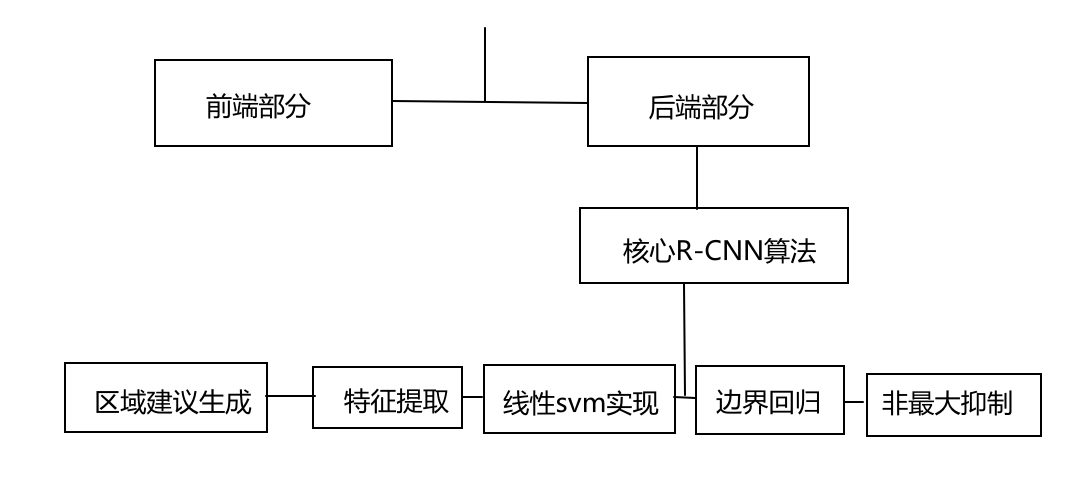
├──images # 测试图像以及视频  
├──data # 训练数据集  
├──models # 模型文件  
├──bbox\_reg.py # 边界框回归器训练  
├──cat\_detector.py # 检测器实现  
├──alex\_finetune.py # 卷积神经网络微调训练  
├──linear\_svm.py # 线性svm分类器训练  
├──sel\_search.py # 区域候选建议算法  
├──window.py # 前端可视化文件  
├──utils   
│ ├──create\_bbox\_dataset.py # 创建边界框回归数据集  
│ ├──create\_svm\_dataset.py # 创建分类器（二分类）数据集  
│ ├──create\_alex\_dataset.py # 创建微调数据集  
│ ├──custom\_batch\_sampler.py # 自定义批量采样器  
│ ├──custom\_bbox\_dataset.py # 自定义边界框回归数据集类  
│ ├──custom\_svm\_dataset.py # 自定义分类器数据集类  
│ ├──custom\_alex\_dataset.py # 自定义微调数据集类  
│ ├──custom\_hnm\_dataset.py # 难例挖掘  
│ ├──create\_voc\_cat.py # 创建cat类别数据集  
│ └──util.py # 辅助函数  
├──README.md # 帮助文档  
└──environments.yml # 环境配置

系统从整体上来说分为两个部分：前端可视化部分和后端逻辑处理部分。

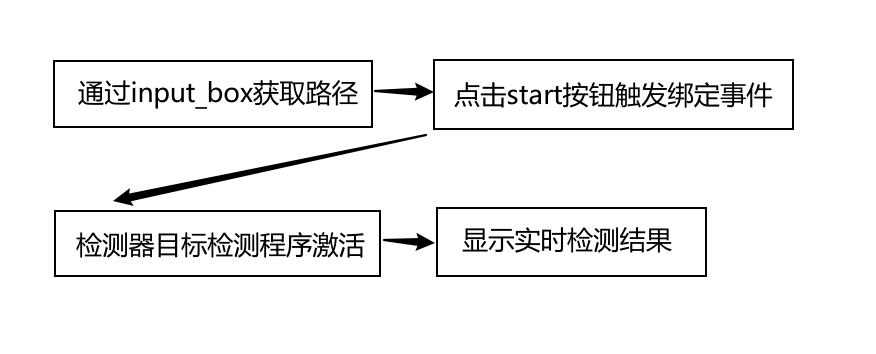
整个项目的核心部分（R-CNN目标检测算法）主要分为如下五个模块：

1. 区域建议生成：借助选择性搜索算法selectivesearch实现，生成类别独立的区域建议；
2. 特征提取：借助卷积神经网络alexnet实现，从每个区域建议中提取固定长度的特征向量；
3. 线性svm实现：输入特征向量，输出每个类别的成绩；
4. 边界回归：使用每个类指定的边界框回归器计算候选建议的坐标偏移；
5. 非最大抑制：对候选建议做非最大抑制，得到最终的候选建议；

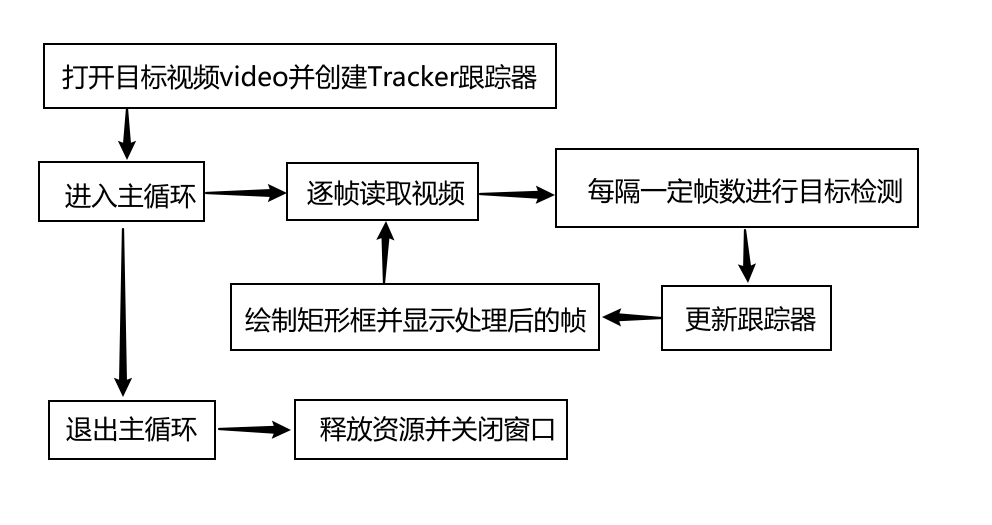
如下是整个系统的框图



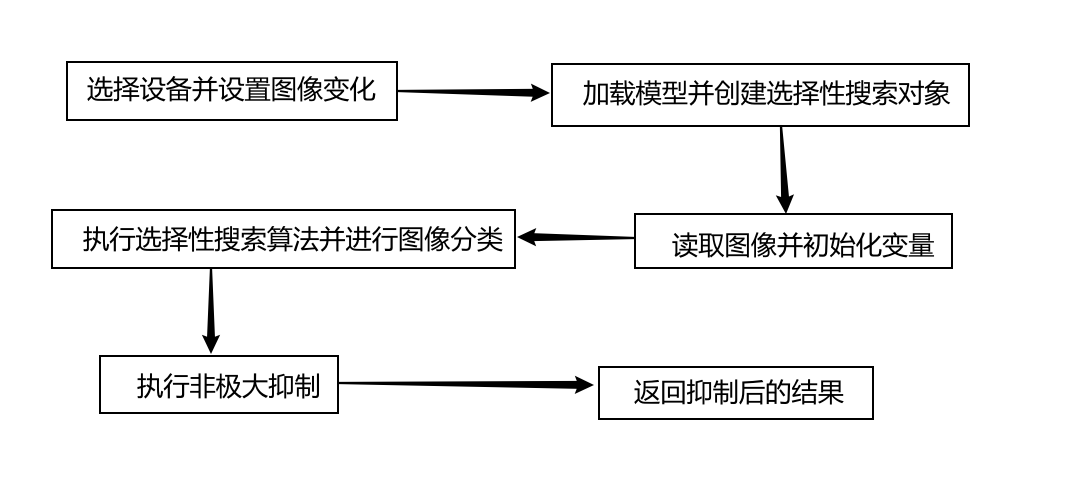
前端可视化部分使用PyQt5搭建，为用户提供了基本的输入框、启动程序、最小化、最大化以及关闭按钮。核心部分是前端与后端的交互，前端通过input\_box获取待检测视频路径。当用户按下start按钮触发start\_function()事件进而调用cat\_detector检测器的cat\_detector.vedio\_detect()方法，该方法将input\_box获取到的路径作为参数传递，对目标路径下的视频进行目标检测。



后端逻辑处理部分相对前端较复杂，主要实现对视频中猫的检测和跟踪，核心部分是基于一个while循环，用于逐帧读取视频并进行处理。当cat\_detector.vedio\_detect()方法被激活后，首先按照传入路径找到并打开目标视频，同时创建一个Tracker对象用于跟踪猫的位置。进入主循环后，逐帧读取视频，由于R-CNN算法本身速度以及电脑GPU性能的影响，一般选择每隔一定帧数进行一次猫的检测而非每帧都进行检测。检测之前需要先将待检测图像存储到指定路径，然后调用目标检测算法pic\_detect检测其中猫的位置，如果成功检测到猫的位置则将第一个检测到的猫的位置作为跟踪目标，同时初始化跟踪器。接下来使用跟踪器来更新猫的位置，如果更新成功则根据猫的位置在帧上绘制矩形框进行标记，最后显示处理后的帧并继续循环。



目标检测算法pic\_detect是整个后端逻辑处理部分的核心。简单来说，目标检测算法pic\_detect通过selective search获取候选框，然后将每个候选框输入到预训练的模型中进行分类，最后使用非极大值抑制来筛选出最终的猫的位置。

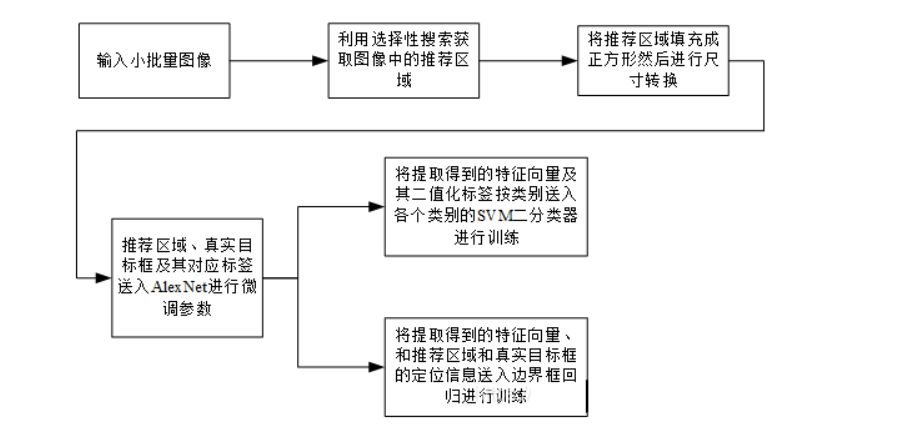


## 3.核心算法设计

整个项目的核心算法是R-CNN目标检测算法，基于该算法整个项目得以实现对图像或视频中猫的检测。R-CNN算法最初只是一个图像检测算法，但是使用一定的技巧可以使得R-CNN应用于一些简单的视频检测。

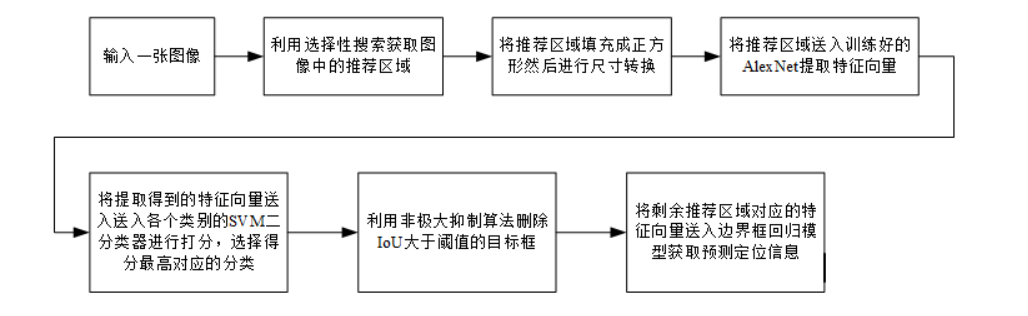
R-CNN的训练流程如下

* 训练阶段alexnet模型采用的是有监督预训练和特定领域内参数微调的训练方式；
* 提取特征完成后，还需要训练每个类别的svm分类器，完成分类任务；
* 除了分类任务，R-CNN还需要完成定位任务，将alexnet获得的特征向量按照类别分别送入x,y,w,h这四个分量回归器，利用梯度下度算进行训练每个分量回归器的权重。注意在这里特征向量的选择必须是与真实框(Ground Truth)之间IoU大于0.6的对应推荐区域提取出来的特征向量；



R-CNN的测试流程（即利用训练好的模型进行图像目标检测）如下

1. 先利用选择性搜索算法获取目标检测框，同时将目标框填充为正方形并转换为尺寸大小227\*227；
2. 通过alexnet提取图像特征；
3. 利用每个类别训练好的svm二分类器对alexnet提取得到的特征向量的每个类别进行打分，选择最高分为预测类别；
4. 将每个类别的特征向量送入每个类别的边界回归器进行定位预测，此时可能产生目标框重叠，故使用NMS删除IoU大于阈值的重复目标框；



综上，要使用python实现一个完整的基于R-CNN算法对图像中猫进行目标检测的详细流程如下：

1.准备数据集，本项目使用PASCAL VOC 2007数据集

2.数据集预处理

3.区域候选建议：使用选择性搜索算法实现

4.创建自定义数据集合和迭代器用于形成input数据送入网络；

5.卷积神经网络训练：

6.分类器训练：R-CNN完成alexnet卷积模型的微调后，额外使用svm二分类器，采用负样本挖掘的方法进行模型训练

7.边界框回归器训练：使用svm对候选建议进行分类后，使用对应类别的边界框回归器预测其坐标偏移值，进一步提高检测精度

8.目标检测器实现：利用训练好的R-CNN对输入图像进行检测：

本部分将按照上述设计流程依次介绍本项目中的用于检测图像中cat的R-CNN算法原理。

### 3.1 辅助函数

#### 3.1.1 区域候选算法

在R-CNN架构的第一步就是寻找推荐区域(Region Proposal)，推荐区域也被称为ROI(Region Of Interest)。获取推荐区域的方法主要有滑动窗口、规则块和选择性搜索：

* 滑动窗口：本质上就是穷举法，利用不同的尺度和长宽比把所有可能的大大小小的块都穷举出来，然后送去识别，识别出来概率大的就留下来。很明显，这样的方法复杂度太高，产生了很多的冗余候选区域，在现实当中不可行。
* 规则块：在穷举法的基础上进行了一些剪枝，只选用固定的大小和长宽比。但是对于普通的目标检测来说，规则块依然需要访问很多的位置，复杂度高。
* 选择性搜索：规则块的问题在于如何有效去除冗余的候选区域，考虑到冗余候选区域大多是发生了重叠，选择性搜索自底向上合并相邻的重叠区域从而减少冗余。

选择性搜索输入的是彩色图像，输出为候选的目标边界框集合。选择性搜索主要流程分为以下几个步骤：

1. 利用felzenszwalb分割算法获取初始区域集合R，同时初始化相似性度量集合S；
2. 遍历整个区域集合S，得到所有的相邻区域对(ri,rj)集合；
3. 遍历所有区域对(ri,rj)集合，计算区域ri和rj的相似性度量，同时将该相似性度量加入集合S；
4. 在S不为空的情况下，进行循环处理，在循环中
   1. 找到相似性度量最大对应的区域对(ri,rj)；
   2. 将区域ri和rj合并，记作rt；
   3. 从集合S中删除与区域ri相邻的其他区域的相似性度量；
   4. 从集合S中删除与区域rj相邻的其他区域的相似性度量；
   5. 计算区域rt与其相邻区域之间的相似性度量，将其加入集合S中；
   6. 将区域rt加入区域集合R；

本项目中实现区域候选算法借助的是opencv库，主要分为三部分

* 使用get\_selective\_search()获取选择性搜索算法对象
* 使用config()配置选择性搜索算法
* 使用get\_rects()获取候选区域

def get\_selective\_search(): # 获取选择性搜索算法对象  
 gs = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()  
 return gs  
  
def config(gs, img, strategy='q'): # 配置选择性搜索算法  
 gs.setBaseImage(img) # 设置基础图像  
 if (strategy == 's'):  
 gs.switchToSingleStrategy() # 设置单一策略  
 elif (strategy == 'f'):  
 gs.switchToSelectiveSearchFast() # 设置快速策略  
 elif (strategy == 'q'):  
 gs.switchToSelectiveSearchQuality() # 设置质量策略  
 else:  
 print(\_\_doc\_\_) # 打印脚本说明文档  
 sys.exit(1)  
  
def get\_rects(gs): # 获取候选区域  
 rects = gs.process()  
 rects[:, 2] += rects[:, 0] # 将候选区域的坐标转换为(x1, y1, x2, y2)的形式  
 rects[:, 3] += rects[:, 1]  
 return rects

在调用时先使用get\_selective\_search()创建selective search对象，然后使用对象的config()方法配置selective search对象(可选模式's','q','f')，最后使用对象的get\_rects()方法获取selective search返回的候选框列表rects。

#### 3.1.2 批量采集器类

pytorch的五大模块为：数据、模型、损失函数、优化器和迭代训练。其中的数据模块可细分为如下四个部分：

* 数据收集：收集样本和标注标签；
* 数据划分：将收集到的数据划分为训练集、验证集和测试集；
* 数据读取：该部分对应pytorch的Dataloader，而Dataloader包括Sampler和Dataset，其中Sampler的功能是生成索引index，Dataset的功能是根据生成的index读取样本及标签；
* 数据预处理：对应pytorch的transforms；

Dataset、Dataloader和Sampler三个类都是torch.utils.data 包下的模块(类)：

* Dataset是数据集的类，主要用于定义数据集
* Sampler是采样器的类，用于定义从数据集中选出数据的规则，比如是随机取数据还是按照顺序取等等
* Dataloader是数据的加载类，Dataset和Sampler会作为参数传递给Dataloader。Dataloader是对于Dataset和Sampler的进一步包装，用于实际读取数据，而Dataset和Sampler则负责定义。模型训练、测试所获得的数据是Dataloader传递的。

Sampler的作用在于生成相应的索引。在DataLoader类的初始化参数里有两种Sampler：sampler和batch\_sampler，都默认为None。前者的作用是生成一系列的index，而batch\_sampler则是将sampler生成的indices打包分组，得到一个又一个batch的index。

Sampler类是一个抽象父类，其主要用于设置从一个序列中返回样本的规则，即采样的规则。所有的采样器（无论是pytorch中已经实现的还是自定义的采样器）都继承自Sampler类。

Sampler是一个可迭代对象，使用step方法可以返回下一个迭代后的结果，因此其主要的类方法就是 iter 方法，定义了迭代后返回的内容。

无论是自定义的Sampler还是pytorch已经实现的Sampler，每次都只会返回一个索引，而在训练时是对批量的数据进行训练，该工作需要BatchSampler来完成。BatchSampler的作用就是将前面的Sampler采样得到的索引值进行合并，当数量等于一个batch大小后就将这一批的索引值返回

CustomBatchSampler类是一个自定义的批量采集器类，主要用于生成用于训练的批量样本。批量采集器根据给定的正样本数量、负样本数量以及正负样本的批量大小，生成相应的样本索引列表，并提供一个迭代器用于按批次获取样本。每个批次由一定数量的正样本和负样本组成，并且在迭代过程中，批量样本的顺序是随机的，有助于增加训练的随机性和多样性，从而提高模型的泛化能力。

class CustomBatchSampler(Sampler):  
  
 def \_\_init\_\_(self, num\_positive, num\_negative, batch\_positive, batch\_negative):  
 self.num\_positive = num\_positive # 正样本数量  
 self.num\_negative = num\_negative # 负样本数量  
 self.batch\_positive = batch\_positive # 正样本批量大小  
 self.batch\_negative = batch\_negative # 负样本批量大小  
  
 self.positive\_idx = torch.randperm(num\_positive).tolist() # 正样本索引  
 self.negative\_idx = torch.randperm(num\_negative).tolist() # 负样本索引  
  
 self.num\_iter = (num\_positive // batch\_positive) + (num\_negative // batch\_negative) # 迭代次数  
 self.batch = batch\_positive + batch\_negative # 批量大小  
  
 def \_\_iter\_\_(self):  
 sampler\_list = [] # 采样器列表  
 for i in range(self.num\_iter):  
 start\_pos = i \* self.batch\_positive # 起始位置  
 end\_pos = (i + 1) \* self.batch\_positive # 结束位置  
 positive\_batch = self.positive\_idx[start\_pos:end\_pos] # 正样本批量  
 negative\_batch = random.sample(self.negative\_idx, self.batch\_negative) # 负样本批量  
 batch = positive\_batch + negative\_batch   
 random.shuffle(batch) # 打乱  
 sampler\_list.extend(batch) # 添加到采样器列表  
 return iter(sampler\_list)  
  
 def \_\_len\_\_(self):  
 return self.num\_iter \* self.batch  
  
 def get\_num\_batch(self):  
 return self.num\_iter

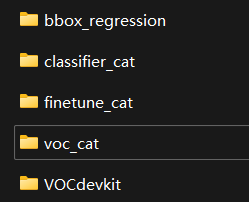
其中的\_\_iter\_\_(self)迭代方法在每次迭代中都会生成一个新的批量样本，最后返回一个迭代器对象。该方法首先生成正样本的索引列表，并随机化顺序，然后从负样本中随机选择一定数量的索引。然后将正样本和负样本索引列表合并成一个批量样本列表，并对其进行随机打乱。最后将批量样本列表扩展到一个总的样本索引列表中，并返回该列表的迭代器。

### 3.2 数据集创建

本部分需要实现一个能够检测图像中cat的目标检测器，在开始训练之前需要依次准备如下数据集：

1. PASCAL VOC 2007数据集
2. 从VOC数据集种提取cat类别数据
3. 创建微调数据集
4. 创建分类器数据集
5. 创建边界框回归数据集

最终的data目录下应该有以下形式的内容



#### 3.2.1 cat类别数据集

VOC 2007数据集可以在PASCAL VOC的官网上下载，下载之后执行create\_voc\_cat.py文件得到数据集voc\_cat，其格式如下

├── train  
│ ├── Annotations  
│ ├── cat.csv  
│ └── JPEGImages  
└── val  
 ├── Annotations  
 ├── cat.csv  
 └── JPEGImages

目录文件名与VOC数据集的目录文件名意义相同：

* Annotation 文件夹存放的是xml文件，该文件是对图片的解释，每张图片都对应一个同名的xml文件；
* JPEGImages 存放 .jpg格式的图片文件

该部分的核心代码如下，主要用于抽取数据集中的cat类别的数据，作为此次训练对象

if \_\_name\_\_ == '\_\_main\_\_':  
 samples = {'train': parse\_train\_val(cat\_train\_path), 'val': parse\_train\_val(cat\_val\_path)}  
 check\_dir(cat\_root\_dir) # 检查目录是否存在，若不存在则创建  
 for name in ['train', 'val']:  
 data\_root\_dir = os.path.join(cat\_root\_dir, name)  
 data\_annotation\_dir = os.path.join(data\_root\_dir, 'Annotations') # 保存类别Cat的样本标注文件的目录  
 data\_jpeg\_dir = os.path.join(data\_root\_dir, 'JPEGImages') # 保存类别Cat的样本图片文件的目录  
  
 check\_dir(data\_root\_dir)  
 check\_dir(data\_annotation\_dir)  
 check\_dir(data\_jpeg\_dir)  
 save\_cat(samples[name], data\_root\_dir, data\_annotation\_dir, data\_jpeg\_dir) # 保存类别Cat的样本图片和标注文件

其中：

* 主函数中先解析了训练集和验证集的样本列表，然后对于每一个集合，创建了一个对应的数据集目录，并调用save\_cat函数来保存cat类别的样本图片和标注文件。
* save\_cat函数将cat类别的样本图片和标注文件保存到指定的目录中。在这个函数中，首先遍历样本列表，对于每一个样本，将其标注文件和图片文件复制到指定的目录中。这里使用了shutil模块来完成文件复制的操作，同时还使用np.savetxt函数来将cat类别的样本列表保存到CSV文件中。

#### 3.2.2 微调数据集

执行create\_alex\_dataset.py文件得到finetune\_cat数据集，其格式如下

├── train  
│ ├── Annotations  
│ ├── cat.csv  
│ └── JPEGImages  
└── val  
 ├── Annotations  
 ├── cat.csv  
 └── JPEGImages

这部分主要是针对其中一个类别(cat)的数据，进行2分类的数据处理，为之后对Alexnet进行微调作准备

* 标注边界框：选择voc数据集中某一类别的图片的真实框体（在标注文件中给出）
* 候选建议：使用区域候选方法（选择性搜索算法）选择多个候选框
* 计算候选框与标注边界框的Iou
  + 正样本：IoU大于等于0.5；
  + 负样本：剩余的候选区域中，IoU大于0，小于0.5且其大小必须大于标注框的1/5；

该部分的核心代码比较简单，在代码文件中有注释故此处不再给出。需要注意的是，在复制文件之后，主函数会调用get\_selective\_search获取一个选择性搜索器，通过选择性搜索算法提取特征以便用于目标检测模型的训练和测试。

#### 3.2.3 分类器数据集

执行create\_svm\_dataset.py文件得到classifier\_cat数据集，其格式如下

├── train  
│ ├── Annotations  
│ ├── cat.csv  
│ └── JPEGImages  
└── val  
 ├── Annotations  
 ├── cat.csv  
 └── JPEGImages

分类器数据集为进一步进行svm二分类器的模型训练提供数据，其中

* 标注边界框：选择voc数据集中某一类别的图片的真实框体（在标注文件中给出）
* 候选建议：使用区域候选方法（选择性搜索算法）选择多个候选框
* 计算候选框与标注边界框的Iou
  + 正样本：标注边界框；
  + 负样本：IoU大于0，小于0.3且其候选建议的大小必须大于标注框的1/5（为了减少负样本的数量）；

这部分的整体代码逻辑和微调数据集的代码逻辑类似，其中需要注意：

* parse\_annotation\_jpeg函数：解析JPEG图片和对应的标注文件，并获取正负样本。该函数会首先读取JPEG图片，并使用选择性搜索算法获取候选建议。然后遍历标注文件中的边界框，对于每一个边界框，计算其与候选建议的IoU值，并根据IoU值的大小将其分类为正样本或负样本。其中，正样本是标注边界框，负样本是IoU大于0，小于等于0.3，且大小必须大于标注框的1/5的候选建议。

#### 3.2.4 边界框回归数据集

利用微调数据集的正样本（IoU>=0.5），再进一步提取IoU>0.6的候选建议(数据集都是正例，故loss都是0)。执行create\_bbox\_dataset.py文件得到bbox\_regression数据集，其格式如下

├── bndboxs  
├── JPEGImages  
├── positive  
├── cat.csv

主函数主要通过遍历样本列表，将每一个样本的标注文件和JPEG图片复制到指定的目录中，并从训练集正样本中提取IoU大于0.6的边界框。程序的输出是一个边界框回归器数据集，可以用于训练边界框回归器模型。

### 3.3 卷积神经网络训练

#### 3.3.1 微调数据集类

Dataset的作用是保存数据集的图片和相应的标签，通过索引能够完成图片的加载以及预处理、标签的加载以及预处理。Datasets是后续构建Dataloader工具函数的实例参数之一。

Dataset 是抽象类，所有自定义的 Dataset 都需要继承该类，并且重写\_\_getitem()\_\_方法和\_\_len\_\_()方法（不覆写这两个方法会直接返回错误） 。\_\_getitem()\_\_方法的作用是接收一个索引，返回索引对应的样本和标签，这是我们自己需要实现的逻辑。\_\_len\_\_()方法是返回所有样本的数量。

CustomFinetuneDataset类继承自Dataset，主要用于加载和处理用于微调alex模型的数据集。CustomFinetuneDataset根据给定的数据集根目录，读取图像和标注文件，提取出目标的坐标和数量，并提供索引方法用于获取图像和目标。通过这个自定义数据集类，可以方便地加载和使用用于微调的数据集，并在训练过程中进行数据增强等操作。

CustomFinetuneDataset类有以下方法和属性

* \_\_init\_\_初始化方法，在初始化的过程中，依次解析csv文件以获取所有样本的名称，并读取每个样本的图像以及相应的正负样本标注文件的路径。同时解析标注文件，提取出每个样本中目标的数量和坐标，并计算出正负样本的总数

def \_\_init\_\_(self, root\_dir, transform=None):  
 samples = parse\_cat\_csv(root\_dir) # 解析csv文件，获取所有样本的名称  
 self.jpeg\_images = [  
 cv2.imread(os.path.join(root\_dir, 'JPEGImages', f"{sample}.jpg")) # 读取所有样本的图像  
 for sample in samples # 样本名称  
 ]  
 self.positive\_annotations = [  
 os.path.join(root\_dir, 'Annotations', f"{sample}\_1.csv") # 正样本的标注文件  
 for sample in samples  
 ]  
 self.negative\_annotations = [  
 os.path.join(root\_dir, 'Annotations', f"{sample}\_0.csv") # 负样本的标注文件  
 for sample in samples  
 ]  
 self.positive\_sizes, self.positive\_rects = self.\_parse\_annotations(self.positive\_annotations) # 解析正样本的标注文件  
 self.negative\_sizes, self.negative\_rects = self.\_parse\_annotations(self.negative\_annotations) # 解析负样本的标注文件  
 self.total\_positive\_num = int(np.sum(self.positive\_sizes)) # 正样本的总数  
 self.total\_negative\_num = int(np.sum(self.negative\_sizes)) # 负样本的总数  
 self.transform = transform # 数据增强

* \_parse\_annotation方法用于解析标注文件。该函数接收标注文件路径列表作为输入，在解析过程中遍历所有样本的标注文件，读取坐标信息，并将坐标存储在rects列表。根据样本中目标数量的不同分别对目标的坐标和数量进行处理，最后返回目标数量和坐标的列表

def \_parse\_annotations(self, annotations):  
 sizes, rects = [], [] # sizes: 样本中目标的数量，rects: 样本中目标的坐标  
 for annotation\_path in annotations: # 遍历所有样本的标注文件  
 rects\_array = np.loadtxt(annotation\_path, dtype=np.int, delimiter=' ') # 读取标注文件  
 if len(rects\_array.shape) == 1: # 如果样本中只有一个目标  
 if rects\_array.shape[0] == 4: # 如果目标的坐标是4个  
 rects.append(rects\_array) # 添加目标的坐标  
 sizes.append(1) # 目标的数量为1  
 else: # 如果目标的坐标不是4个  
 sizes.append(0) # 目标的数量为0  
 else: # 如果样本中有多个目标  
 rects.extend(rects\_array) # 添加目标的坐标  
 sizes.append(len(rects\_array)) # 目标的数量为目标的个数  
 return sizes, rects # 返回样本中目标的数量和坐标

* \_\_getitem\_\_索引方法，主要用于获取指定索引位置的样本和目标。该索引方法根据索引的大小与正样本的总数进行比较，确定是正样本还是负样本。如果是正样本，它通过索引获取正样本的坐标索引和图像索引，并从正样本坐标列表中获取相应的坐标。如果是负样本，它通过索引计算负样本的索引，并获取负样本的坐标索引和图像索引，然后从负样本坐标列表中获取相应的坐标。最后，它根据图像索引从图像列表中获取相应的图像，并根据坐标裁剪出目标区域，同时对数据进行增强操作。

def \_\_getitem\_\_(self, index):  
 if index < self.total\_positive\_num: # 如果索引小于正样本的总数  
 # 正样本  
 target = 1  
 rect\_index, image\_id = self.\_get\_rect\_index\_and\_image\_id(index, self.positive\_sizes) # 获取正样本的坐标索引和图像索引  
 xmin, ymin, xmax, ymax = self.positive\_rects[rect\_index] # 获取正样本的坐标  
 else:  
 # 负样本  
 target = 0  
 idx = index - self.total\_positive\_num # 获取负样本的索引  
 rect\_index, image\_id = self.\_get\_rect\_index\_and\_image\_id(idx, self.negative\_sizes) # 获取负样本的坐标索引和图像索引  
 xmin, ymin, xmax, ymax = self.negative\_rects[rect\_index] # 获取负样本的坐标  
  
 image = self.jpeg\_images[image\_id][ymin:ymax, xmin:xmax] # 获取样本的图像  
 if self.transform:  
 image = self.transform(image)  
 return image, target

* \_\_len\_\_：返回数据集的总样本数量，即正样本和负样本的总和。
* get\_positive\_num：返回正样本的数量。
* get\_negative\_num：返回负样本的数量。
* \_get\_rect\_index\_and\_image\_id方法，用于根据索引获取坐标索引和图像索引，在遍历过程中根据索引的范围确定目标所在的样本和目标在样本中的索引，然后返回目标的坐标索引和样本的图像索引

def \_get\_rect\_index\_and\_image\_id(self, index, sizes):  
 image\_id = 0  
 for i, size in enumerate(sizes): # 遍历所有样本  
 if np.sum(sizes[:i]) <= index < np.sum(sizes[:i + 1]): # 如果索引在当前样本中  
 rect\_index = index - np.sum(sizes[:i]) # 获取坐标索引  
 image\_id = i # 获取图像索引  
 break  
 return rect\_index, image\_id

#### 3.3.2 微调实现

微调模型采用的是pytorch提供的alexnet预训练模型，使用finetune的方法继承alexnet的网络模型和参数，微调alexnet网络使其能够实现2分类。

本项目中对alexnet预训练模型进行微调的主要步骤如下：

1. 加载模型及数据
   1. 确定训练设备（CPU or GPU）
   2. 加载数据集，返回dataloader和dataset\_size；
   3. 加载预训练alexnet模型；
   4. 获取分类器最后一层的输入特征数，这个特征数将作为全连接层的输入尺寸；
   5. 将分类器的输出特征数改为2，以适应微调的新任务，其中2是新任务的类别数；
   6. 转移模型到第一步选择的设备上；
2. 定义参数
   1. 定义交叉熵损失函数；
   2. 定义随机梯度下降优化器并设置学习率和momentum，用于更新模型参数；
   3. 定义StepLR调度器学习率调度器，用于动态调整学习率；
3. 训练模型并将训练好的最佳模型参数保存到文件中；

微调部分有两个主要的函数，load\_data数据加载函数和train\_model模型训练函数。

其中load\_data数据加载函数接收数据集的根目录路径以及批量大小和工作线程，返回训练和验证数据加载器以及它们各自的数据集大小。通过应用数据转换、创建自定义数据集和批次采样器，并使用DataLoader创建数据加载器，实现了数据的加载和准备，为训练和验证提供了经过预处理的数据，并通过采样器平衡了正负样本的数量。

def load\_data(data\_root\_dir, batch\_size=128, num\_workers=8):  
 # 定义应用于数据集中每个图像的转换  
 data\_transforms = transforms.Compose([  
 transforms.Resize((227, 227)), # 将图像大小调整为AlexNet所需的大小  
 transforms.RandomHorizontalFlip(), # 随机水平翻转  
 transforms.ToTensor(), # 将图像转换为张量  
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # 标准化图像  
 ])  
  
 # 使用CustomFinetuneDataset创建训练和验证数据集  
 train\_dataset = CustomFinetuneDataset(os.path.join(data\_root\_dir, 'train'), transform=data\_transforms)  
 val\_dataset = CustomFinetuneDataset(os.path.join(data\_root\_dir, 'val'), transform=data\_transforms)  
  
 # 使用CustomBatchSampler平衡每个批次中的正样本和负样本数量  
 train\_sampler = CustomBatchSampler(train\_dataset.get\_positive\_num(), train\_dataset.get\_negative\_num(), 32, 96)  
  
 # 使用DataLoader创建训练和验证数据加载器  
 train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, sampler=train\_sampler, num\_workers=num\_workers, drop\_last=True)  
 val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False, num\_workers=num\_workers, drop\_last=True)  
  
 # 返回一个带有训练和验证数据加载器及其各自数据集大小的字典  
 return {'train': train\_loader, 'val': val\_loader}, {'train': train\_sampler.\_\_len\_\_(), 'val': val\_dataset.\_\_len\_\_()}

train\_model模型训练函数就是常见的训练流程：在每个epoch中遍历训练集和验证集，通过遍历数据加载器中的数据，将输入数据和标签移动到指定设备上（GPU/CPU），通过前向传播获得模型的输出。根据是否为训练阶段，计算损失、进行反向传播和参数更新，同时累加损失和正确预测的样本数。然后计算当前阶段的平均损失和准确率并输出。

for epoch in range(num\_epochs):  
 print(f"Epoch {epoch}/{num\_epochs-1}") # 打印当前训练轮数  
 print("-" \* 10) # 打印分割线  
  
 for phase in ["train", "val"]: # 遍历训练和验证阶段  
 is\_train = phase == "train"  
 model.train(is\_train)  
 running\_loss, running\_corrects = 0.0, 0  
  
 for inputs, labels in data\_loaders[phase]: # 遍历数据加载器  
 inputs, labels = inputs.to(device), labels.to(device) # 将数据移动到device上(GPU/CPU)  
 optimizer.zero\_grad() # 梯度清零  
  
 with torch.set\_grad\_enabled(is\_train): # 设置梯度计算开关  
 outputs = model(inputs) # 前向传播  
 \_, preds = torch.max(outputs, 1) # 获取预测结果  
 loss = criterion(outputs, labels) # 计算损失  
  
 if is\_train:  
 loss.backward() # 反向传播  
 optimizer.step() # 更新参数  
  
 running\_loss += loss.item() \* inputs.size(0) # 累加损失  
 running\_corrects += torch.sum(preds == labels.data) # 累加正确预测的样本数  
  
 epoch\_loss = running\_loss / len(data\_loaders[phase].dataset) # 计算平均损失  
 epoch\_acc = running\_corrects.double() / len(data\_loaders[phase].dataset) # 计算平均准确率  
 print(f"{phase} Loss: {epoch\_loss:.4f} Acc: {epoch\_acc:.4f}") # 打印损失和准确率  
  
 if phase == "val" and epoch\_acc > best\_acc: # 如果是验证阶段且准确率更高  
 best\_acc = epoch\_acc # 更新最好的准确率  
 best\_model\_weights = copy.deepcopy(model.state\_dict()) # 更新最好的模型参数  
  
 if is\_train:  
 lr\_scheduler.step() # 更新学习率  
  
 print()

### 3.4 分类器训练

在完成对alexnet预训练模型的微调后，额外使用svm二分类器，采用负样本挖掘的方法进行模型训练(linear\_svm.py)

* 先取得和正例相同数目的负例作为训练样本；
* 进行第一轮训练并计算出准确率和loss，根据验证集准确率表现，同时判断是否可以存储为最好的模型参数；
* 进行难分辨负样本挖掘，将挖掘好的难分辨负样本数据加到负样本总数据中，进行下一轮训练；
* 经过多轮训练，存储一个最好的二分类器；

#### 3.4.1 二分类数据集类

\_\_init\_\_方法用于创建svm数据集类，加载图像和对应的正负样本标注。主要设计流程如下

1. 使用parse\_cat\_csv函数解析包含猫样本信息的CSV文件，获取样本列表；
2. 初始化存储图像、正样本和负样本的空列表；
3. 遍历样本列表，对每个样本进行处理；
4. 通过样本名称拼接图像路径，并借助OpenCV的imread函数读取图像，将图像添加到jpeg\_images列表中；
5. 构建正样本标注路径，加载标注信息到positive\_annotations数组中；
6. 检查正样本标注数组的形状
   * 如果是一维数组且长度为4，则表示只有一个正样本标注。创建包含边界框和图像ID的字典，将其添加到positive\_list列表中；
   * 如果正样本标注数组的形状是二维数组，则表示有多个正样本标注。对于每个正样本标注，创建包含边界框和图像ID的字典，将其添加到positive\_list列表中；
7. 类似地，处理负样本标注，将负样本的边界框和图像ID添加到negative\_list列表中；
8. 完成样本列表的遍历和处理后，数据集类的实例就包含了图像和对应的正负样本标注，可以在训练或测试中使用；

其相关代码实现如下

def \_\_init\_\_(self, root\_dir, transform=None):  
 samples = parse\_cat\_csv(root\_dir)  
 self.root\_dir = root\_dir  
 self.transform = transform # 转换器  
 self.jpeg\_images, self.positive\_list, self.negative\_list = [], [], [] # 图片，正样本，负样本  
 for idx, sample\_name in enumerate(samples): # 遍历所有样本  
 jpeg\_image = cv2.imread(os.path.join(root\_dir, 'JPEGImages', f"{sample\_name}.jpg")) # 读取图片  
 self.jpeg\_images.append(jpeg\_image) # 添加图片  
  
 positive\_annotation\_path = os.path.join(root\_dir, 'Annotations', f"{sample\_name}\_1.csv") # 正样本标注路径  
 positive\_annotations = np.loadtxt(positive\_annotation\_path, dtype=np.int, delimiter=' ') # 读取正样本标注  
 if len(positive\_annotations.shape) == 1 and positive\_annotations.shape[0] == 4: # 如果正样本标注只有一个  
 positive\_dict = {'rect': positive\_annotations, 'image\_id': idx}  
 self.positive\_list.append(positive\_dict) # 添加正样本  
 elif len(positive\_annotations.shape) > 1: # 如果正样本标注有多个  
 for positive\_annotation in positive\_annotations: # 遍历所有正样本  
 positive\_dict = {'rect': positive\_annotation, 'image\_id': idx}  
 self.positive\_list.append(positive\_dict) # 添加正样本  
  
 negative\_annotation\_path = os.path.join(root\_dir, 'Annotations', f"{sample\_name}\_0.csv") # 负样本标注路径  
 negative\_annotations = np.loadtxt(negative\_annotation\_path, dtype=np.int, delimiter=' ') # 读取负样本标注  
 if len(negative\_annotations.shape) == 1 and negative\_annotations.shape[0] == 4: # 如果负样本标注只有一个  
 negative\_dict = {'rect': negative\_annotations, 'image\_id': idx}  
 self.negative\_list.append(negative\_dict) # 添加负样本  
 elif len(negative\_annotations.shape) > 1: # 如果负样本标注有多个  
 for negative\_annotation in negative\_annotations: # 遍历所有负样本  
 negative\_dict = {'rect': negative\_annotation, 'image\_id': idx}  
 self.negative\_list.append(negative\_dict) # 添加负样本

二分类数据集类的\_\_getitem\_\_方法根据给定的索引从数据集中获取对应的图像和标签，核心算法原理如下：

1. 函数接收index索引，首先对索引进行判断：
   * if index < len(self.positive\_list): # 如果索引小于正样本数量  
      target = 1 # 标签为1  
      positive\_dict = self.positive\_list[index] # 获取正样本  
      cache\_dict = positive\_dict # 缓存字典  
     else:  
      target = 0 # 标签为0  
      idx = index - len(self.positive\_list) # 获取负样本索引  
      negative\_dict = self.negative\_list[idx] # 获取负样本  
      cache\_dict = negative\_dict # 缓存字典
   * 如果索引小于正样本的数量，则表示要获取的是正样本。依次将目标标签设置为1并获取对应索引的正样本字典并将其存储在缓存字典中；
   * 否则，表示要获取的是负样本，处理同上；
2. 接着从缓存字典中获取图像id和标注框的坐标。使用获取到的id和坐标从jpeg\_images列表中获取对应的图像片段
   * image\_id = cache\_dict['image\_id'] # 获取图片id  
     xmin, ymin, xmax, ymax = cache\_dict['rect'] # 获取标注框  
     image = self.jpeg\_images[image\_id][ymin:ymax, xmin:xmax] # 获取图片
3. 如果有transform操作则进行transform，否则返回图像、目标标签和缓存字典

#### 3.4.2 难例挖掘数据集类

自定义的难例挖掘类CustomHardNegativeMiningDataset继承自Dataset，主要用于难例挖掘任务中构建难例挖掘的Dataset。

类的构造函数\_\_init\_\_接收负样本列表、JPEG图像列表以及bool类型的transform，并将其存储在实例变量中。其中最核心的\_\_getitem\_\_方法根据给定的索引来获取数据集中的样本。

def \_\_getitem\_\_(self, index):  
 negative\_dict = self.negative\_list[index] # 获取负样本字典  
 xmin, ymin, xmax, ymax = negative\_dict['rect'] # 获取负样本矩形框坐标  
 image\_id = negative\_dict['image\_id'] # 获取负样本图像ID  
  
 image = self.jpeg\_images[image\_id][ymin:ymax, xmin:xmax] # 获取负样本图像  
  
 if self.transform:  
 image = self.transform(image)  
  
 target = 0 # 负样本标签为0  
  
 return image, target, negative\_dict

依次从负样本列表中获取对应索引的负样本字典、从负样本字典中获取负样本矩形框的坐标和负样本图像的id、使用负样本图像ID和矩形框的坐标从JPEG图像列表中获取对应的负样本图像，最后将获取到的负样本图像、目标标签以及负样本字典作为样本结果进行返回。

#### 3.4.2 分类器实现

线性svm分类器包含线性回归和折页损失hinge\_loss，其中hinge\_loss根据模型的输出和真实的标签来计算折页损失。它首先获取正确类别的得分，然后计算得分与模型输出的差值加上边界间隔，最后取所有样本的最大间隔项并计算平均值作为折页损失。

def hinge\_loss(outputs, labels):  
 """  
 折页损失计算  
 :param outputs: 大小为(N, num\_classes)  
 :param labels: 大小为(N)  
 :return: 损失值  
 """  
 num\_labels = len(labels)  
 corrects = outputs[range(num\_labels), labels].unsqueeze(0).T  
  
 # 最大间隔  
 margin = 1.0  
 margins = outputs - corrects + margin  
 loss = torch.sum(torch.max(margins, 1)[0]) / len(labels)  
  
 # # 正则化强度  
 # reg = 1e-3  
 # loss += reg \* torch.sum(weight \*\* 2)  
  
 return loss

线性svm分类器的训练采用了难例挖掘的方法，其实现流程如下：

1. 设置初始训练集，正负样本数比值为1:1（以正样本数目为基准）
2. 每轮训练完成后，使用分类器对剩余负样本进行检测，如果检测为正，则加入到训练集中
3. 重新训练分类器，重复第二步，直到检测精度开始收敛

训练过程的核心部分对训练和验证过程进行了迭代，根据当前是训练还是验证阶段，设置模型为训练模式或验证模式。

然后使用一个循环迭代数据加载器中的每个batch。在每个batch中，将输入数据和标签移动到设备（如GPU）上，并执行前向传播、计算损失、反向传播和参数更新等操作，同时记录损失和正确的样本数。

在每个epoch的末尾，计算平均损失和准确率，并打印结果。如果当前是验证阶段，并且当前的验证准确率超过了之前保存的最佳准确率，则更新最佳准确率和最佳模型权重。

for phase in ['train', 'val']:  
 is\_training = phase == 'train'  
 model.train(is\_training) # 设置模型为训练模式或验证模式  
 running\_loss, running\_corrects = 0.0, 0 # 记录损失和正确的样本数  
  
 data\_set = data\_loaders[phase].dataset # 获取数据集  
 print(f'{phase} - positive\_num: {data\_set.get\_positive\_num()} - negative\_num: {data\_set.get\_negative\_num()} - data size: {data\_sizes[phase]}') # 打印数据集信息  
  
 for inputs, labels, cache\_dicts in data\_loaders[phase]: # 获取一个batch的数据  
 inputs, labels = inputs.to(device), labels.to(device) # 将数据移动到GPU上  
  
 optimizer.zero\_grad() # 梯度清零  
 with torch.set\_grad\_enabled(is\_training): # 设置是否计算梯度  
 outputs = model(inputs) # 前向传播  
 \_, preds = torch.max(outputs, 1) # 获取预测结果  
 loss = criterion(outputs, labels) # 计算损失  
 if is\_training:  
 loss.backward() # 反向传播  
 optimizer.step() # 更新参数  
  
 running\_loss += loss.item() \* inputs.size(0) # 累加损失  
 running\_corrects += torch.sum(preds == labels.data) # 累加正确的样本数  
  
 if is\_training:  
 lr\_scheduler.step() # 更新学习率  
  
 epoch\_loss = running\_loss / data\_sizes[phase] # 计算平均损失  
 epoch\_acc = running\_corrects.double() / data\_sizes[phase] # 计算平均正确率  
  
 print(f'{phase} Loss: {epoch\_loss:.4f} Acc: {epoch\_acc:.4f}')  
  
 if phase == 'val' and epoch\_acc > best\_acc:  
 best\_acc = epoch\_acc  
 best\_model\_weights = copy.deepcopy(model.state\_dict())

下面这部分代码用于进行难例挖掘（负样本挖掘）。获取了训练集、剩余负样本列表、训练数据集的图像和transform后，使用剩余负样本列表构建一个新的数据集，并创建一个数据加载器用于遍历这个数据集的batch。在每个batch中，执行前向传播，计算预测结果，并记录正确的样本数。然后根据预测结果和缓存字典获取困难负样本和容易负样本，并将困难负样本添加到训练数据集的负样本列表中。最后计算剩余负样本的准确率并打印，同时更新训练数据集的负样本列表，并相应地更新训练数据集的大小。

train\_dataset = data\_loaders['train'].dataset # 获取训练数据集  
remain\_negative\_list = data\_loaders['remain'] # 获取剩余的负样本  
jpeg\_images = train\_dataset.get\_jpeg\_images() # 获取训练数据集的图片  
transform = train\_dataset.get\_transform() # 获取训练数据集的transform  
  
with torch.set\_grad\_enabled(False):  
 remain\_dataset = CustomHardNegativeMiningDataset(remain\_negative\_list, jpeg\_images, transform=transform) # 构建剩余负样本数据集  
 remain\_data\_loader = DataLoader(remain\_dataset, batch\_size=batch\_total, num\_workers=8, drop\_last=True) # 构建剩余负样本数据集的数据加载器  
  
 negative\_list = train\_dataset.get\_negatives() # 获取训练数据集的负样本  
 add\_negative\_list = data\_loaders.get('add\_negative', []) # 获取已经添加的负样本  
 running\_corrects = 0 # 记录正确的样本数  
  
 for inputs, labels, cache\_dicts in remain\_data\_loader:  
 inputs, labels = inputs.to(device), labels.to(device) # 将数据移动到GPU上  
  
 optimizer.zero\_grad() # 梯度清零  
 outputs = model(inputs) # 前向传播  
 \_, preds = torch.max(outputs, 1) # 获取预测结果  
 running\_corrects += torch.sum(preds == labels.data)  
  
 hard\_negative\_list, easy\_neagtive\_list = get\_hard\_negatives(preds.cpu().numpy(), cache\_dicts)  
 add\_hard\_negatives(hard\_negative\_list, negative\_list, add\_negative\_list)  
  
 remain\_acc = running\_corrects.double() / len(remain\_negative\_list)  
 print(f'remiam negative size: {len(remain\_negative\_list)}, acc: {remain\_acc:.4f}')  
  
 train\_dataset.set\_negative\_list(negative\_list) # 更新训练数据集的负样本  
 tmp\_sampler = CustomBatchSampler(train\_dataset.get\_positive\_num(), train\_dataset.get\_negative\_num(),  
 batch\_positive, batch\_negative)  
 data\_loaders['train'] = DataLoader(train\_dataset, batch\_size=batch\_total, sampler=tmp\_sampler,  
 num\_workers=8, drop\_last=True)  
 data\_loaders['add\_negative'] = add\_negative\_list  
 data\_sizes['train'] = len(tmp\_sampler) # 更新训练数据集的大小

### 3.5 边界回归训练

目标检测相较于传统的图像分类，不仅需要实现对目标的分类，还需要解决目标的定位问题（即获取目标在原始图像中的位置信息），R-CNN利用边界框回归来预测物体的目标检测框。

#### 3.5.1 边界回归数据集类

BBoxRegressionDataset数据集类同样继承自Dataset类，用于加载图像和边界框数据，并提供了对应的图像裁剪和边界框回归目标的计算。

BBoxRegressionDataset类主要有以下几个类函数、

* \_\_init\_\_构造函数用于初始化类实例，根据接收的根目录和transform加载图像、边界框和正样本数据。
* \_\_len\_\_函数返回数据集的样本数量。
* \_\_getitem\_\_函数获取数据集中的索引为index的样本。
  + 根据索引从self.box\_list中获取图像ID、正样本和边界框
  + 根据图像ID获取对应的图像，同时借助正样本的坐标从图像中裁剪出对应的区域
  + 计算正样本和边界框之间的平移比例和尺度比例，并返回处理后的图像和边界框回归目标

def \_\_getitem\_\_(self, index):  
 assert index < self.\_\_len\_\_(), f"Index {index} is out of range for dataset of size {self.\_\_len\_\_()}"  
  
 box\_dict = self.box\_list[index] # 获取box字典  
 image\_id = box\_dict['image\_id'] # 图片id  
 positive = box\_dict['positive'] # 正样本  
 bndbox = box\_dict['bndbox'] # 目标框  
  
 jpeg\_img = self.jpeg\_list[image\_id] # 获取图片  
 xmin, ymin, xmax, ymax = positive # 正样本坐标  
 image = jpeg\_img[ymin:ymax, xmin:xmax] # 截取正样本  
  
 if self.transform:  
 image = self.transform(image)  
  
 p\_w, p\_h = xmax - xmin, ymax - ymin # 正样本宽高  
 p\_x, p\_y = xmin + p\_w / 2, ymin + p\_h / 2 # 正样本中心点  
  
 xmin, ymin, xmax, ymax = bndbox # 目标框坐标  
 g\_w, g\_h = xmax - xmin, ymax - ymin # 目标框宽高  
 g\_x, g\_y = xmin + g\_w / 2, ymin + g\_h / 2 # 目标框中心点  
  
 t\_x, t\_y = (g\_x - p\_x) / p\_w, (g\_y - p\_y) / p\_h # 坐标偏移  
 t\_w, t\_h = np.log(g\_w / p\_w), np.log(g\_h / p\_h) # 宽高缩放  
  
 return image, np.array((t\_x, t\_y, t\_w, t\_h))

* get\_bndbox函数根据正样本和边界框计算IOU分数
  + 如果边界框是一维数组，则直接返回边界框；
    - 如果边界框是二维数组，则计算每个边界框与正样本的IOU分数，返回具有最高分数的边界框；

这个函数用于在存在多个边界框时选择最匹配的边界框

def get\_bndbox(self, bndboxes, positive):  
 if len(bndboxes.shape) == 1: # 只有一个bndbox  
 return bndboxes  
 else: # 多个bndbox  
 scores = util.iou(positive, bndboxes) # 计算IOU  
 return bndboxes[np.argmax(scores)]

#### 3.5.2 边界回归实现

使用svm对候选建议进行分类后，使用对应类别的边界框回归器预测其坐标偏移值，进一步提高检测精度

* 通过提高IoU阈值（>0.6）过滤正样本候选建议，将候选建议和标注边界框之间的转换看成线性回归问题，并通过岭回归（ridge regression）来训练权重w；
* 在读取alexnet的网络的基础上，冻结住alexnet的网络，并且取得alexnet中feature层的输出，送入一个线性的计算的模型，计算出4个输出，用于衡量偏移情况；
* 进行框体的非极大抑制的处理；

边界框回归训练使用的优化器和损失函数分别是Adam优化器和均方误差损失函数，基本的训练步骤为：

1. 调用load\_data加载bbox\_regression数据集的数据加载器；
2. 选择训练设备；
3. 调用get\_model函数加载CNN模型；
4. 计算输入特征的维度并设置输出特征的维度（用于边界框回归）；
5. 创建线性模型、均方误差损失函数、Adam优化器、学习率衰减器；
6. 开始训练，训练完毕后绘制损失曲线

### 3.6 检测器实现

利用已经实现的R-CNN目标检测算法遵循以下步骤可以实现一个对图像中猫的检测：

* 输入图像；
* 使用选择性搜索算法计算得到候选建议；
* 逐个计算候选建议：
  + 使用alexnet模型计算特征
  + 使用线性svm分类器计算得到分类结果
* 对所有分类为cat的候选建议执行非最大抑制；

这部分主要对应于cat\_detector中的pic\_detect部分，在流程图中也就对应R-CNN的测试流程。检测器代码的执行步骤主要为：

1. 准备阶段
   * 选择设备并将加载好的模型放在设备上；
   * 加载图像；
2. 创建并配置selective search对象，将其用于图像获取候选框列表
   * 对于每个候选框，从原图像中截取对应的区域并使用transform进行转换
   * 将处理后的图像输入模型，获取输出结果
3. 设置svm阈值并创建空列表，用于存储检测到的目标框和对应的分类score
   * 如果输出结果中最大值的索引为1，表示猫的概率最高，加入positive\_list列表
4. 执行非极大值抑制，对positive\_list中的矩形框进行非极大值抑制

def pic\_detect(img\_path): # 目标检测算法  
 # 设置设备和变换  
 # 通过torch.cuda.is\_available()判断是否可用CUDA加速，如果可用则使用GPU设备，否则使用CPU设备  
 device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')  
 # 创建一个变换对象transform，该对象包含了一系列的图像变换操作，包括将图像转换为PIL图像对象、调整大小为(227, 227)、随机水平翻转、转换为张量、以及归一化处理  
 transform = transforms.Compose([  
 transforms.ToPILImage(),  
 transforms.Resize((227, 227)),  
 transforms.RandomHorizontalFlip(),  
 transforms.ToTensor(),  
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  
 ])  
 # 加载模型和创建selective search对象  
 model = get\_model(device) # 调用get\_model(device)函数获取模型，并将其加载到设备上  
 gs = selectivesearch.get\_selective\_search() # 创建selective search对象gs  
 # 读取图像和初始化变量  
 img = cv2.imread(img\_path) # 读取图片，并将其保存为img变量  
 dst = copy.deepcopy(img) # 通过深拷贝创建dst变量，用于绘制矩形框  
 svm\_thresh = 0.60 # 设置SVM分类阈值  
 positive\_list = []  
 score\_list = []  
  
 # 执行selective search和分类  
 selectivesearch.config(gs, img, strategy='f') # 配置selective search对象，将其应用于图像 -- 最好不要使用s模式，尽管速度快但准确率太低了，f模式相对来说速度和准确率折中  
 rects = selectivesearch.get\_rects(gs) # 获取selective search返回的候选框列表rects  
 for rect in rects:  
 xmin, ymin, xmax, ymax = rect  
 rect\_img = img[ymin:ymax, xmin:xmax]  
 rect\_transform = transform(rect\_img).to(device) # 对于每个候选框，将其在原图像中截取出来，并应用之前定义的变换transform进行处理  
 output = model(rect\_transform.unsqueeze(0))[0] # 将处理后的图像输入模型，获取输出结果output  
  
 # 如果输出结果中最大值的索引为1，表示猫的概率最高，则将其加入到positive\_list列表  
 if torch.argmax(output).item() == 1:  
 probs = torch.softmax(output, dim=0).cpu().numpy()  
 if probs[1] >= svm\_thresh:  
 score\_list.append(probs[1])  
 positive\_list.append(rect)  
 # 执行非极大值抑制，对positive\_list中的矩形框进行非极大值抑制，得到抑制后的矩形框列表nms\_rects和得分列表nms\_scores  
 # nms\_rects, nms\_scores = nms(positive\_list, [1.0] \* len(positive\_list))  
 nms\_rects, nms\_scores = nms(positive\_list, score\_list)  
 # draw\_box\_with\_text(dst, nms\_rects, nms\_scores) # 绘制边框及其分类概率  
 # cv2.imshow('img', dst)  
 # cv2.waitKey(0)  
 return nms\_rects, nms\_scores

## 4.系统实现

通过前面的介绍，我们已经得到了一个可用于图像中cat检测的目标检测器cat\_detector，下面我们介绍系统的其他部分的实现。最后通过组合本部分介绍的其他组件，可以得到完整的实时目标检测系统。

### 4.1 视频检测器

要基于图像检测器实现一个能对视频中的猫做检测的视频检测器，基本的思想是将视频中的帧抽取出来，对帧中的物体进行图像检测后将检测结果返回，使用tracker跟踪器持续更新检测出物体的位置。

这里需要注意tracker的选择，一开始选择使用cv2的TrackerCSRT\_create对象发现效果并不好，绘制的矩形框与实际图像检测出的矩形框大小和位置都不同，后来切换为TrackerKCF\_create后检测效果有明显的提升。

# 创建一个cv2.TrackerCSRT\_create对象，用于跟踪猫的位置  
# tracker = cv2.TrackerCSRT\_create()  
tracker = cv2.TrackerKCF\_create()

实现视频检测器的核心就是在while循环中逐帧读取视频并进行处理

while True:  
 # 通过video.read()读取视频的下一帧  
 success, frame = video.read()  
 if not success: # 读取失败时跳出循环，读取成功则继续  
 break  
 if frame\_count % detection\_interval == 0: # 控制检测频率（每隔一定帧数进行猫的检测）  
 # 将当前帧保存为一张图片，并通过pic\_detect函数检测其中的猫的位置  
 img\_path = f'./images/frame\_{frame\_count}.jpg'  
 cv2.imwrite(img\_path, frame)  
 boxes, scores = pic\_detect(img\_path)  
 # 如果检测到猫的位置（len(boxes) > 0），则将第一个检测到的猫的位置作为跟踪目标，并使用tracker.init(frame, bbox)来初始化跟踪器  
 if len(boxes) > 0:  
 bbox = boxes[0] # 将第一个检测到的猫的位置作为跟踪目标  
 tracker.init(frame, bbox) # 使用tracker.init(frame, bbox)来初始化跟踪器  
 # 使用跟踪器来更新猫的位置  
 if bbox is not None:  
 ret, bbox = tracker.update(frame) # 通过ret, bbox = tracker.update(frame)获取更新后的位置  
 if ret: # 如果更新成功（ret为True），则根据猫的位置在帧上绘制矩形框，并用绿色标记  
 x1, y1, w, h = [int(i) for i in bbox] # 将bbox中的位置信息转换为整数  
 x2, y2 = x1 + w, y1 + h # 计算矩形框的右下角坐标  
  
 # factor = 1.5 # 矩形框大小的缩放因子  
 # x2, y2 = x1 + int(factor \* w), y1 + int(factor \* h)  
  
 cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)  
 # 使用cv2.imshow显示处理后的帧，如果按下键盘上的'q'键，循环退出  
 cv2.imshow('frame', frame)  
 if cv2.waitKey(1) & 0xFF == ord('q'):  
 break  
 frame\_count += 1

### 4.2 前端可视化

前端可视化界面的实现基于Pyqt5，关于可视化界面的搭建可以使用designer设计师拖拽组件实现，难点在于绑定按钮的激活函数

def start\_function(self):  
 # 获取视频路径  
 file\_path = self.input\_box.text()  
 # 若输入的视频路径为空，弹出提示  
 if file\_path == '':  
 self.label.setText('Error: Please enter your file path!')  
 return  
 # 若输入的视频路径不存在，弹出提示  
 elif not os.path.exists(file\_path):  
 self.label.setText('Error: The file path does not exist!')  
 return  
 # 若输入的不是视频文件，弹出提示  
 elif not file\_path.endswith('.mp4'):  
 self.label.setText('Error: Please enter a video file path!')  
 return  
 # 开始检测  
 else:  
 self.label.setText('Please wait for the result...')  
 cat\_detector.vedio\_detect(file\_path)  
 # 重新输入文件  
 self.label.setText('Please enter your file path:')

start\_function函数与start按钮绑定

# 创建Start按钮  
self.button = QPushButton('Start', self)  
self.button.clicked.connect(self.start\_function)

当检测到clicked行为被触发后，相应的start\_function函数会开始执行。start\_function函数主要检测从输入框中输入的检测函数是否符合标准，如果不合规则向用户提示相应错误并要求重新输入，仅当用户输入合法的视频文件路径时，调用cat\_detector类的vedio\_detect视频检测函数进行视频检测，并将结果返回在屏幕中

## 5.实验

### 5.1 单元测试

#### 5.1.1 批量采集器类

test用于演示一个数据集的自定义批量采样器的用法和功能

def test():  
 root\_dir = '../data/finetune\_cat/train' # 数据集根目录  
 train\_data\_set = CustomFinetuneDataset(root\_dir) # 数据集  
 train\_sampler = CustomBatchSampler(train\_data\_set.get\_positive\_num(), train\_data\_set.get\_negative\_num(), 32, 96) # 自定义批量采样器  
  
 print('sampler len: %d' % train\_sampler.\_\_len\_\_()) # 批量采样器长度  
 print('sampler batch num: %d' % train\_sampler.get\_num\_batch()) # 批量采样器批量数量  
  
 first\_idx\_list = list(train\_sampler.\_\_iter\_\_())[:128] # 批量采样器迭代器  
 # print('first\_idx\_list：',first\_idx\_list) # 打印批量采样器迭代器  
 print('positive batch: %d' % np.sum(np.array(first\_idx\_list) < 66517)) # 单次批量中正样本个数

分别输出了批量采集器的长度、数量以及单批量中正样本的个数

sampler len: 243328  
sampler batch num: 1901  
positive batch: 91

test2演示了如何使用transforms.Compose()对数据集进行转换，创建一个带有自定义采样器的DataLoader对象，并从DataLoader中检索成批的数据

def test2():  
 root\_dir = '../data/finetune\_cat/train'  
 transform = transforms.Compose([  
 transforms.ToPILImage(), # 转换为PIL图像  
 transforms.Resize((227, 227)), # 调整图像大小  
 transforms.ToTensor(), # 转换为张量  
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # 标准化  
 ])  
  
 train\_data\_set = CustomFinetuneDataset(root\_dir, transform=transform) # 数据集  
 train\_sampler = CustomBatchSampler(train\_data\_set.get\_positive\_num(), train\_data\_set.get\_negative\_num(), 32, 96) # 自定义批量采样器  
 data\_loader = DataLoader(train\_data\_set, batch\_size=128, sampler=train\_sampler, num\_workers=8, drop\_last=True) # 数据加载器  
  
 inputs, targets = next(data\_loader.\_\_iter\_\_()) # 获取数据  
 print(targets) # 打印标签  
 print(inputs.shape) # 打印输入张量形状

分别输出标签targets以及输入张量的形状

tensor([1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0,  
 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,  
 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0,  
 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,  
 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0,  
 1, 0, 1, 0, 0, 0, 0, 0])  
torch.Size([128, 3, 227, 227])

#### 5.1.2 微调数据集类

test演示如何使用CustomFinetuneDataset类来访问数据集中的特定项目，检索其目标标签，并将图像转换为PIL图像对象

def test(idx):  
 root\_dir = '../data/finetune\_cat/train'  
 train\_data\_set = CustomFinetuneDataset(root\_dir) # 创建数据集  
  
 print('positive num: %d' % train\_data\_set.get\_positive\_num(1)) # 获取正样本的数量  
 print('negative num: %d' % train\_data\_set.get\_negative\_num()) # 获取负样本的数量  
 print('total num: %d' % train\_data\_set.\_\_len\_\_()) # 获取样本的总数  
  
  
 image, target = train\_data\_set.\_\_getitem\_\_(idx) # 获取第idx个样本  
 print('target: %d' % target) # 输出标签  
  
 image = Image.fromarray(image) # 将numpy数组转换为PIL图像  
 print(image) # 输出图像  
 print(type(image)) # 输出图像的类型

输出如下

positive num: 20901  
negative num: 119832  
total num: 140733  
target: 0  
<PIL.Image.Image image mode=RGB size=157x189 at 0x215802D4F48>  
<class 'PIL.Image.Image'>

#### 5.1.3 二分类数据集类

如下代码使用CustomClassifierDataset类来访问数据集中的特定项目，检索它们的目标标签，检索相关的字典，并将图像转换成PIL图像对象

def test(idx):  
 root\_dir = '../data/classifier\_cat/val'  
 train\_data\_set = CustomClassifierDataset(root\_dir)   
  
 print('positive num: %d' % train\_data\_set.get\_positive\_num())   
 print('negative num: %d' % train\_data\_set.get\_negative\_num())  
 print('total num: %d' % train\_data\_set.\_\_len\_\_())  
  
 image, target, cache\_dict = train\_data\_set.\_\_getitem\_\_(idx)  
 print('target: %d' % target)  
 print('dict: ' + str(cache\_dict))  
  
 image = Image.fromarray(image)  
 print(image)  
 print(type(image))

输出如下

positive num: 190  
negative num: 100230  
total num: 100420  
target: 0  
dict: {'rect': array([ 21, 93, 97, 210]), 'image\_id': 44}  
<PIL.Image.Image image mode=RGB size=76x117 at 0x1A19647EC08>  
<class 'PIL.Image.Image'>

#### 5.1.4 边界回归数据集类

类似的，下面使用BBoxRegressionDataset类来访问数据集中的特定项目，检索图像和目标值，并使用transforms.Compose()应用数据转换，并输出关于数据集的长度以及检索的图像和目标的形状和数据类型。

def test():  
 transform = transforms.Compose([  
 transforms.ToPILImage(),  
 transforms.Resize((227, 227)),  
 transforms.RandomHorizontalFlip(),  
 transforms.ToTensor(),  
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  
 ])  
  
 data\_root\_dir = '../data/bbox\_regression'  
 data\_set = BBoxRegressionDataset(data\_root\_dir, transform=transform)  
  
 print(data\_set.\_\_len\_\_())  
 image, target = data\_set.\_\_getitem\_\_(10)  
 print(image.shape)  
 print(target)  
 print(target.dtype)

输出如下

13477  
torch.Size([3, 227, 227])  
[0.01932367 0. 0.02857337 0.03636764]  
float64

接着测试Dataloader的使用

def test2():  
 transform = transforms.Compose([  
 transforms.ToPILImage(),  
 transforms.Resize((227, 227)),  
 transforms.RandomHorizontalFlip(),  
 transforms.ToTensor(),  
 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  
 ])  
  
 data\_root\_dir = '../data/bbox\_regression'  
 data\_set = BBoxRegressionDataset(data\_root\_dir, transform=transform)  
 data\_loader = DataLoader(data\_set, batch\_size=128, shuffle=True, num\_workers=8)  
  
 items = next(data\_loader.\_\_iter\_\_())  
 datas, targets = items  
 print(datas.shape)  
 print(targets.shape)  
 print(targets.dtype)

输出如下

torch.Size([128, 3, 227, 227])  
torch.Size([128, 4])  
torch.float64

#### 5.1.5 难例挖掘数据集类

def test():  
 root\_dir = '../data/classifier\_cat/train'  
 data\_set = CustomClassifierDataset(root\_dir) # 读取训练数据集  
  
 negative\_list = data\_set.get\_negatives() # 获取负样本列表  
 jpeg\_images = data\_set.get\_jpeg\_images() # 获取JPEG图像列表  
 transform = data\_set.get\_transform() # 获取图像变换操作  
  
 hard\_negative\_dataset = CustomHardNegativeMiningDataset(negative\_list, jpeg\_images,  
 transform=transform) # 创建难例挖掘数据集对象  
 image, target, negative\_dict = hard\_negative\_dataset.\_\_getitem\_\_(100) # 获取第100个样本  
  
 print(image.shape) # 打印图像形状  
 print(target) # 打印标签  
 print(negative\_dict) # 打印负样本字典

输出如下

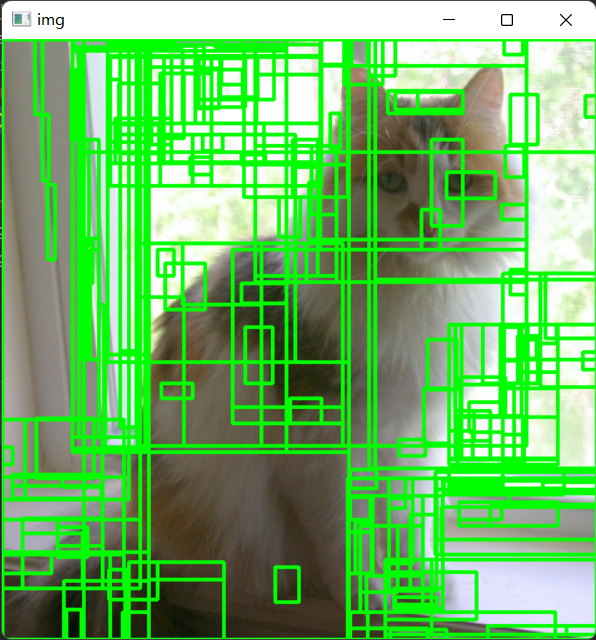
(63, 135, 3)  
0  
{'rect': array([207, 148, 342, 211]), 'image\_id': 0}

#### 5.1.6 区域候选算法

区域候选算法输入的是彩色图像，输出为候选的目标边界框集合

if \_\_name\_\_ == '\_\_main\_\_':  
 gs = get\_selective\_search()  
 img = cv2.imread('./images/test.jpg', cv2.IMREAD\_COLOR)  
 config(gs, img, strategy='s') # single模式  
 rects = get\_rects(gs)  
 # print(rects)  
 # 在原图上绘制候选区域  
 for x1, y1, x2, y2 in rects:  
 cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)  
 cv2.imshow('img', img)  
 cv2.waitKey(0)

将输出框绘制在原图上



### 5.2 核心算法评价

#### 5.2.1 卷积神经网络

微调部分的参数设置如下：

* load\_data：批量大小128，工作线程数8，即每次训练128个图像，其中32个正样本，96个负样本
* SGD优化器：学习率为0.001，动量为0.9
* 学习率调度器： step\_size=7，gamma=0.1，即每经过7个epoch，将学习率乘以0.1
* 训练epoch：25轮

先来看一下训练结果，可以看到模型在训练过程中出现了过拟合的情况，有条件的话（训练一次需要的时间太长了，所以我后面使用的还是此处训练好的模型并没有额外训练）可以尝试减小学习率、添加L2权重衰减或使用其他优化器进行优化

Epoch 0/24  
----------  
train Loss: 0.2108 Acc: 0.9088  
val Loss: 0.4699 Acc: 0.8319  
  
Epoch 1/24  
----------  
train Loss: 0.1271 Acc: 0.9481  
val Loss: 0.5266 Acc: 0.8305  
  
Epoch 2/24  
----------  
train Loss: 0.1048 Acc: 0.9572  
val Loss: 0.4723 Acc: 0.8414  
  
Epoch 3/24  
----------  
train Loss: 0.0897 Acc: 0.9640  
val Loss: 0.5704 Acc: 0.8302  
  
Epoch 4/24  
----------  
train Loss: 0.0814 Acc: 0.9668  
val Loss: 0.5699 Acc: 0.8359  
  
Epoch 5/24  
----------  
train Loss: 0.0716 Acc: 0.9716  
val Loss: 0.6233 Acc: 0.8337  
  
Epoch 6/24  
----------  
train Loss: 0.0679 Acc: 0.9726  
val Loss: 0.5797 Acc: 0.8406  
  
Epoch 7/24  
----------  
train Loss: 0.0544 Acc: 0.9788  
val Loss: 0.6551 Acc: 0.8342  
  
Epoch 8/24  
----------  
train Loss: 0.0530 Acc: 0.9790  
val Loss: 0.6711 Acc: 0.8369  
  
Epoch 9/24  
----------  
train Loss: 0.0516 Acc: 0.9794  
val Loss: 0.6709 Acc: 0.8361  
  
Epoch 10/24  
----------  
train Loss: 0.0495 Acc: 0.9802  
val Loss: 0.6953 Acc: 0.8350  
  
Epoch 11/24  
----------  
train Loss: 0.0503 Acc: 0.9799  
val Loss: 0.6824 Acc: 0.8376  
  
Epoch 12/24  
----------  
train Loss: 0.0491 Acc: 0.9806  
val Loss: 0.7024 Acc: 0.8336  
  
Epoch 13/24  
----------  
train Loss: 0.0473 Acc: 0.9808  
val Loss: 0.6969 Acc: 0.8344  
  
Epoch 14/24  
----------  
train Loss: 0.0484 Acc: 0.9804  
val Loss: 0.7073 Acc: 0.8349  
  
Epoch 15/24  
----------  
train Loss: 0.0475 Acc: 0.9812  
val Loss: 0.6925 Acc: 0.8361  
  
Epoch 16/24  
----------  
train Loss: 0.0460 Acc: 0.9816  
val Loss: 0.7225 Acc: 0.8337  
  
Epoch 17/24  
----------  
train Loss: 0.0475 Acc: 0.9816  
val Loss: 0.7099 Acc: 0.8355  
  
Epoch 18/24  
----------  
train Loss: 0.0468 Acc: 0.9816  
val Loss: 0.7137 Acc: 0.8348  
  
Epoch 19/24  
----------  
train Loss: 0.0483 Acc: 0.9807  
val Loss: 0.7020 Acc: 0.8345  
  
Epoch 20/24  
----------  
train Loss: 0.0460 Acc: 0.9819  
val Loss: 0.7125 Acc: 0.8345  
  
Epoch 21/24  
----------  
train Loss: 0.0471 Acc: 0.9815  
val Loss: 0.7070 Acc: 0.8343  
  
Epoch 22/24  
----------  
train Loss: 0.0464 Acc: 0.9813  
val Loss: 0.7074 Acc: 0.8349  
  
Epoch 23/24  
----------  
train Loss: 0.0475 Acc: 0.9815  
val Loss: 0.7037 Acc: 0.8335  
  
Epoch 24/24  
----------  
train Loss: 0.0467 Acc: 0.9813  
val Loss: 0.7080 Acc: 0.8346  
  
Training complete in 611m 28s  
Best val Acc: 0.841443

#### 5.2.2 SVM分类器

本部分的训练参数设置如下：

1. load\_data：每次训练128个图像，其中32个正样本，96个负样本
2. 优化器：学习率1e-4、动量0.9
3. 学习率调度器：每隔4轮衰减一次，参数因子α=0.1
4. 训练epoch：10

训练结果如下，可以看到，从第二个epoch开始，分类器的损失开始收敛

Epoch 0/9  
----------  
train - positive\_num: 186 - negative\_num: 186 - data size: 256  
train Loss: 1.2566 Acc: 0.4844  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.2150 Acc: 0.5266  
remiam negative size: 93081, acc: 0.6104  
Epoch 1/9  
----------  
train - positive\_num: 186 - negative\_num: 35812 - data size: 35968  
train Loss: 1.0272 Acc: 0.9295  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0776 Acc: 0.8865  
remiam negative size: 93081, acc: 0.9910  
Epoch 2/9  
----------  
train - positive\_num: 186 - negative\_num: 36101 - data size: 36224  
train Loss: 1.0070 Acc: 0.9726  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0807 Acc: 0.8844  
remiam negative size: 93081, acc: 0.9956  
Epoch 3/9  
----------  
train - positive\_num: 186 - negative\_num: 36124 - data size: 36224  
train Loss: 1.0037 Acc: 0.9816  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0818 Acc: 0.8871  
remiam negative size: 93081, acc: 0.9966  
Epoch 4/9  
----------  
train - positive\_num: 186 - negative\_num: 36133 - data size: 36224  
train Loss: 1.0031 Acc: 0.9852  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0821 Acc: 0.8853  
remiam negative size: 93081, acc: 0.9967  
Epoch 5/9  
----------  
train - positive\_num: 186 - negative\_num: 36138 - data size: 36224  
train Loss: 1.0030 Acc: 0.9849  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0831 Acc: 0.8845  
remiam negative size: 93081, acc: 0.9968  
Epoch 6/9  
----------  
train - positive\_num: 186 - negative\_num: 36141 - data size: 36224  
train Loss: 1.0030 Acc: 0.9851  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0825 Acc: 0.8869  
remiam negative size: 93081, acc: 0.9970  
Epoch 7/9  
----------  
train - positive\_num: 186 - negative\_num: 36143 - data size: 36224  
train Loss: 1.0029 Acc: 0.9861  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0828 Acc: 0.8848  
remiam negative size: 93081, acc: 0.9969  
Epoch 8/9  
----------  
train - positive\_num: 186 - negative\_num: 36143 - data size: 36224  
train Loss: 1.0029 Acc: 0.9861  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0825 Acc: 0.8860  
remiam negative size: 93081, acc: 0.9970  
Epoch 9/9  
----------  
train - positive\_num: 186 - negative\_num: 36143 - data size: 36224  
train Loss: 1.0030 Acc: 0.9852  
val - positive\_num: 190 - negative\_num: 100230 - data size: 100352  
val Loss: 1.0824 Acc: 0.8864  
remiam negative size: 93081, acc: 0.9971  
Training complete in 107m 30s  
Best val Acc: 0.887058

#### 5.2.3 边界回归

边界回归的训练结果如下（边界框回归的训练效果相对来说不是很好，可能是数据的问题）

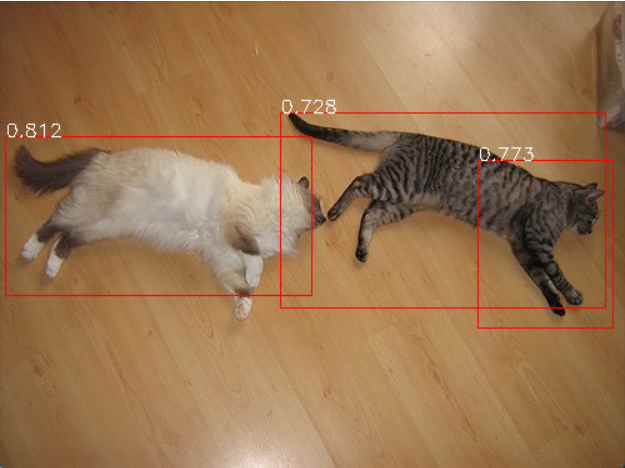
Epoch 0/11  
----------  
0 Loss: 0.4217  
Epoch 1/11  
----------  
1 Loss: 0.4111  
Epoch 2/11  
----------  
2 Loss: 0.4147  
Epoch 3/11  
----------  
3 Loss: 0.4135  
Epoch 4/11  
----------  
4 Loss: 0.4129  
Epoch 5/11  
----------  
5 Loss: 0.4131  
Epoch 6/11  
----------  
6 Loss: 0.4136  
Epoch 7/11  
----------  
7 Loss: 0.4139  
Epoch 8/11  
----------  
8 Loss: 0.4138  
Epoch 9/11  
----------  
9 Loss: 0.4118  
Epoch 10/11  
----------  
10 Loss: 0.4125  
Epoch 11/11  
----------  
11 Loss: 0.4108  
  
Training complete in 9m 53s

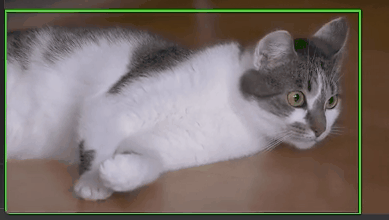
#### 5.2.4 目标检测器

目标检测器主要分为两个部分，第一个部分是对图片进行检测，第二个部分是将对图片进行检测的函数与tracker跟踪器结合实现对视频的检测

def test():  
 # test pic\_detect  
 img\_path = './images/test\_2.jpg'  
 boxes, scores = pic\_detect(img\_path)  
 print(boxes[0])  
 print(scores)  
  
def test\_2():  
 # test vedio\_detect  
 vedio\_detect('./images/cat.mp4')

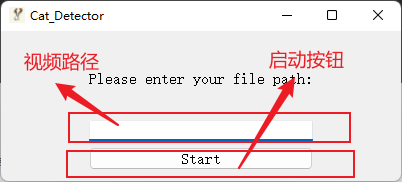
测试阶段同样分为对图片的检测和对视频的检测，测试结果会以图片以及视频帧的形式分别返回





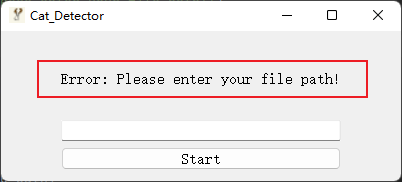
### 5.3 系统测试

Cat\_Detector目标检测系统启动后的可视化界面如下

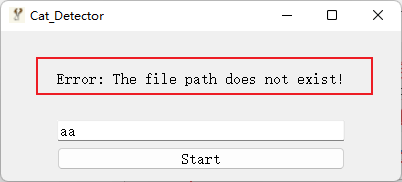


左上角分别是该系统的图标（吉祥物“鹿哥”）以及该系统的名称“Cat\_Detector”，右上角分别是“最小化”、“最大化”和“关闭程序”按钮。

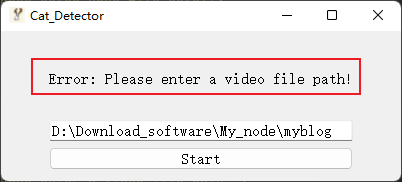
界面的中间有一个“视频路径”输入框和Start“启动按钮，系统要求在“视频路径”输入框输入合法的视频路径。假如不输入任何视频路径点击“Start”，会提醒输入文件路径



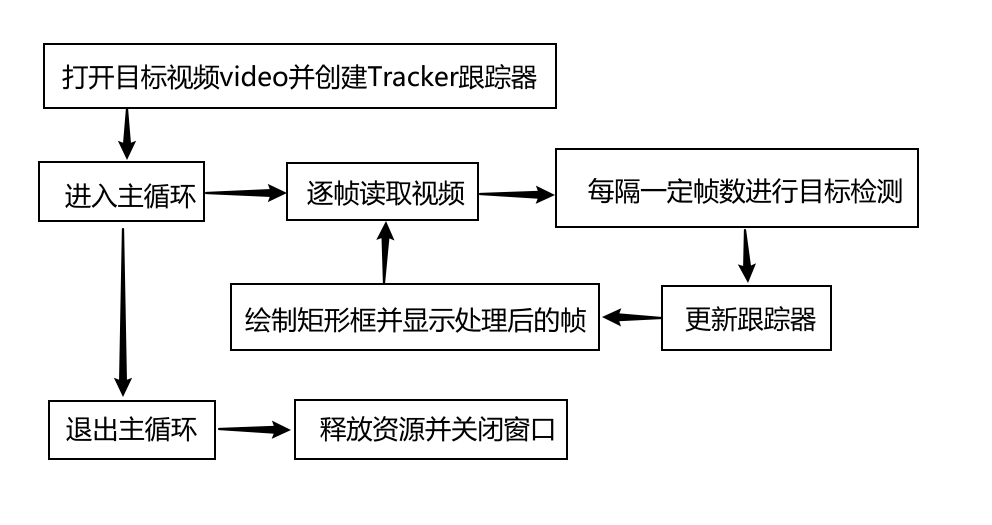
假如输入的视频路径不存在，会提示文件不存在、



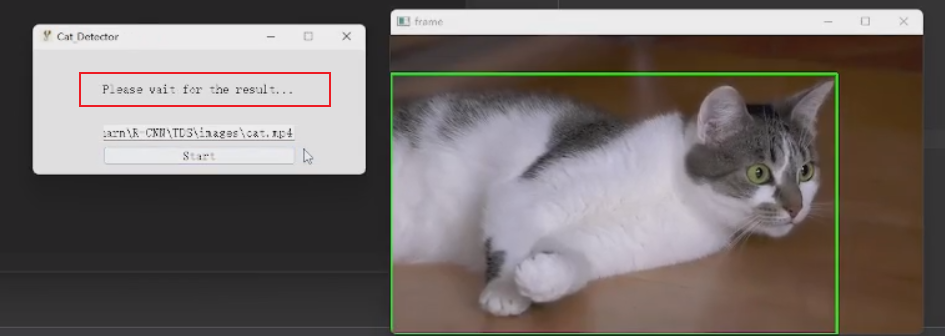
假如输入的文件路径不是视频文件，会提示错误并要求重新输入



仅当输入完整并且正确的视频路径时，点击“Start”按钮会开始进行视频检测（程序的响应速度取决于电脑性能），点击“Start”按钮之后系统的运行逻辑如下



程序响应后会弹出检测视频以及在检测视频中使用矩形框标注出猫的位置（检测视频参考根目录下的.mp4文件）



## 6.结论

从整体上看，我们的Cat\_Detector目标检测系统能够实现基本的实时目标检测，并且上手简单没有技术门槛，这是相对于当前市面上众多实时检测系统的一大优势。但是从开发者的角度来说，我认为这套系统还存在一定的不足。首先是用户界面功能不够完善，如保存历史识别信息、进行摄像头实时监测等。其次是因为底层使用的检测算法是R-CNN，这个算法的速度较慢，一般用于对图像的检测，不是特别适合用于对视频进行检测，可以考虑使用YOLO系列的检测算法或Faster-RCNN等升级算法实现更快速、更精确的视频检测。另一方面，因为网络参数设置以及训练算力的限制，本系统使用的R-CNN模型并没有发挥出全部的实力。最后，本系统可以进一步的延展，考虑不仅仅是对猫进行检测，也可以对其他更多的物体进行检测，这需要更多的训练数据以及更多的训练时间。这些都是系统还可以改进的地方。总之，本次实现的Cat\_Detector目标检测系统具备基本的功能，但还需再接再厉。

## 7.参考链接

* [(5条消息) 目标检测（Object Detection）\_图像算法AI的博客-CSDN博客](https://blog.csdn.net/yegeli/article/details/109861867)；
* [RCNN全系列详解及代码实现*哔哩哔哩*bilibili](https://www.bilibili.com/video/BV11U4y1w7yL/?spm_id_from=333.1007.top_right_bar_window_history.content.click&vd_source=276d55048634a5b508b1b53a1ecd56b3)；
* [RCNN 论文阅读记录 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/42643788)；
* [(11条消息) 目标检测\_daipuweiai的博客-CSDN博客](https://blog.csdn.net/qq_30091945/category_9816446.html)；