计算机科学与技术学院

毕业设计

论文题目	基于 SSD 网络模型的房屋瓦片损害检测		
学校导师	刘立	职称	教授
企业导师	刘立	职称	教授
学生姓名	李开运	学号	20144330106
专业班级	物联网	班级	14 级 01 班
系主任	毛宇	院长	刘振宇
起止时间	2017 年 6 月 5 日至 2018 年 5 月 22 日		

2018 年 3 月 8 日

目录

基于 SSD 网络模型的房屋瓦片损害检测

摘　要：　这也是一个摘要

关键词： 人工智能，机器视觉

基于 SSD 网络模型的房屋瓦片损害检测

摘　要：　这也是一个摘要

关键词： 人工智能，机器视觉

第一章　绪论

1.1　课题背景及研究意义

1.1.1　研究背景

自然灾害频发：美国中文网根据今日美国报道，美国国家海洋和大气管理局 (NOAA) 周一宣布，由于三次强大的飓风和凶猛的野火，2017 年是美国遭受自然灾害最为严重的一年。这些自然灾害让美国遭受了 3060 亿美元的损失。

2017 年，美国历经 16 次天气和气候灾害，每一项灾害的损失都超过了 10 亿美元。总损失约为 3060 亿美元，创下了新纪录。它打破了 2005 年的纪录，当年飓风卡特里娜等灾害给美国造成了 2150 亿美元的损失。NOAA 说，去年的灾难共造成全美 (包括波多黎各)362 人死亡。然而，NOAA 气候学家亚当 · 史密斯 (Adam Smith) 表示，死亡人数可能会随着波多黎各的后续报告而增加。史密斯说，这也是有史以来破坏最强的飓风季，损失达到 2650 亿美元。也是是有史以来损失最大的野火季，损失达到了 180 亿美元。飓风哈维总共造成了 1250 亿美元的损失，在近 30 年来造成的破坏仅次于飓风卡特里娜。飓风玛丽亚和艾玛分别造成了 900 亿美元和 500 亿美元的损失。这个消息是在德州奥斯汀的美国气象学会年会上公布的。美国大陆和阿拉斯加在 2017 年的气温也是连续第三年高于平均水平。亚利桑那州、乔治亚州、新墨西哥州、北卡罗莱纳州和南卡罗莱纳州 5 个州在 2017 年都经历了有记录以来最温暖的一年。包括阿拉斯加在内的 32 个州 2017 年的气温也创下有记录以来的前十高温。[1]

无人机的商用：从手掌大小的微型飞行器到可用于检查输电线路的商用无人机，目前市面上在售的无人机种类和数量都在迅速增加。较小的最低 40 美元就可以买到，但高端无人机的价格至少也要数千美元（军用无人机的成本更加高昂）。消费类无人机的用途主要是娱乐和拍摄，大的可以执行任务的无人机则开始用于商业投递。

说起无人机技术，大家可以想到层面会是多种多样，有军用无人机技术领域的“全球鹰”、“捕食者”，有民用无人机领域的测绘、航拍甚至快递。那个距离我们最近的“智能科技”，未来发展之路是如何呢? 可以说，无人机技术是智能科技皇冠上的一颗璀璨宝石，因为它有最小的体积，集成了最高的人类科技——1981 年，第一台商用 GPS 接收机诞生，重达 50 磅，价格高达 10 万美元。现在 GPS 仅重 0.3 克，芯片成本也大幅下降，无人机将 GPS 技术集纳；1976 年，柯达推出了第一款数字相机，像素只有 10 万，重量为 3.75 磅，价格超过 1 万美元。而无人机将最新的数字相机技术整合，并根据用途，细分了数种功能性镜头；除此之外，计算机技术、蓝牙通讯技术更是无人机的必备功能，有了这些智能抗美科技的“加持”，说无人机是一枚宝石并不为过。

机器视觉的发展：随着中国制造业的蓬勃发展，机器视觉行业也在中国市场度过了发展的最初时期，不仅国际知名品牌纷纷在中国开展业务，中国本土的企业也

[1]美国中文网,2018-1-8

逐渐兴起，机器视觉已为广大客户所熟知，应用范围也逐步扩大，由起初的电子制造业和半导体生产企业，发展到了包装，汽车，交通和印刷等多个行业。

1.1.2 研究意义

本文提出了一种对于房屋瓦片损害检测的算法方案，为检测行业提供利用机器视觉处理问题的高效手段。

1.2 研究现状及发展难点

1.2.1 研究现状

定位和分类可以迭代起来，最终在一张图片汇总对多个对象进行检测和分类。对象检测是在图像上发现和分类一个变量的问题。对象检测与定位、分类相比，重要的区别是这个“变量”。对象检测的输出长度是可变的，因为检测到的对象的数量会根据图像的不同而变化。在本文中，我们将深入了解对象检测的实际应用、作为机器学习的对象检测的主要问题是什么、以及深度学习如何在这几年里解决这个问题。

1.2.2 发展难点

可变数量的对象 (Variable number of objects) 我们之前提到了关于对象数量可变的问题，但我们却没讲它为什么是一个问题。在训练机器学习模型时，通常需要将数据表示为固定大小的向量。但是，由于图片中对象的数量事先不知道，所以我们不知道正确的输出维度。因此需要一些后期处理，这增加了模型的复杂性。一般使用滑动窗口的方法来处理可变数量的对象，通过滑动固定大小的窗口，在所有的地方生成固定大小的特征。在得到这些被过滤后的特征之后，一些被丢弃，另一些被合并以生成最终的结果。这里有个滑动窗口的例子：滑动窗口的动图。

调整对象检测窗口大小 (Resizing) 另一个巨大的挑战是各种可能的对象大小，即在进行分类时，既希望占图片大部分的对象进行分类，又想要找到一些可能只有 12 个像素、或者是原始图像一小部分的小对象。使用不同尺寸的滑动窗口可以解决这个问题，但效率很低。

建模第三个挑战是同时解决两个问题——如何用一个简单的模型解决两种不同的需求，即定位和分类。

1.3 研究内容及章节安排

第一章：绪论。本章概要阐述本文主要内容，及研究背景及意义。

第二章：目标检测相关算法。本章按照两条主线系统讲解国内外对于目标检测算法的研究。

第三章：瓦片损害检测算法设计。通过第二章的综述，我们对 SSD 算法进行了两点改进：

1 、Loss 函数 2 、Soft-NMS

第四章：瓦片损害检测算法实现。本章展示具体的算法实现

第五章：实验结果及分析。本章结合经典的目标检测算法与本算法进行对比，并展示该算法对于瓦片损害检测的效果

第六章：总结及展望。

第七章：致谢。

第二章　目标检测相关算法

2.1　目标检测算法概述

目标检测一直是计算机视觉的基础问题，在 2010 年左右就开始停滞不前了。自 2013 年一篇论文的发表，目标检测从原始的传统手工提取特征方法变成了基于卷积神经网络的特征提取，从此一发不可收拾。根着历史的潮流，简要地探讨“目标检测”算法的两种思想和这些思想引申出的算法，主要涉及那些主流算法。

在深度学习正式介入之前，传统的“目标检测”方法都是区域选择、提取特征、分类回归三部曲，这样就有两个难以解决的问题；其一是区域选择的策略效果差、时间复杂度高；其二是手工提取的特征鲁棒性较差。云计算时代来临后，“目标检测”算法大家族主要划分为两大派系，一个是 R-CNN 系两刀流，另一个则是以 YOLO 为代表的一刀流派。下面分别解释一下“两刀流”和“一刀流”。

两刀流: 顾名思义，两刀解决问题：

1、生成可能区域（Region Proposal）& CNN 提取特征

2、放入分类器分类并修正位置

这一流派的算法都离不开 Region Proposal ，即是优点也是缺点，主要代表人物就是 R-CNN 系。

一刀流: 顾名思义，一刀解决问题，直接对预测的目标物体进行回归。回归解决问题简单快速，但是太粗暴了，主要代表人物是 YOLO 和 SSD 。

无论“两刀流”还是“一刀流”，他们都是在同一个天平下选取一个平衡点、或者选取一个极端——要么准，要么快。两刀流的天平主要倾向准，一刀流的天平主要倾向快。但最后万剑归宗，大家也找到了自己的平衡，平衡点的有略微的不同。接下来我们花开两朵各表一支，一朵两刀流的前世今生，另一朵一刀流的发展历史。

2.2　Viola-Jones 人脸检测器

物体检测在整个计算机领域里，比较成功的一个例子，就是在大概 2000 年前后出现的 Viola-Jones 人脸检测器，其使得物体检测相比而言成了一项较为成熟的技术。这个方法基本的思路就是滑动窗口式的，用一个固定大小的窗口在输入图像进行滑动，窗口框定的区域会被送入到分类器，去判断是人脸窗口还是非人脸窗口。滑动的窗口其大小是固定的，但是人脸的大小则多种多样，为了检测不同大小的人脸，还需要把输入图像缩放到不同大小，使得不同大小的人脸能够在某个尺度上和窗口大小相匹配。这种滑动窗口式的做法有一个很明显的问题，就是有太多的位置要去检查，去判断是人脸还是非人脸。

判断是不是人脸，这是两个分类问题，在 2000 年的时候，采用的是 AdaBoost 分类器。进行分类时，分类器的输入用的是 Haar 特征，这是一种非常简单的特征，在图上可以看到有很多黑色和白色的小块，Haar 特征就是把黑色区域所有像素值之和减去白色区域所有像素值之和，以这个差值作为一个特征，黑色块和白色块有不同的大小和相对位置关系，这就形成了很多个不同的 Haar 特征。AdaBoost 分类器

是一种由多个弱分类器组合而成的强分类器，Viola-Jones 检测器是由多个 AdaBoost 分类器级联组成，这种级联结构的一个重要作用就是加速。

2000 年人脸检测技术开始成熟起来之后，就出现了相关的实际应用，例如数码相机中的人脸对焦的功能，照相的时候，相机会自动检测人脸，然后根据人脸的位置把焦距调整得更好。

2.3 可变形部件模型（DPM）

Viola-Jones 人脸检测器之后，在 2009 年出现了另外一个比较重要的方法：deformable part model（DPM），即可变形部件模型。就人脸检测而言，人脸可以大致看成是一种刚体，通常情况下不会有非常大的形变，比方说嘴巴变到鼻子的位置上去。但是对于其它物体，例如人体，人可以把胳膊抬起来，可以把腿翘上去，这会使得人体有非常多非常大的非刚性变换，而 DPM 通过对部件进行建模就能够更好地处理这种变换。刚开始的时候大家也试图去尝试用类似于 Haar 特征 +AdaBoost 分类器这样的做法来检测行人，但是发现效果不是很好，到 2009 年之后，有了 DPM 去建模不同的部件，比如说人有头有胳膊有膝盖，然后同时基于局部的部件和整体去做分类，这样效果就好了很多。DPM 相对比较复杂，检测速度比较慢，但是其在人脸检测还有行人和车的检测等任务上还是取得了一定的效果。后来出现了一些加速 DPM 的方法，试图提高其检测速度。DPM 引入了对部件的建模，本身是一个很好的方法，但是其被深度学习的光芒给盖过去了，深度学习在检测精度上带来了非常大的提升，所以研究 DPM 的一些人也快速转到深度学习上去了。

2.4 R-CNN 系列

R-CNN 其实是一个很大的家族，自从 rbg 大神发表那篇论文，子孙无数、桃李满天下。在此，我们只探讨 R-CNN 直系亲属，他们的发展顺序如下：

他们在整个家族进化的过程中，一致暗埋了一条主线：充分榨干 feature maps 的价值。

2.4.1 R-CNN

这个模型，是利用卷积神经网络来做「目标检测」的开山之作，其意义深远不言而喻。

解决问题一、速度传统的区域选择使用滑窗，每滑一个窗口检测一次，相邻窗口信息重叠高，检测速度慢。R-CNN 使用一个启发式方法（Selective search），先生成候选区域再检测，降低信息冗余程度，从而提高检测速度。

解决问题二、特征提取传统的手工提取特征鲁棒性差，限于如颜色、纹理等低层次（Low level）的特征。使用 CNN （卷积神经网络）提取特征，可以提取更高层面的抽象特征，从而提高特征的鲁棒性。

该方法将 PASCAL VOC 上的检测率从 35.1% 提升到 53.7 % ，提高了好几个量

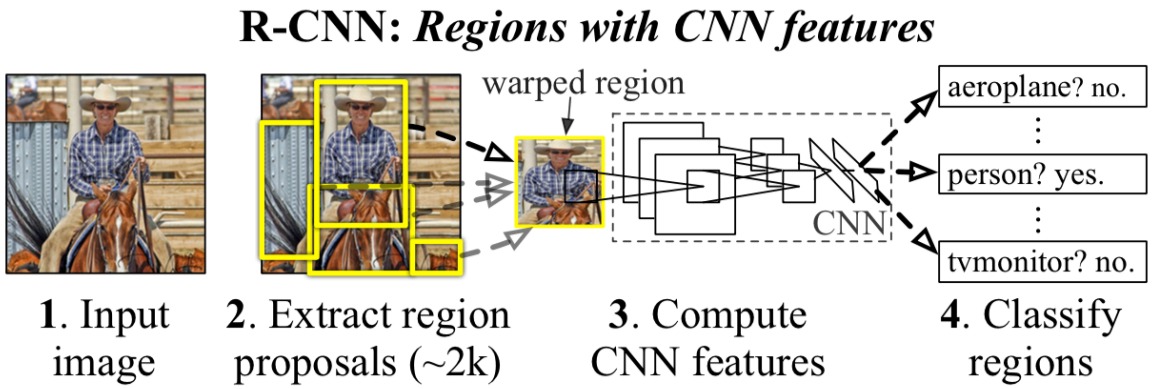

图 2.1: RCNN

级。虽然比传统方法好很多，但是从现在的眼光看，只能是初窥门径。

2.4.2 SPP Net

R-CNN 提出后的一年，以何恺明、任少卿为首的团队提出了 SPP Net ，这才是真正摸到了卷积神经网络的脉络。也不奇怪，毕竟这些人鼓捣出了 ResNet 残差网络，对神经网络的理解是其他人没法比的。尽管 R-CNN 效果不错，但是他还有两个硬伤：

硬伤一、算力冗余先生成候选区域，再对区域进行卷积，这里有两个问题：其一是候选区域会有一定程度的重叠，对相同区域进行重复卷积；其二是每个区域进行新的卷积需要新的存储空间。何恺明等人意识到这个可以优化，于是把先生成候选区域再卷积，变成了先卷积后生成区域。“简单地”改变顺序，不仅减少存储量而且加快了训练速度。

硬伤二、图片缩放无论是剪裁（Crop）还是缩放（Warp），在很大程度上会丢失图片原有的信息导致训练效果不好，如上图所示。直观的理解，把车剪裁成一个门，人看到这个门也不好判断整体是一辆车；把一座高塔缩放成一个胖胖的塔，人看到也没很大把握直接下结论。人都做不到，机器的难度就可想而知了。

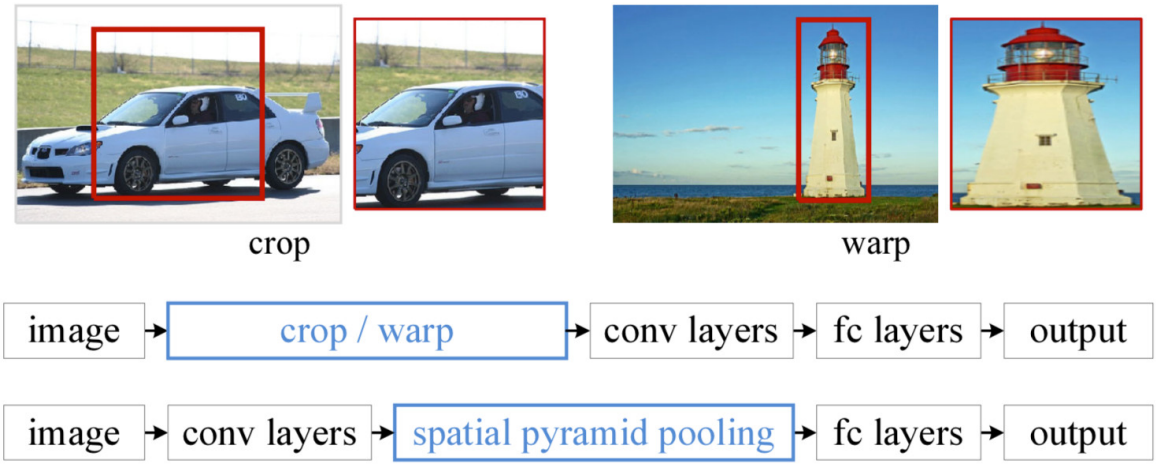

图 2.2: RCNN

何恺明等人发现了这个问题，于是思索有什么办法能不对图片进行变形，将图片原汁原味地输入进去学习。最后，他们发现问题的根源是 FC Layer（全连接层）需要确定输入维度，于是他们在输入全连接层前定义一个特殊的池化层，将输入的任意尺度 feature maps 组合成特定维度的输出，这个组合可以是不同大小的拼凑，如同拼凑七巧板般。举个例子，我们要输入的维度 64*256 ，那么我们可以这样组合 32*256+16*256+8*256+8*256 。

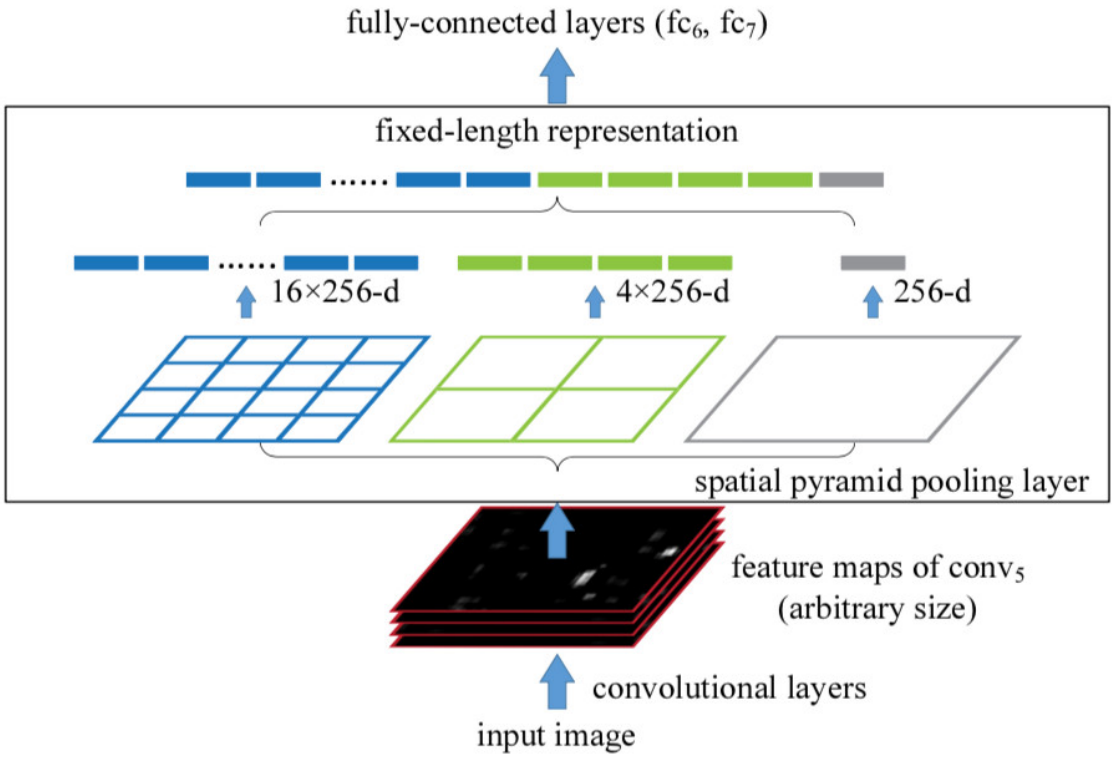

图 2.3: RCNN

SPP Net 的出现是如同一道惊雷，不仅减少了计算冗余，更重要的是打破了固定尺寸输入这一束缚，让后来者享受到这一缕阳光。

2.4.3 Fast R-CNN

在这篇论文中，引用了 SPP Net 的工作，并且致谢其第一作者何恺明的慷慨解答。纵观全文，最大的建树就是将原来的串行结构改成并行结构

原来的 R-CNN 是先对候选框区域进行分类，判断有没有物体，如果有则对 Bounding Box 进行精修回归。这是一个串联式的任务，那么势必没有并联的快，所以 rbg 就将原有结构改成并行——在分类的同时，对 Bbox 进行回归。这一改变将 Bbox 和 Clf 的 loss 结合起来变成一个 Loss 一起训练，并吸纳了 SPP Net 的优点，最终不仅加快了预测的速度，而且提高了精度。

2.4.4 Faster R-CNN

在 Faster R-CNN 前，我们生产候选区域都是用的一系列启发式算法，基于 Low Level 特征生成区域。这样就有两个问题：

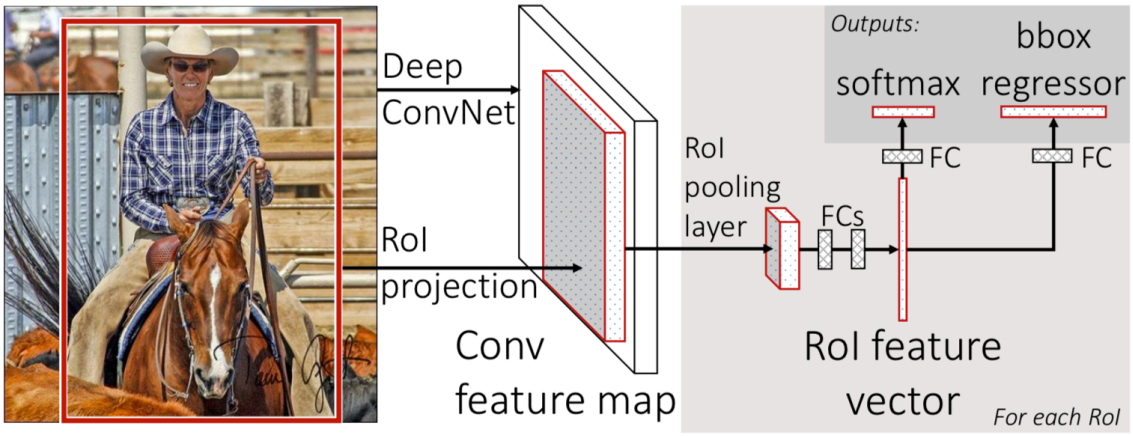

图 2.4: RCNN

第一个问题是生成区域的靠谱程度随缘，而两刀流算法正是依靠生成区域的靠谱程度——生成大量无效区域则会造成算力的浪费、少生成区域则会漏检；

第二个问题是生成候选区域的算法是在 CPU 上运行的，而我们的训练在 GPU 上面，跨结构交互必定会有损效率。

那么怎么解决这两个问题呢？于是乎，任少卿等人提出了一个 Region Proposal Networks 的概念，利用神经网络自己学习去生成候选区域。

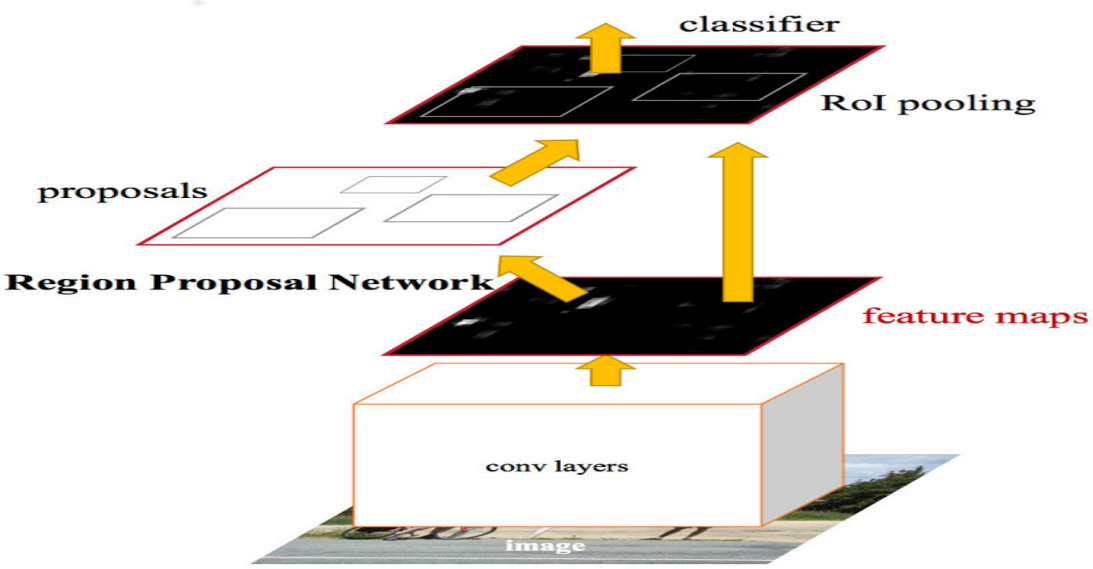

图 2.5: RCNN

这种生成方法同时解决了上述的两个问题，神经网络可以学到更加高层、语义、抽象的特征，生成的候选区域的可靠程度大大提高；可以从上图看出 RPNs 和 RoI Pooling 共用前面的卷积神经网络——将 RPNs 嵌入原有网络，原有网络和 RPNs 一起预测，大大地减少了参数量和预测时间。

在 RPNs 中引入了 anchor 的概念，feature map 中每个滑窗位置都会生成 k 个

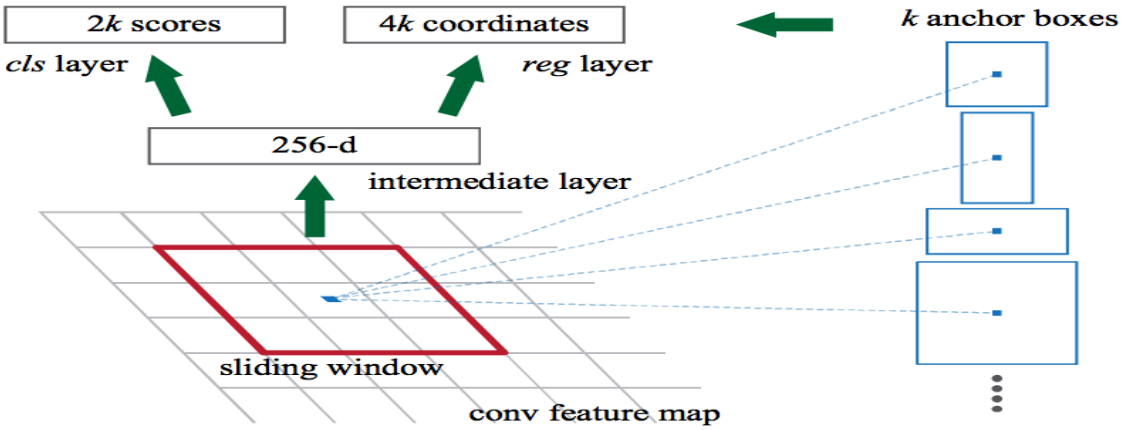

图 2.6: RCNN

anchors，然后判断 anchor 覆盖的图像是前景还是背景，同时回归 Bbox 的精细位置，预测的 Bbox 更加精确。

2.4.5 Mask R-CNN

时隔一年，何恺明团队再次更新了 R-CNN 家族，改进 Faster R-CNN 并使用新的 backbone 和 FPN 创造出了 Mask R-CNN 。

加一条通道

我们纵观发展历史，发现 SPP Net 升级为 Fast R-CNN 时结合了两个 loss ，也就是说网络输入了两种信息去训练，结果精度大大提高了。何恺明他们就思考着再加一个信息输入，即图像的 Mask ，信息变多之后会不会有提升呢？于是乎 Mask R-CNN 就这样出来了，不仅可以做「目标检测」还可以同时做「语义分割」，将两个计算机视觉基本任务融入一个框架。没有使用什么 trick ，性能却有了较为明显的提升，这个升级的版本让人们不无啧啧惊叹。作者称其为 meta algorithm ，即一个基础的算法，只要需要「目标检测」或者「语义分割」都可以使用这个作为 Backbone 。

2.5 YOLO 系列

YOLO 是单阶段方法的开山之作。它将检测任务表述成一个统一的、端到端的回归问题，并且以只处理一次图片同时到位置和分类而得名。一刀流的想法就比较暴力，给定一张图像，使用回归的方式输出这个目标的边框和类别。一刀流最核心的还是利用了分类器优秀的分类效果，首先给出一个大致的范围（最开始就是全图）进行分类，然后不断迭代这个范围直到一个精细的位置，

如上图从蓝色的框框到红色的框框。这就是一刀流回归的思想，这样做的优点就是快，但是会有许多漏检。

2.5.1 YOLO

YOLO 就是使用回归这种做法的典型算法。首先将图片 Resize 到固定尺寸，然后通过一套卷积神经网络，最后接上 FC 直接输出结果，这就他们整个网络的基本结构。更具体地做法，是将输入图片划分成一个 SxS 的网格，每个网格负责检测网

图 2.7: RCNN

格里面的物体是啥，并输出 Bbox Info 和置信度。这里的置信度指的是该网格内含有什么物体和预测这个物体的准确度。

更具体的是如下定义:

$$Pr(Class_i|Object) * Pr(Object) * IOU_{pred}^{truth} = Pr(Class_i) * IOU_{pred}^{truth}$$

这个想法其实就是一个简单的分而治之想法，将图片卷积后提取的特征图分为 SxS 块，然后利用优秀的分类模型对每一块进行分类，将每个网格处理完使用 NMS（非极大值抑制）的算法去除重叠的框，最后得到我们的结果。

2.5.2 SSD

YOLO 这样做的确非常快，但是问题就在于这个框有点大，就会变得粗糙——小物体就容易从这个大网中漏出去，因此对小物体的检测效果不好。

所以 SSD 就在 YOLO 的主意上添加了 Faster R-CNN 的 Anchor 概念，并融合不同卷积层的特征做出预测。

第三章将详细阐述 SSD 算法

2.5.3 YOLO9000

到了 SSD ，回归方法的目标检测应该一统天下了，但是 YOLO 的作者不服气，升级做了一个 YOLO9000 ——号称可以同时识别 9000 类物体的实时监测算法。讲道理，YOLO9000 更像是 SSD 加了一些 Trick ，而并没有什么本质上的进步:

加了 BN 层，扩大输入维度，使用了 Anchor，训练的时候数据增强…所以强是强，但没啥新意，SSD 和 YOLO9000 可以归为一类。

2.6 本章小结

回顾过去，从 YOLO 到 SSD ，人们兼收并蓄把不同思想融合起来。YOLO 使用了分治思想，将输入图片分为 SxS 的网格，不同网格用性能优良的分类器去分类。SSD 将 YOLO 和 Anchor 思想融合起来，并创新使用 Feature Pyramid 结构。但是 Resize 输入，必定会损失许多的信息和一定的精度，这也许是一刀流快的原因。无论如何，YOLO 和 SSD 这两篇论文都是让人不得不赞叹他们想法的精巧，让人受益良多在目标检测中有两个指标：快（Fast）和准（Accurate）。一刀流代表的是快，但是最后在快和准中找到了平衡，第一是快，第二是准。两刀流代表的是准，虽然没有那么快但是也有 6 FPS 可接受的程度，第一是准，第二是快。两类算法都有其适用的范围，比如说实时快速动作捕捉，一刀流更胜一筹；复杂、多物体重叠，两刀流当仁不让。没有不好的算法，只有不合适的使用场景。我相信 Mask R-CNN 并不是「目标检测」的最终答案，江山代有才人出，展望未来。

第三章　瓦片损害检测算法设计

3.1　瓦片损害检测流程

1、无人机拍照

2、图像切割

3、分小图送检

4、输出结果

3.2　SSD 算法核心思想

基于" Proposal + Classification" 的 Object Detection 的方法，RCNN 系列 (R-CNN、SPPnet、Fast R-CNN 以及 Faster R-CNN) 取得了非常好的效果，因为这一类方法先预先回归一次边框，然后再进行骨干网络训练，所以精度要高，这类方法被称为 two stage 的方法。但也正是由于此，这类方法在速度方面还有待改进。由此，YOLO[8] 应运而生，YOLO 系列只做了一次边框回归和打分，所以相比于 RCNN 系列被称为 one stage 的方法，这类方法的最大特点就是速度快。但是 YOLO 虽然能达到实时的效果，但是由于只做了一次边框回归并打分，这类方法导致了小目标训练非常不充分，对于小目标的检测效果非常的差。简而言之，YOLO 系列对于目标的尺度比较敏感，而且对于尺度变化较大的物体泛化能力比较差。

针对 YOLO 和 Faster R-CNN 的各自不足与优势，WeiLiu 等人提出了 Single Shot MultiBox Detector，简称为 SSD。SSD 整个网络采取了 one stage 的思想，以此提高检测速度。并且网络中融入了 Faster R-CNN 中的 anchors 思想，并且做了特征分层提取并依次计算边框回归和分类操作，由此可以适应多种尺度目标的训练和检测任务。SSD 的出现使得大家看到了实时高精度目标检测的可行性。

SSD 和 Yolo 一样都是采用一个 CNN 网络来进行检测，但是却采用了多尺度的特征图，其基本架构如图 3 所示。下面将 SSD 核心设计理念总结为以下三点:

1、采用多尺度特征图用于检测

所谓多尺度采用大小不同的特征图，CNN 网络一般前面的特征图比较大，后面会逐渐采用 stride=2 的卷积或者 pool 来降低特征图大小，这正如图 3 所示，一个比较大的特征图和一个比较小的特征图，它们都用来做检测。这样做的好处是比较大的特征图来用来检测相对较小的目标，而小的特征图负责检测大目标，如图 4 所示，8x8 的特征图可以划分更多的单元，但是其每个单元的先验框尺度比较小。

2、采用卷积进行检测

与 Yolo 最后采用全连接层不同，SSD 直接采用卷积对不同的特征图来进行提取检测结果。对于形状为 $m \times n \times p$ 的特征图，只需要采用 $3 \times 3 \times p$ 这样比较小的卷积核得到检测值。

3、设置先验框

在 Yolo 中，每个单元预测多个边界框，但是其都是相对这个单元本身 (正方块)，但是真实目标的形状是多变的，Yolo 需要在训练过程中自适应目标的形状。而 SSD

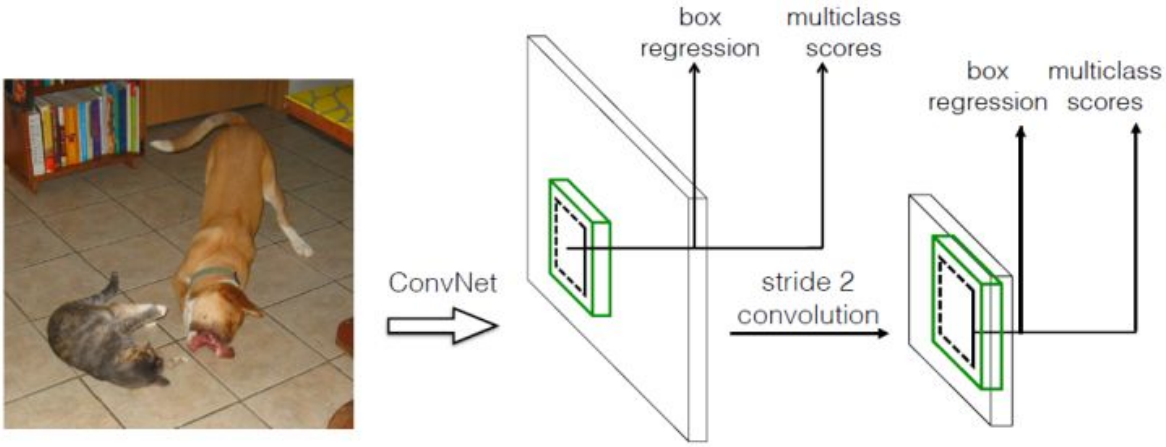

图 3.1: RCNN

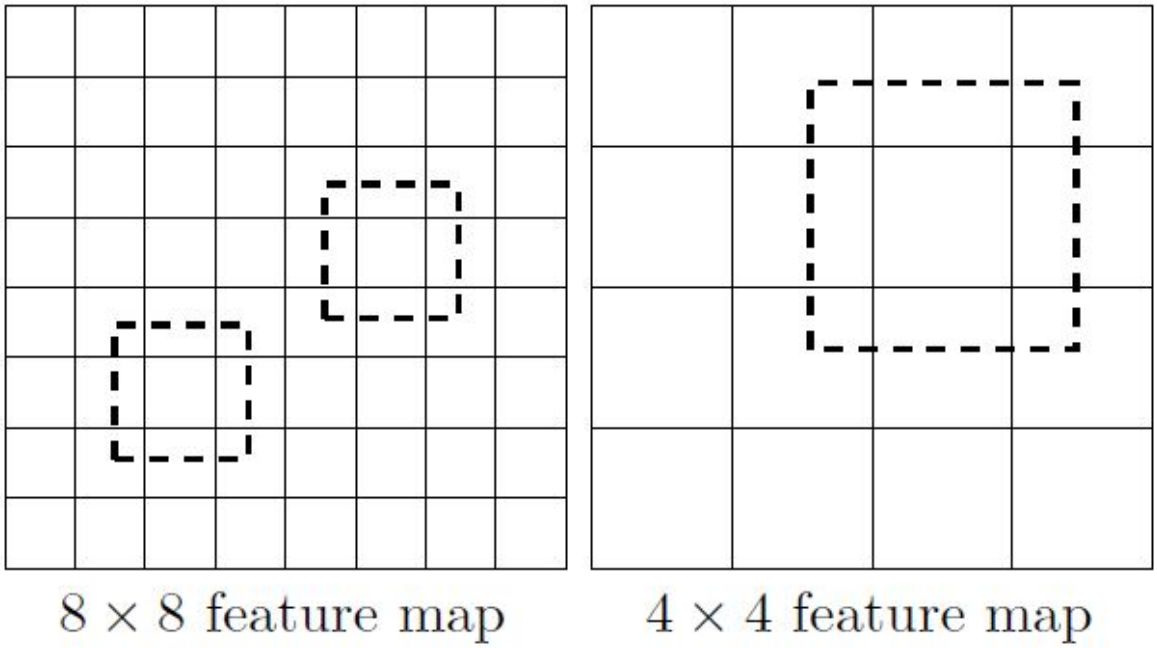

图 3.2: RCNN

借鉴了 Faster R-CNN 中 anchor 的理念，每个单元设置尺度或者长宽比不同的先验框，预测的边界框（bounding boxes）是以这些先验框为基准的，在一定程度上减少训练难度。一般情况下，每个单元会设置多个先验框，其尺度和长宽比存在差异，如图 5 所示，可以看到每个单元使用了 4 个不同的先验框，图片中猫和狗分别采用最适合它们形状的先验框来进行训练，后面会详细讲解训练过程中的先验框匹配原则。

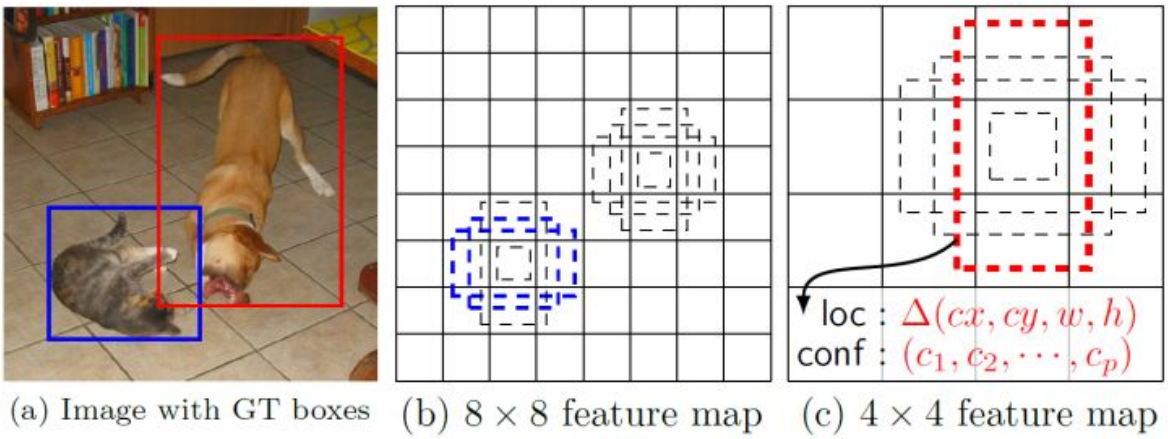

图 3.3: RCNN

3.3 SSD 模型结构

SSD 网络主体设计的思想是特征分层提取，并依次进行边框回归和分类。因为不同层次的特征图能代表不同层次的语义信息，低层次的特征图能代表低层语义信息 (含有更多的细节)，能提高语义分割质量，适合小尺度目标的学习。高层次的特征图能代表高层语义信息，能光滑分割结果，适合对大尺度的目标进行深入学习。所以作者提出的 SSD 的网络理论上能适合不同尺度的目标检测。

所以 SSD 网络中分为了 6 个 stage，每个 stage 能学习到一个特征图，然后进行边框回归和分类。SSD 网络以 VGG16 的前 5 层卷积网络作为第 1 个 stage，然后将 VGG16 中的 fc6 和 fc7 两个全连接层转化为两个卷积层 Conv6 和 Conv7 作为网络的第 2、第 3 个 stage。接着在此基础上，SSD 网络继续增加了 Conv8、Conv9、Conv10 和 Conv11 四层网络，用来提取更高层次的语义信息。如下图 3.1 所示就是 SSD 的网络结构。在每个 stage 操作中，网络包含了多个卷积层操作，每个卷积层操作基本上都是小卷积。

骨干网络：SSD 前面的骨干网络选用的 VGG16 的基础网络结构，如上图所示，虚线框内的是 VGG16 的前 5 层网络。然后后面的 Conv6 和 Conv7 是将 VGG16 的后两层全连接层网络 (fc6, fc7) 转换而来。另外：在此基础上，SSD 网络继续增加了 Conv8 和 Conv9、Conv10 和 Conv11 四层网络。图中所示，立方体的长高表示特征图的大小，厚度表示是 channel。

3.4 SSD 模型训练

训练 SSD 和基于 region proposal 方法的最大区别就是：SSD 需要精确的将 ground truth 映射到输出结果上。这样才能提高检测的准确率。文中主要采取了以下

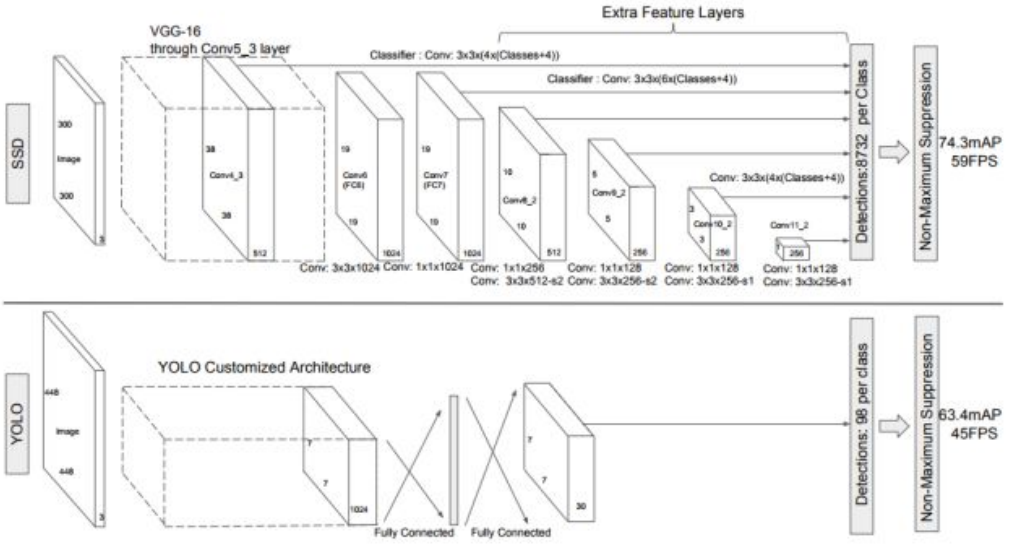

图 3.4: RCNN

几个技巧来提高检测的准确度。

匹配策略 Default boxes 生成器 Hard Negative Mining Data Augmentation

1. **匹配策略**

这里的匹配是指的 ground truth 和 Default box 的匹配。这里采取的方法与 Faster R-CNN 中的方法类似。主要是分为两步：第一步是根据最大的 overlap 将 ground truth 和 default box 进行匹配 (根据 ground truth 找到 default box 中 IOU 最大的作为正样本)，第二步是将 default boxes 与 overlap 大于某个阈值 (目标检测中通常选取 0.5，Faster R-CNN 中选取的是 0.7) 的 ground truth 进行匹配。

2. Default Boxes **生成器**

3. Hard Negative Mining 经过匹配策略会得到大量的负样本，只有少量的正样本。这样导致了正负样本不平衡，经过试验表明，正负样本的不均衡是导致检测正确率低下的一个重要原因。所以在训练过程中采用了 Hard Negative Mining 的策略，根据 Confidence Loss 对所有的 box 进行排序，使得正负样本的比例控制在 1:3 之内，经过作者实验，这样做能提高 4% 左右的准确度。

4. Data Augmentation

为了模型更加鲁棒，需要使用不同尺度目标的输入，作者对数据进行了增强处理。

1、使用整张图像作为输入

2、使用 IOU 和目标物体为 0.1、0.3、0.5、0.7 和 0.9 的 patch，这些 patch 在原图的大小的 [0.1, 1] 之间，相应的宽高比在 [1/2, 2] 之间。

3、随机采取一个 patch

3.5 算法改进方案

3.5.1 损失函数-Focus Loss

我们知道 object detection 的算法主要可以分为两大类：two-stage detector 和 one-stage detector。前者是指类似 Faster RCNN，RFCN 这样需要 region proposal 的检测算法，这类算法可以达到很高的准确率，但是速度较慢。虽然可以通过减少 proposal 的数量或降低输入图像的分辨率等方式达到提速，但是速度并没有质的提升。后者是指类似 YOLO，SSD 这样不需要 region proposal，直接回归的检测算法，这类算法速度很快，但是准确率不如前者。作者提出 focal loss 的出发点也是希望 one-stage detector 可以达到 two-stage detector 的准确率，同时不影响原有的速度。

既然有了出发点，那么就要找 one-stage detector 的准确率不如 two-stage detector 的原因，作者认为原因是：样本的类别不均衡导致的。我们知道在 object detection 领域，一张图像可能生成成千上万的 candidate locations，但是其中只有很少一部分是包含 object 的，这就带来了类别不均衡。那么类别不均衡会带来什么后果呢? 引用原文讲的两个后果：(1) training is inefficient as most locations are easy negatives that contribute no useful learning signal; (2) en masse, the easy negatives can overwhelm training and lead to degenerate models. 什么意思呢? 负样本数量太大，占总的 loss 的大部分，而且多是容易分类的，因此使得模型的优化方向并不是我们所希望的那样。其实先前也有一些算法来处理类别不均衡的问题，比如 OHEM（online hard example mining），OHEM 的主要思想可以用原文的一句话概括：In OHEM each example is scored by its loss, non-maximum suppression (nms) is then applied, and a minibatch is constructed with the highest-loss examples。OHEM 算法虽然增加了错分类样本的权重，但是 OHEM 算法忽略了容易分类的样本。

因此针对类别不均衡问题，作者提出一种新的损失函数：focal loss，这个损失函数是在标准交叉熵损失基础上修改得到的。这个函数可以通过减少易分类样本的权重，使得模型在训练时更专注于难分类的样本。为了证明 focal loss 的有效性，作者设计了一个 dense detector：RetinaNet，并且在训练时采用 focal loss 训练。实验证明 RetinaNet 不仅可以达到 one-stage detector 的速度，也能有 two-stage detector 的准确率。

3.5.2 非极大抑制-Soft-NMS

NMS 算法的大致过程可以看原文这段话：First, it sorts all detection boxes on the basis of their scores. The detection box M with the maximum score is selected and all other detection boxes with a significant overlap (using a pre-defined threshold) with M are suppressed. This process is recursively applied on the remaining boxes. 那么传统的 NMS 算法存在什么问题呢？可以看 Figure1。在 Fiugre1 中，检测算法本来应该输出两个框，但是传统的 NMS 算法可能会把 score 较低的绿框过滤掉（如果绿框和红框的 IOU 大于设定的阈值就会被过滤掉），导致只检测出一个 object（一个马），显然这样 object 的 recall 就比较低了。可以看出 NMS 算法是略显粗暴（hard），

因为 NMS 直接将和得分最大的 box 的 IOU 大于某个阈值的 box 的得分置零，那么有没有 soft 一点的方法呢？这就是本文提出 Soft NMS。那么 Soft-NMS 算法到底是什么样呢？简单讲就是：An algorithm which decays the detection scores of all other objects as a continuous function of their overlap with M. 换句话说就是用稍低一点的分数来代替原有的分数，而不是直接置零。另外由于 Soft NMS 可以很方便地引入到 object detection 算法中，不需要重新训练原有的模型，因此这是该算法的一大优点。

3.5.3 数据增强

采用数据扩增（Data Augmentation）可以提升 SSD 的性能，主要采用的技术有水平翻转（horizontal flip），随机裁剪加颜色扭曲（random crop & color distortion），随机采集块域（Randomly sample a patch）（获取小目标训练样本）

3.6 本章小结

第四章　瓦片损害检测算法实现

4.1　图像预处理

4.1.1　标注工具

labelImg 介绍

4.1.2　数据集

VOC 数据集介绍

4.2　网络模型实现代码

Listing 1 SSD 网络模型

```
import math
import itertools

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init

from torch.autograd import Variable

from multibox_layer import MultiBoxLayer


class L2Norm2d(nn.Module):
'''L2Norm layer across all channels.'''
def __init__(self, scale):
super(L2Norm2d, self).__init__()
self.scale = scale

def forward(self, x, dim=1):
'''out = scale * x / sqrt(\sum x_i^2)'''
return self.scale * x * \
        x.pow(2).sum(dim).clamp(min=1e-12).rsqrt().expand_as(x)


class SSD300(nn.Module):
input_size = 300

def __init__(self):
super(SSD300, self).__init__()

```

```
32: # model
33: self.base = self.VGG16()
34: self.norm4 = L2Norm2d(20)
35:
36: self.conv5_1 = nn.Conv2d(512, 512, kernel_size=3, padding=1, dilation=1)
37: self.conv5_2 = nn.Conv2d(512, 512, kernel_size=3, padding=1, dilation=1)
38: self.conv5_3 = nn.Conv2d(512, 512, kernel_size=3, padding=1, dilation=1)
39:
40: self.conv6 = nn.Conv2d(512, 1024, kernel_size=3, padding=6, dilation=6)
41:
42: self.conv7 = nn.Conv2d(1024, 1024, kernel_size=1)
43:
44: self.conv8_1 = nn.Conv2d(1024, 256, kernel_size=1)
45: self.conv8_2 = nn.Conv2d(256, 512, kernel_size=3, padding=1, stride=2)
46:
47: self.conv9_1 = nn.Conv2d(512, 128, kernel_size=1)
48: self.conv9_2 = nn.Conv2d(128, 256, kernel_size=3, padding=1, stride=2)
49:
50: self.conv10_1 = nn.Conv2d(256, 128, kernel_size=1)
51: self.conv10_2 = nn.Conv2d(128, 256, kernel_size=3)
52:
53: self.conv11_1 = nn.Conv2d(256, 128, kernel_size=1)
54: self.conv11_2 = nn.Conv2d(128, 256, kernel_size=3)
55:
56: # multibox layer
57: self.multibox = MultiBoxLayer()
58:
59: def forward(self, x):
60: hs = []
61: h = self.base(x)
62: hs.append(self.norm4(h))  # conv4_3
63:
64: h = F.max_pool2d(h, kernel_size=2, stride=2, ceil_mode=True)
65:
66: h = F.relu(self.conv5_1(h))
67: h = F.relu(self.conv5_2(h))
68: h = F.relu(self.conv5_3(h))
69: h = F.max_pool2d(h, kernel_size=3, padding=1, stride=1, ceil_mode=True)
70:
71: h = F.relu(self.conv6(h))
72: h = F.relu(self.conv7(h))
73: hs.append(h)  # conv7
```

```
74:
75: h = F.relu(self.conv8_1(h))
76: h = F.relu(self.conv8_2(h))
77: hs.append(h)  # conv8_2
78:
79: h = F.relu(self.conv9_1(h))
80: h = F.relu(self.conv9_2(h))
81: hs.append(h)  # conv9_2
82:
83: h = F.relu(self.conv10_1(h))
84: h = F.relu(self.conv10_2(h))
85: hs.append(h)  # conv10_2
86:
87: h = F.relu(self.conv11_1(h))
88: h = F.relu(self.conv11_2(h))
89: hs.append(h)  # conv11_2
90:
91: loc_preds, conf_preds = self.multibox(hs)
92: return loc_preds, conf_preds
93:
94: def VGG16(self):
95: '''VGG16 layers.'''
96: cfg = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512]
97: layers = []
98: in_channels = 3
99: for x in cfg:
100: if x == 'M':
101: layers += [nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)]
102: else:
103: layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
104: nn.ReLU(True)]
105: in_channels = x
106: return nn.Sequential(*layers)
```

4.3 损失函数

Listing 2 损失函数

```
1: from __future__ import print_function
2:
3: import math
4:
5: import torch
6: import torch.nn as nn
```

```
 7: import torch.nn.init as init
 8: import torch.nn.functional as F
 9:
10: from torch.autograd import Variable
11:
12:
13: class MultiBoxLoss(nn.Module):
14: num_classes = 21
15:
16: def __init__(self):
17: super(MultiBoxLoss, self).__init__()
18:
19: def cross_entropy_loss(self, x, y):
20: '''Cross entropy loss w/o averaging across all samples.
21:
22: Args:
23: x: (tensor) sized [N,D].
24: y: (tensor) sized [N,].
25:
26: Return:
27: (tensor) cross entroy loss, sized [N,].
28: '''
29: xmax = x.data.max()
30: log_sum_exp = torch.log(torch.sum(torch.exp(x-xmax), 1)) + xmax
31: return log_sum_exp - x.gather(1, y.view(-1,1))
32:
33: def test_cross_entropy_loss(self):
34: a = Variable(torch.randn(10,4))
35: b = Variable(torch.ones(10).long())
36: loss = self.cross_entropy_loss(a,b)
37: print(loss.mean())
38: print(F.cross_entropy(a,b))
39:
40: def hard_negative_mining(self, conf_loss, pos):
41: '''Return negative indices that is 3x the number as postive indices.
42:
43: Args:
44: conf_loss: (tensor) cross entroy loss between conf_preds and conf_targets, si
45: pos: (tensor) positive(matched) box indices, sized [N,8732].
46:
47: Return:
48: (tensor) negative indices, sized [N,8732].
```

```
49: '''
50: batch_size, num_boxes = pos.size()
51:
52: conf_loss[pos] = 0  # set pos boxes = 0, the rest are neg conf_loss
53: conf_loss = conf_loss.view(batch_size, -1)  # [N,8732]
54:
55: _,idx = conf_loss.sort(1, descending=True)  # sort by neg conf_loss
56: _,rank = idx.sort(1)  # [N,8732]
57:
58: num_pos = pos.long().sum(1)  # [N,1]
59: num_neg = torch.clamp(3*num_pos, max=num_boxes-1)  # [N,1]
60:
61: neg = rank < num_neg.expand_as(rank)  # [N,8732]
62: return neg
63:
64: def forward(self, loc_preds, loc_targets, conf_preds, conf_targets):
65: '''Compute loss between (loc_preds, loc_targets) and (conf_preds, conf_target
66:
67: Args:
68: loc_preds: (tensor) predicted locations, sized [batch_size, 8732, 4].
69: loc_targets: (tensor) encoded target locations, sized [batch_size, 8732, 4].
70: conf_preds: (tensor) predicted class confidences, sized [batch_size, 8732, nu
71: conf_targets: (tensor) encoded target classes, sized [batch_size, 8732].
72:
73: loss:
74: (tensor) loss = SmoothL1Loss(loc_preds, loc_targets) + CrossEntropyLoss(conf_
75: '''
76: batch_size, num_boxes, _ = loc_preds.size()
77:
78: pos = conf_targets>0  # [N,8732], pos means the box matched.
79: num_matched_boxes = pos.data.long().sum()
80: if num_matched_boxes == 0:
81: return Variable(torch.Tensor([0]))
82:
83: ##################################################################
84: # loc_loss = SmoothL1Loss(pos_loc_preds, pos_loc_targets)
85: ##################################################################
86: pos_mask = pos.unsqueeze(2).expand_as(loc_preds)    # [N,8732,4]
87: pos_loc_preds = loc_preds[pos_mask].view(-1,4)      # [#pos,4]
88: pos_loc_targets = loc_targets[pos_mask].view(-1,4)  # [#pos,4]
89: loc_loss = F.smooth_l1_loss(pos_loc_preds, pos_loc_targets, size_average=Fals
90:
```

```
91: ##########################################################################
92: # conf_loss = CrossEntropyLoss(pos_conf_preds, pos_conf_targets)
93: #             + CrossEntropyLoss(neg_conf_preds, neg_conf_targets)
94: ##########################################################################
95: conf_loss = self.cross_entropy_loss(conf_preds.view(-1,self.num_classes), \
96: conf_targets.view(-1))  # [N*8732,]
97: neg = self.hard_negative_mining(conf_loss, pos)     # [N,8732]
98:
99: pos_mask = pos.unsqueeze(2).expand_as(conf_preds)  # [N,8732,21]
100: neg_mask = neg.unsqueeze(2).expand_as(conf_preds)  # [N,8732,21]
101: mask = (pos_mask+neg_mask).gt(0)
102:
103: pos_and_neg = (pos+neg).gt(0)
104: preds = conf_preds[mask].view(-1,self.num_classes)  # [#pos+#neg,21]
105: targets = conf_targets[pos_and_neg]                 # [#pos+#neg,]
106: conf_loss = F.cross_entropy(preds, targets, size_average=False)
107:
108: loc_loss /= num_matched_boxes
109: conf_loss /= num_matched_boxes
110: print('%f␣%f' % (loc_loss.data[0], conf_loss.data[0]), end='␣')
111: return loc_loss + conf_loss
```

4.4 模型训练/测试代码

4.5 瓦片检测

第五章　实验结果与分析

第六章　总结及展望

第七章　致谢

参考文献