基于 SSD 算法的多目标识别学习研究

——《数据挖掘》课程学习实验报告

2018-1-9

目录

1 概述

目标识别一直是计算机视觉（CV）的一个重要领域，本文主要记录了我们小组成员学习研究目标识别算法的过程以及一些实验结果。文章结构为：相关背景知识的介绍，包括对卷积神经网络和目标识别的一些基本概念的引入；然后介绍了目前几种常见的基于深度学习的目标识别算法，包括 R-CNN, Fast-R-CNN,YOLO 和 SSD 算法；最后是实验部分，本文主要针对 SSD 算法做了重点的研究和实验，在预训练好的 SSD 网络模型中，我们采取对图片采取了不同的仿射变换、亮度以及失真等处理，来检测 SSD 模型的识别效果。在一系列测试后我们发现，SSD 算法在应对一些简单的仿射变换时，可以保持较好的识别率；而在应对失真以及噪声等可以改变图片原有目标特征的情况下，表现不是特别好。但是，总体来说，该算法在处理“有损”的图片多目标识别的准确率还是很高的。

2 背景介绍

目标识别，是指用计算机实现人的视觉功能，它的研究目标就是使计算机具有从一幅或多幅图像或者是视频中认知周围环境的能力（包括对客观世界三维环境的感知、识别与理解）。

目标识别早在 20 世纪 50 年代就广泛应用在军事、空间探测和地球遥感等领域。但随着日常生活对智能化和自动化的需求日益加剧，目标识别需要更加精准和细致。事实上，自然界的一切图像都是连续变化的模拟图像，在日常生活中，这些图像中的运动目标往往是我们比较关心的，如：行人、行驶的交通工具以及其他的物体。随着社会经济的不断发展，城市化步伐的不断加速，城市的工作、生活秩序显得越来越紊乱，实时的人数统计有着重要意义。如：可以通过统计等候电梯的人数来优化调度电梯，以此提高电梯的利用率，减少用户的等待时间；可以通过统计经过十字路口、丁字路口人群流动繁忙的交通场合的人数，可以合理安排交通警察或保安人员的工作时间和工作额度；同时，现在火热的无人驾驶技术，也要求我们对目标实体进行快速准确地区分。

幸运的是，近几年深度学习的发展大大促进了目标识别技术的发展。目前，用于目标识别的方法有二十余种[8]，但常用的且效果较好的有三种，分别是 Faster R-CNN、 R-FCN 和 SSD，其余方法本质都与三种方法类似，因此接下来的分析和讨论，我们重点关注这三种方法。

由于这三种方法都是基于深度学习，考虑到报告的完整性，在本节中，我们先对模型中涉及到的深度学习的方法做简单梳理，然后在对这三种模型做一个简单的介绍。

2.1 卷积神经网络（CNN）

卷积神经网络是人工神经网络的一种，已成为当前语音分析和图像识别领域的研究热点。它的权值共享网络结构使之更类似于生物神经网络，降低了网络模型的复杂度，减少了权值的数量。

所谓的权值共享网络结构可以通过下图 1 表述。对传统的神经网络，若输入图片的像素为 N*N，与之连接的有 M 个神经元，那么这一层连接将有 O(N*N*M)数量级的权值，当 N、M 较大时这个数量级将相当大。然而图像的空间联系是局部的，就像人是通过一个局部的感受野去感受外界图像一样，每一个神经元都不需要对全局图像做感受，每个神经元只感受局部的图像区域，然后在更高层，将这些感受不同局部的神经元综合

起来就可以得到全局的信息了。这样，我们就可以减少连接的数目，也就是减少神经网络需要训练的权值参数的个数了。如下图所示，原来全连接的左图可以用右图来代替，对于每一个局部的连接，我们可以用相同的权重，每一个局部的连接就是一个卷积核（滤波器），一个卷积核提取特征就是通过移动滤波器遍历整张图片获得的，如下图 2 所示。于是，通过不同的卷积核，我们可以提取多种不同的特征。

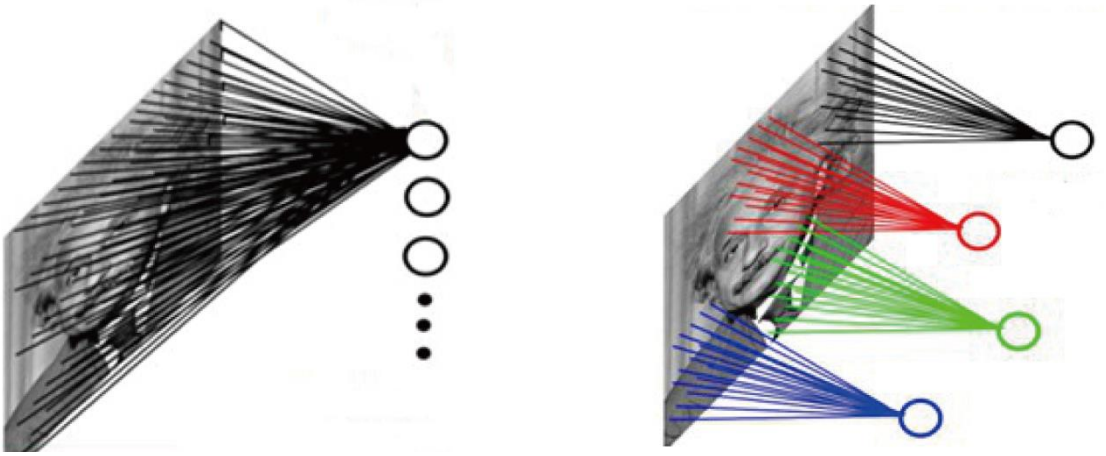


图 1：CNN 权值共享网络结构[1]

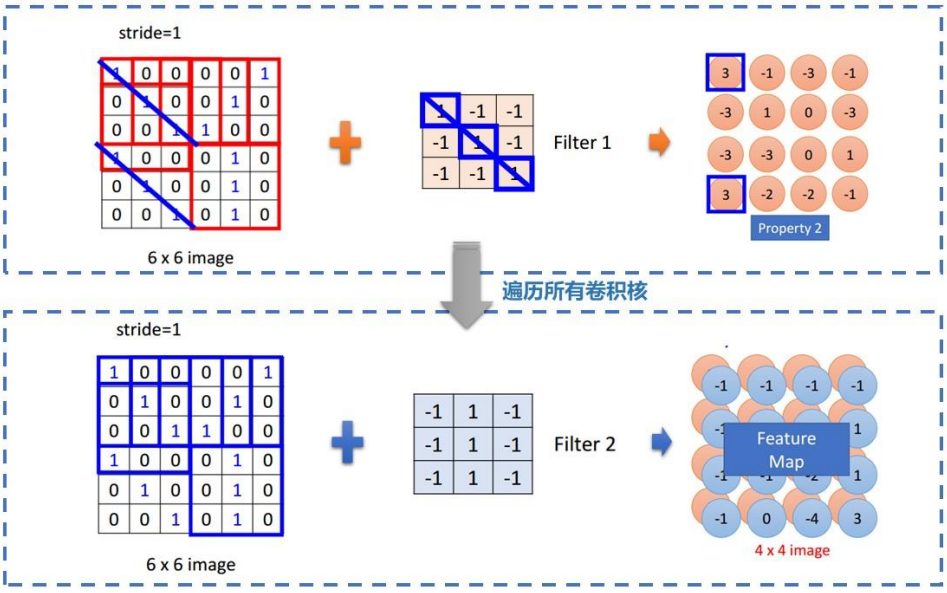


图 2 CNN 特征提取

对于提取出的特征，我们对其进行池化（pooling），所谓池化，也就是进一步提取特征中的“代表值”，如提取最大值（max pooling）是一种常用的方法。如下图 3 所示，我们利用 2*2 的矩阵遍历特征，每次取出其中的最大值。

[1] 图片来源：http://img.blog.csdn.net/20161201192409769

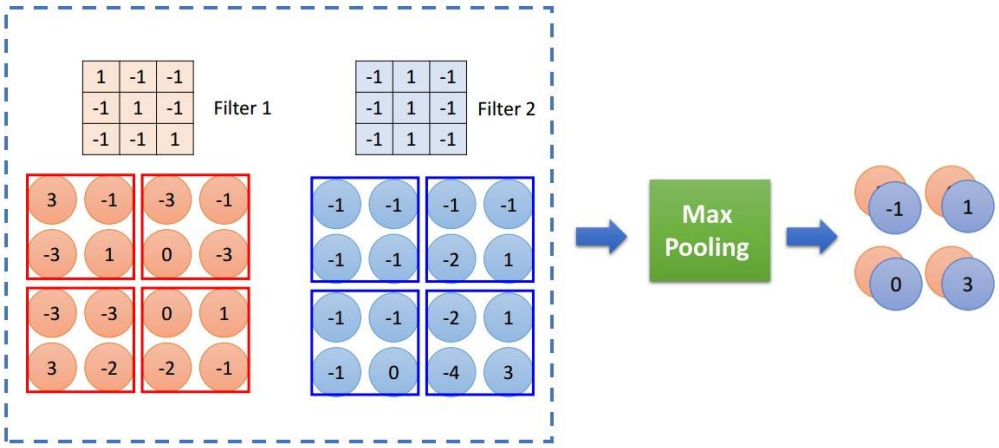


图 3 CNN 的池化(Pooling)

事实上，所有 CNN 模型都是卷积层和池化层交替叠加，图像的一小部分（局部感受区域）作为层级结构的最低层的输入，信息再依次传输到不同的层，每层通过一个数字滤波器去获得观测数据的最显著的特征，最后将特征压平后，用一个全连接层连接的过程。下图？所示给出了一个早期使用的卷积层结构，该结构曾应用于美国银行支票手写体识别，可见其“威力”非同一般。

LeNet-5 的输入层图像为 32*32 大小，卷积核大小 5*5，卷积核数量 6，每次滑动单位 1，则经过第一层卷积层 C1，输出特征图大小为 28*28(32-5+1)，神经元数量为 4707(28*28)*6；S2 层通过 2*2 的卷积核进行池化，卷积核数量 6，每个特征层得到采样图 6 个，大小为 14*14(28/2)，神经元数量为 1176(14*14)*6；类似的，C3 层也是一个卷积层，S4 层是一个池化层，C5 层为卷积层，与 F6 层全连接，F6 层有 84 个单元，有 10164 个可训练参数，如同经典神经网络。最后，输出层输出层由欧式径向基函数（EuclideanRadial Basis Function）单元组成，每类一个单元，每个有 84 个输入。换句话说，每个输出 RBF 单元计算输入向量和参数向量之间的欧式距离。输入离参数向量越远，RBF 输出的越大。一个 RBF 输出可以被理解为衡量输入模式和与 RBF 相关联类的一个模型的匹配程度的惩罚项。用概率术语来说，RBF 输出可以被理解为 F6 层配置空间的高斯分布的负 log-likelihood。给定一个输入模式，损失函数应能使得 F6 的配置与 RBF 参数向量（即模式的期望分类）足够接近。

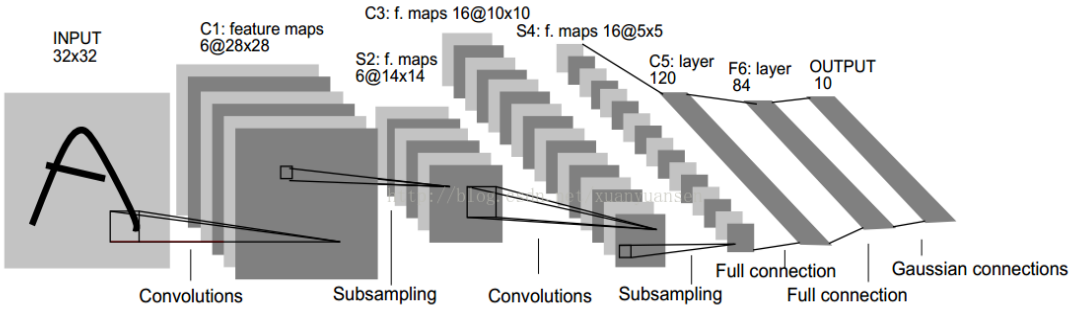


图 4 LeNet-5 结构图[2]

除了 LeNet-5，现在常用的网络结构还有 AlexNet（2012 年）、 GoogleNet（2014 年）、 VGG（2014 年）、 Deep Residual Learning（2015 年）。

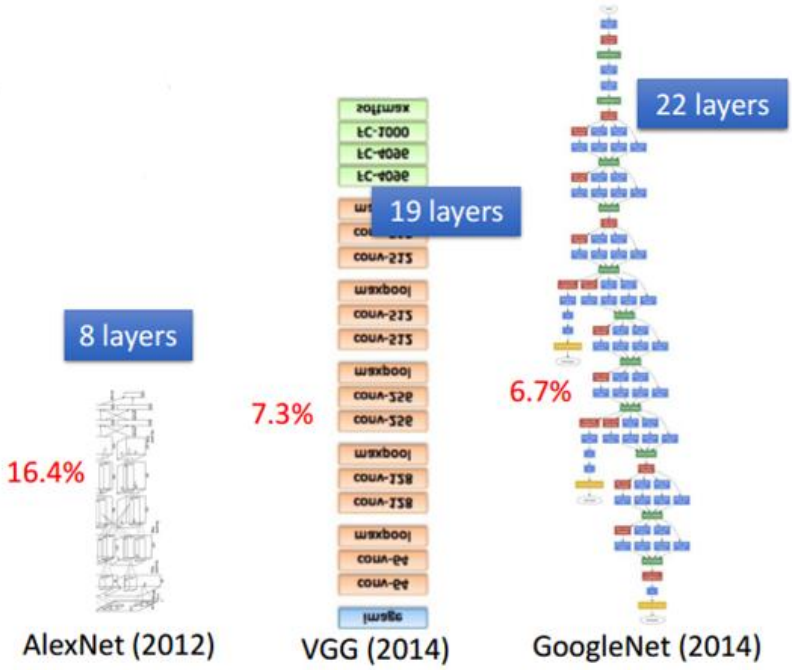


图 5 其他常用的 CNN 网络[3]

总的来说，CNN 的层间联系和空域信息的紧密关系，使其适于图像处理和理解。而且，其在自动提取图像的显著特征方面还表现出了比较优的性能。由于 CNN 的特征检测层通过训练数据进行学习，所以在使用 CNN 时，避免了显示的特征抽取，而隐式地从训练数据中进行学习；再者由于同一特征映射面上的神经元权值相同，所以网络可以并行学习，这也是卷积网络相对于神经元彼此相连网络的一大优势。

2.2 基准框(Groundtruth box)

Ground-true 实际上就是给定的标记。在有监督学习中，数据是有标注的，以(x, t)的形式出现，其中 x 是输入数据，t 是标注.正确的 t 标注是 ground truth， 错误的标记则不是。而在接下来将要提到的 SSD 目标识别中，基准框就是给定的有目标实体的范围。下图 6 给出了一个例子。

[2] 图片来源：
http://img.blog.csdn.net/20141208104822281?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQveHVhbnl1YW5zZW4=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

[3] 图片来源：http://cs231n.stanford.edu/slides/2016/

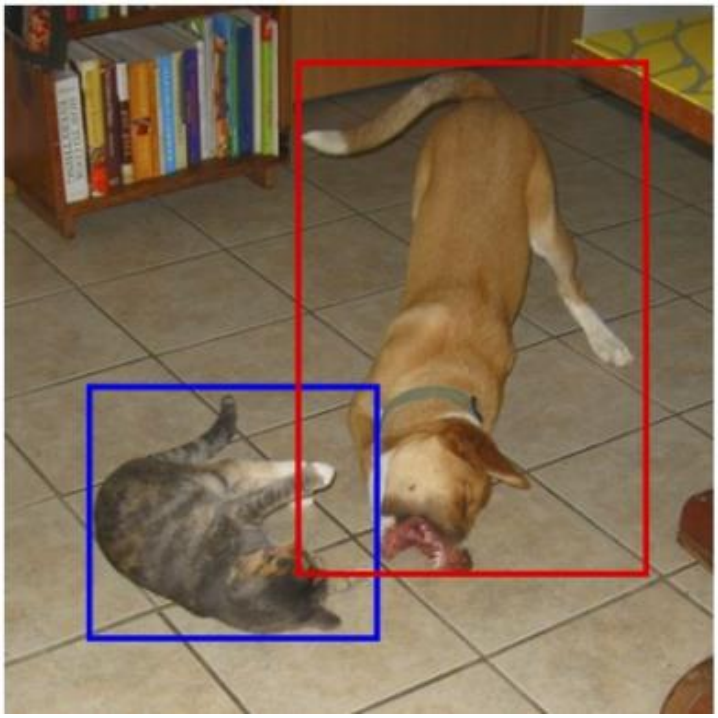


图 6 两个目标物(Cat & Dog)的基准框

基准框标记错误将高估模型效果。另外，标记数据还被用来更新权重，错误标记的数据会导致权重更新错误。

2.3 非极大值抑制（Non-Maximum Suppression, NMS）

非极大值抑制（Non-Maximum Suppression，NMS）实际上是一种局部最大搜索方法。这个局部代表的是一个邻域，邻域有两个参数可变，一是邻域的维数，二是邻域的大小。在目标检测中，NUS 就是提取分数最高的窗口。例如在行人检测中，滑动窗口提取特征，经分类器分类识别后，每个窗口都会得到一个分数。但是滑动窗口会导致很多窗口与其他窗口存在包含或者大部分交叉的情况。这时就需要用到 NMS 来选取那些邻域里分数最高（是行人的概率最大），并且抑制那些分数低的窗口。

NMS 在计算机视觉领域有着非常重要的应用，如视频目标跟踪、数据挖掘、3D 重建、目标识别以及纹理分析等。

2.4 Jaccard Index

Jaccard index 实际上是一种距离的度量。记$\mathrm{M}_1 = (x_1, x_2, \cdots, x_n)$，$\mathrm{M}_2 = (y_1, y_2, \cdots, y_n)$，那么广义的 Jaccard 相似度定义为：

$$\mathrm{J}(M_1, M_2) = \frac{\sum x_i y_i}{\sum x_i^2 + \sum y_i^2 - \sum x_i y_i}$$

若M_1和M_2是两个集合，$\mathrm{J}(M_1, M_2) = \frac{|\mathrm{M}_1 \cap \mathrm{M}_2|}{|\mathrm{M}_1 \cup \mathrm{M}_2|} = \frac{|\mathrm{M}_1 \cap \mathrm{M}_2|}{|\mathrm{M}_1| + |\mathrm{M}_2| - |\mathrm{M}_1 \cap \mathrm{M}_2|}$，用于衡量两个集合的相似度。

以上介绍了一些在做多目标检测识别上的基本知识和概念，方便后文的理解。

3 模型

3.1 目标检测方法简介

本节简单介绍一下目前几种基于深度学习的目标检测的方法[3].

3.1.1 R-CNN

R-CNN (**R**egion-based **C**onvolutional **N**eural **N**etwork) [4]，该算法框架如下：

- **Input image:** 通过选择性搜索算法(Selective Search)[4]，在输入图像中产生可能包含待检测目标物的 n 个区域框(n<2000)；
- **Feature extraction:** 用深层卷积网络(CNN)分别对 n 个区域框进行特征提取；
- **Classify& Localization**：对于每个 CNN 输出的 Feature Map：
    - **SVM**：对区域框内的待检测目标进行分类
    - **LingerRegression:** 将区域框与默认框(groundtruth box)的位置镜进行校对

具体结构见图 7。

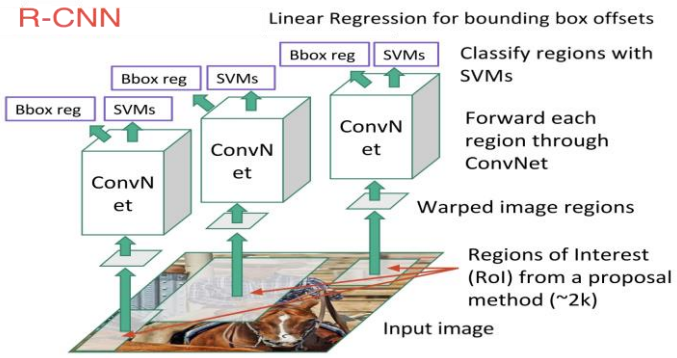


图 7　Architecture of R-CNN

R-CNN 的方法可概括为：从图片中划分不同的区域框，对不同的区域框进行特征提取，在顶层利用 SVM 进行分类，用 LinearRegression 对边框的位置进行校准。即通过分割将目标识别问题变成一个分类问题。

3.1.2 Fast R-CNN

Fast R-CNN，该算法是对 R-CNN 的一个改进，其创新点在于：

- 先对图像进行 CNN 特征提取；
- 再在顶层的 Feature Maps 上进行区域框的划分；
- 添加 Softmax 层代替 SVM 进行分类，

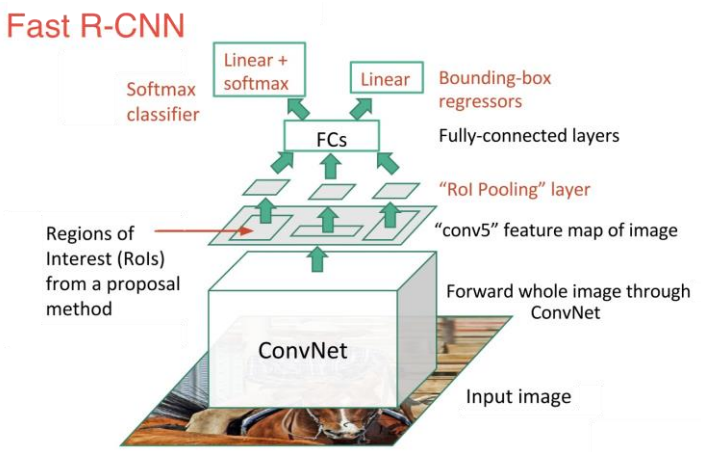


图 8 Fast R-CNN

3.1.3 Faster R-CNN

Faster R-CNN[5], Fast R-CNN 的另外一个改进版本， 其特点在于改进了 R-CNN 的区域框搜索算法，引进了一种 RPN(**R**egion **P**roposal **N**etwork)[5]算法，其他的都是基于 Fast R-CNN。 RPN 算法的结构如下：

- 在 CNN 顶层的 Feature Maps 上，构造一个 3*3 的滑窗(Slide Window)，去遍历 feature map，并通过卷积将其映射到低维的结构；
- 基于默认的边界框(Anchor boxes)，滑窗依照设定的比率 k 产生多个边界框；
- 对于每个 RPN 生成的边界框：
    - 通过分类层计算该区域包含目标物的得分
    - 通过回归层计算改边框于默认框之间的位置补偿

其核心就是基于 Anchor boxes 去产生多个边界框，再通过 Softmax 层去计算该框包含目标物的概率，设定阈值，最终输出的是有很大概率包含待检测目标物的边界框(以及框的位置信息)。然后再结合 Fast R-CNN 的网络结构，就构成了改进的 Faster R-CNN 算法。如图 9。

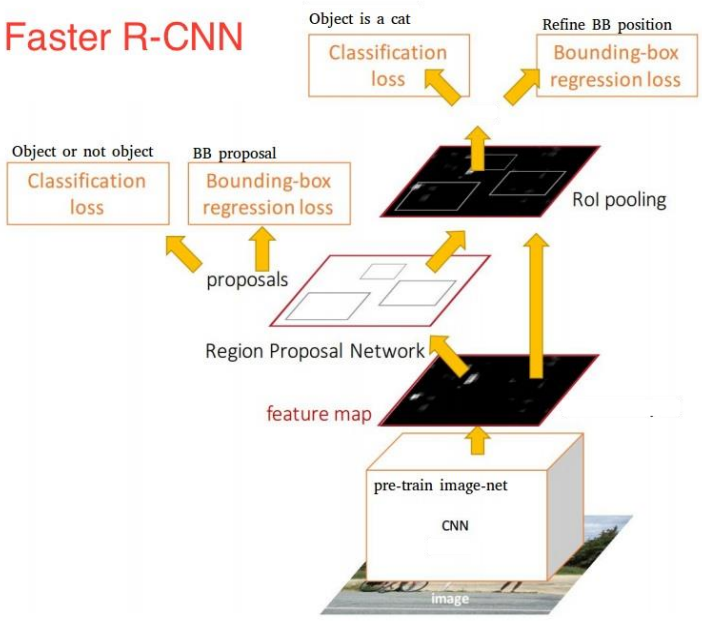


图 9 Faster R-CNN

Faster R-CNN 再改进就是本文学习研究的 SSD(**S**ingle-**S**hot **D**etector)[1]，具体在后文再详述，在此之前还有另一个'派别'的目标检测算法 YOLO(**Y**ou **O**nly **L**ook **O**nce)[6], 该算法也衍生出一系列改进的算法，在此简单介绍。

3.1.4 YOLO

YOLO：

- YOLO 首先将图片分成 s*s 的格子(grid cell);
- 如果某个目标物的中心落入格子中，该格子就负责该目标物的检测；
- 对于每个 RPN 生成的边界框：
    - 通过分类层计算该区域包含目标物的得分
    - 通过回归层计算改边框于默认框之间的位置补偿

不同于 R-CNN 系列的目标识别，YOLO 算法不需要先通过算法去预测边界框，在通过分类器去分类的两个步骤，YOLO 算法的 grid cell 的生成是于卷积的过程同步进行的，具体见图 10：

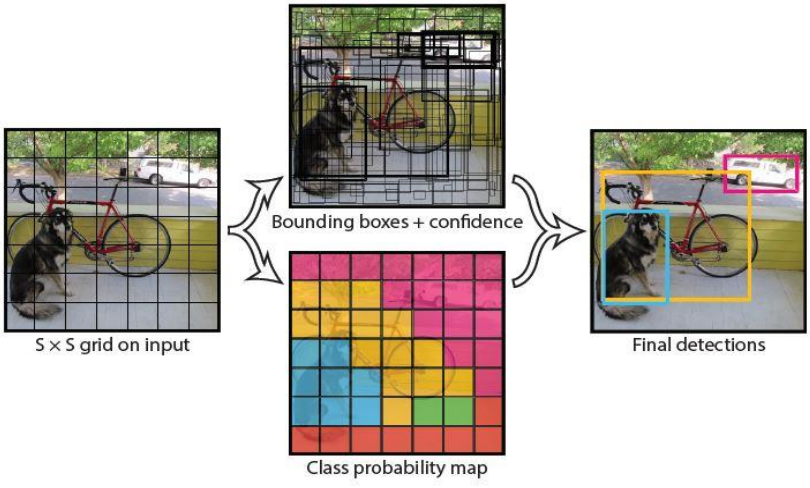


图 10 Grid Cell of YOLO

这样做的优势就是在于不需要搭建网络来做边界框的预处理，而是直接对原始图片进行处理，其优势就是在于网络结构简单，速度很快。同时速度快的代价就是牺牲了目标物检测的精度，原因在于 YOLO 算法顶层会产生重叠、较多的边界框，还需要 Intersect over Union (**IoU**) [6] 的处理重叠框和 NMS(Non-Maxima Suppression)[2] 非极大值抑制算法来选择最优框。

3.2 SSD (Single-Shot Multibox Detector)

SSD 算法是对 F-R-CNN 的一种改进算法，其整体架构还是先生成 Bounding box，再去做 Classify。具体来说，SSD 是基于一个前向传播 CNN 网络(VGG-16 [7])，添加额外的卷积层，来生成不同尺寸的特征层(feature map)，再从不同的特征层产生一系列固定大小的边界框(bounding boxes)，以及每一个 box 中包含物体实例的置信度(score)。最后，进行一个非极大值抑制(Non-maximum suppression, NMS) 得到最终的预测。

SSD 方法的核心就是预测物体以及其归属类别的 score（得分）,同时，在 feature map 上使用小的卷积核，去预测一系列 bounding boxes 的位置补偿(box offsets)。

3.2.1 Model：

其算法架构如下图 11。

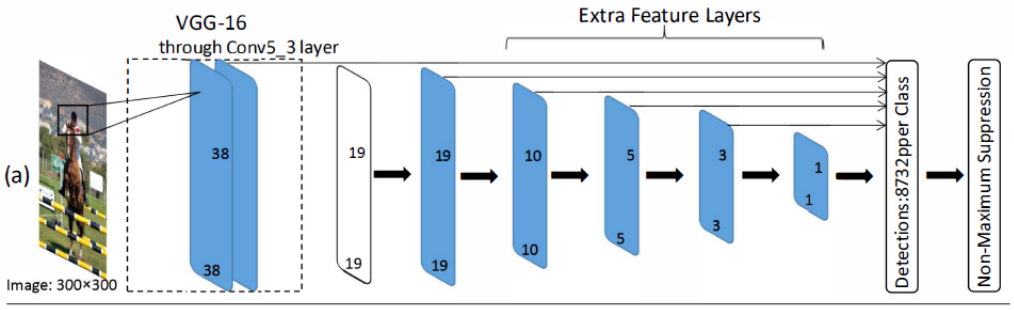


图 11 SSD (Single-Shot Multibox Detector)

具体描述为：

SSD

- 以 VGG-16[7]为底层网络对输入图片进行预处理，在此基础上添加额外的卷积核生成不同尺寸的特征层(feature maps),如(19*19,10*10,5*5 等等)；
- 对于每一个添加的特征层（或者在基础网络结构中的特征层），再使用一系列 convolutional filters，去产生一系列固定大小的预测框(bounding boxes) (不同尺寸的卷积核对应不同大小的目标物)；
- 对于不同特征层预测产生的不同尺寸的(bounding boxes)，需要计算：
    - 每个边界框包含目标物属于各个类(20 类)的概率(score)
    - 每个边界框(bounding boxes)相对于默认框(default boxes,类似于 R-CNN 的 anchor boxes)的位置偏移量
- 将所有的 bounding boxes 真实的基准框(groundtruth box)做 IoU[6]比较，IoU[6]值大于 0.5 的标记为正类(positive boxes),其他标记为负类(negative boxes);
- 基于正类和负类的样本集，通过 Jaccard [1]系数计算和基准框的相似度(图【】)，构造综合有位置和置信度的损失函数，训练判别模型

图 12 显示了 SSD 网络结构具体网络构造和维度信息：

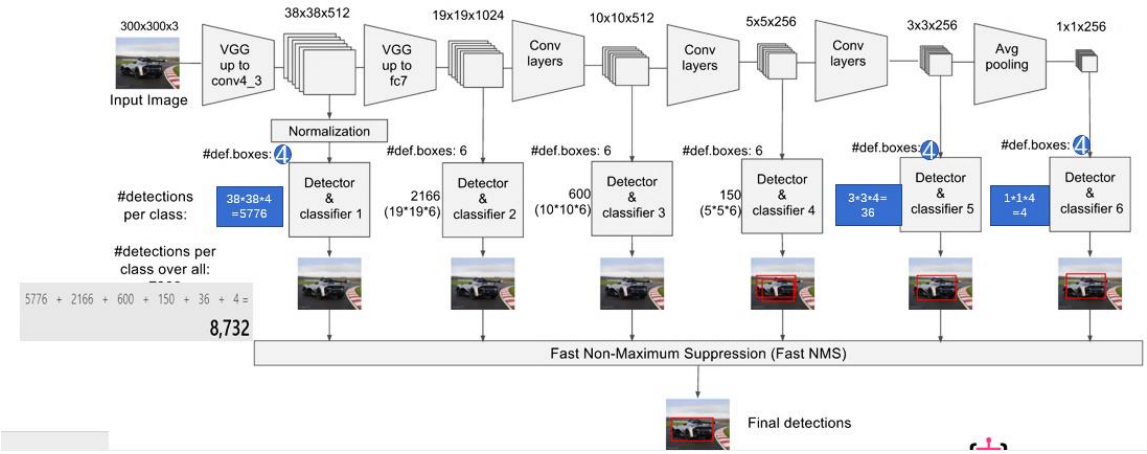


图 12 SSD 网络结构图[4]

图 12 中从左到右为 SSD 网络的卷积结构，从上到下为在 4、7、8、9、10、11 这六层卷积后生成搜索框(bounding box)，其位置是相对固定的，这几层网络的具体结构参数如表 1 所示。

[4] 图片来源：https://docs.google.com

表 1 SSD Bounding Boxes Feature map 层结构

层数	卷积操作后的特征大小(m*n*p)	增强比例 ratio	k	总的 boxes 数目
4	[38,38,512]	[2,0.5]	4	4 * 38 * 38
7	[19,19,1024]	[2,0.5,3,1/3]	6	6 * 19 * 19
8	[10,10,512]	[2,0.5,3,1/3]	6	6 * 10 * 10
9	[5,5,256]	[2,0.5,3,1/3]	6	6 * 5 * 5
10	[3,3,256]	[2,0.5]	4	4 * 3 * 3
11	[1,1,256]	[2,0.5]	4	4 * 1 * 1

上表中网格增强比例是指指的是在同一位置，原有宽度乘以一个系数、长度除以一个系数，得到新的长宽。[2,0.5]指长度变为原来 box 的二倍，宽度变为原来的 1/2。[2,0.5, 3,1/3]表示有个不同尺寸的放缩比。k 表示由原来的 groundtruth box 生成多少个 default boxes。

3.2.2 Train

下面简单介绍一下模型的训练过程和损失函数的定义：

SSD 的训练过程实际上就是对于默认框（default boxes）和基准框（groundtruth box）的匹配，基本思路是：让每一个默认框回归并且到真框，这个过程的调控需要损失层的帮助，它会计算真实值和预测值之间的误差，从而指导学习过程的收敛方向。具体过程如图 13 所示。

让每一个默认框经过 Jaccard 系数计算和真实框的相似度，阈值只有大于百分之五十的才可以列为候选名单；

假设选择出来的是 N 个匹配度高于百分之五十的框，我们令 i 表示第 i 个默认框，j 表示第 j 个真实框，p 表示第 p 个类：那么如图 13，我们就可以结合真实框和默认框(预测框)进行定位的损失计算；以及结合 c 进行置信度的计算（这里的 c 是表示置信度）。

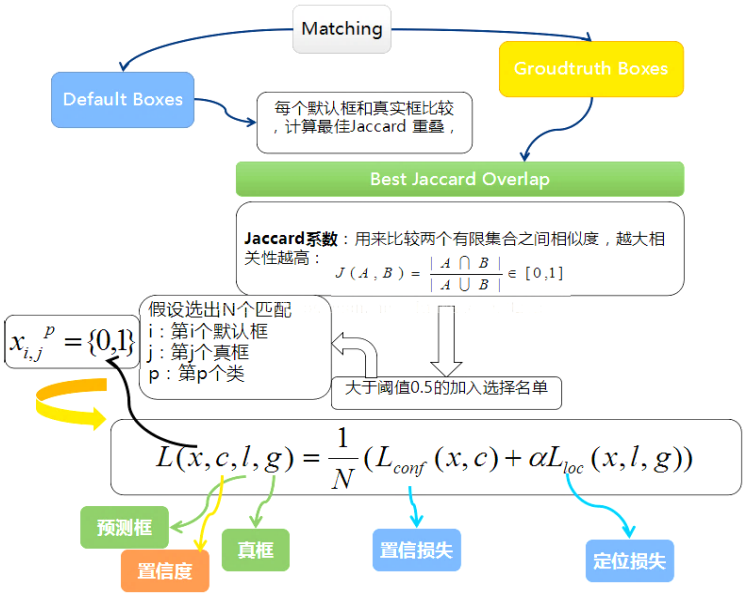


图 13 Training & Matching of SSD[5]

关于 SSD 的损失函数的定义（符号说明见上图 13）：

$$L(x,c,l,g)=\frac{1}{N}(L_{conf}(x,c)+\alpha L_{loc}(x,l,g))$$

其损失函数是从两方面定义的：一方面是 box 的定位损失，

$$L_{loc}(x,l,g)=\sum_{i\in Pos}^{N}\sum_{m\in\{cx,cy,w,h\}}x_{ij}^{k}\text{smooth}_{\text{L1}}(l_i^m-\hat{g}_j^m)$$

$$\hat{g}_j^{cx}=(g_j^{cx}-d_i^{cx})/d_i^w \qquad \hat{g}_j^{cy}=(g_j^{cy}-d_i^{cy})/d_i^h$$

$$\hat{g}_j^{w}=\log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j^{h}=\log\left(\frac{g_j^h}{d_i^h}\right)$$

另一方面是于目标物匹配的置信损失，

$$L_{conf}(x,c)=-\sum_{i\in Pos}^{N}x_{ij}^{p}log(\hat{c}_i^p)-\sum_{i\in Neg}log(\hat{c}_i^0)\quad\text{where}\quad\hat{c}_i^p=\frac{\exp(c_i^p)}{\sum_p\exp(c_i^p)}$$

总体的损失是由两个损失函数加权（权值为α）求和再平均得到的。

最后，就 SSD 算法代码的主函数及参数加以说明（见附录部分）。

4 实验及结果

我们利用一些图片对本文模型进行测试。通过对原始图片做以下六种变换，再使用本文模型对图片进行目标识别，探究变换对本文模型识别目标效果的影响。

4.1 压缩

为了探究图像压缩对目标识别的影响，我们选取了如下图 12 所示的图片(原图的尺寸为 3819*2831)，按照压缩比为 0.03、0.05、0.1、0.2 对其进行图像压缩处理。将压缩后的图像进行目标识别，其效果如下图 13 所示，其具体的目标识别率如表 2 所示。

[5] 图片来源：https://read01.com/mELQ3Pj.html

图 12 压缩测试图

表 2：不同压缩率目标被正确识别的平均概率

压缩比	0.03	0.05	0.1	0.2
person	0	0.756	0.847	0.853
car	0.723	0.849	0.779	0.784

由表 2 和图 13 可以看出，图像被压缩得越厉害，模型对于人(person)的识别越差，而对于车的识别，正确识别为车的概率大致稳定在 0.7~0.8 之间，压缩比为 0.03 和压缩比为 0.2 的图像对车的正确识别概率没有本质的差别。

(a)压缩率为 0.03

(b)压缩率为 0.05

(c)压缩率为 0.1

(d)压缩率为 0.2

图 13 SSD 对不同压缩率图片的识别效果

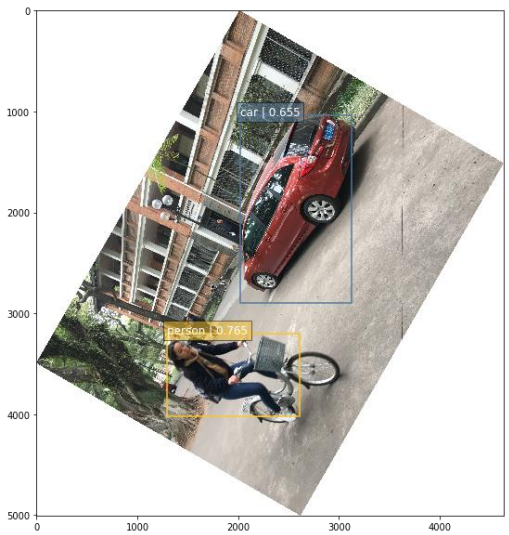
4.2 旋转

为了探究图像旋转对目标识别的影响，我们选取了在校园内拍摄的一张图片。对图片进行 60°、90°、120°、180°、-60°。对旋转后的图像进行目标识别，其效果如下图 3 所示，其具体的目标识别率如表 3 所示。

表 3 不同旋转角度图片的目标识别效果

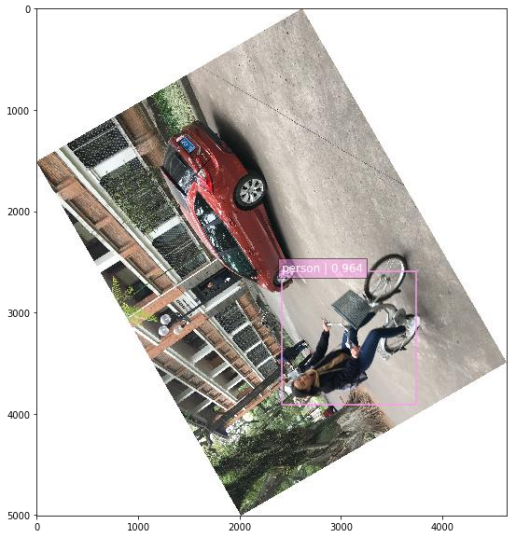
旋转角度	60°	90°	120°	180°	-60°	0°
person	0.765	0.663	0.964	0.909	0.985	0.942
car	0.655	0	0	0	0.912	1.000
bicycle	0	0	0	0	0	0.982

由表 3 和图 14 可以看出，对于图片中的自行车(BICYCLE)，旋转角度对其识别的影响最大，对于旋转后的图像，模型无法识别出图中的自行车；随着图片与水平方向的角度的变大，对图片中人(PERSON)的识别率有所下降，但无论旋转任何角度，都能够将人正确识别出来；当图片的旋转角度的绝对值大于 60° 时，模型无法识别出小汽车(CAR)，当旋转角小于 60° 时，模型能够正确识别出小汽车。

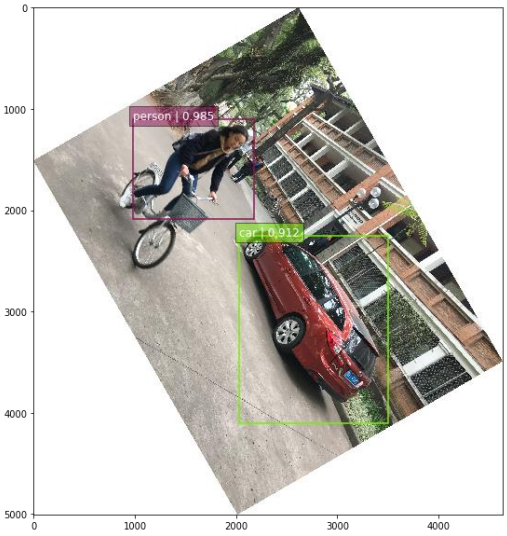


(a)旋转 60°

(b)旋转 90°



(c)旋转 120°



(d)旋转-60°

(e)旋转 180°

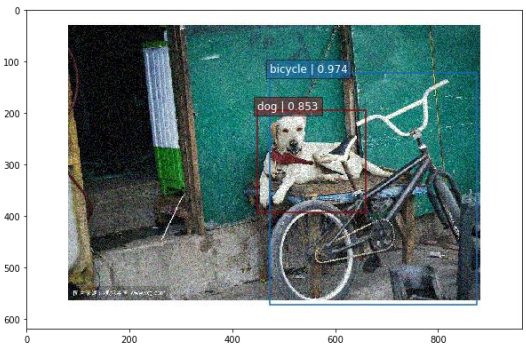
(f)原图（旋转 0°）

图 14 不同旋转角度图像的目标识别效果

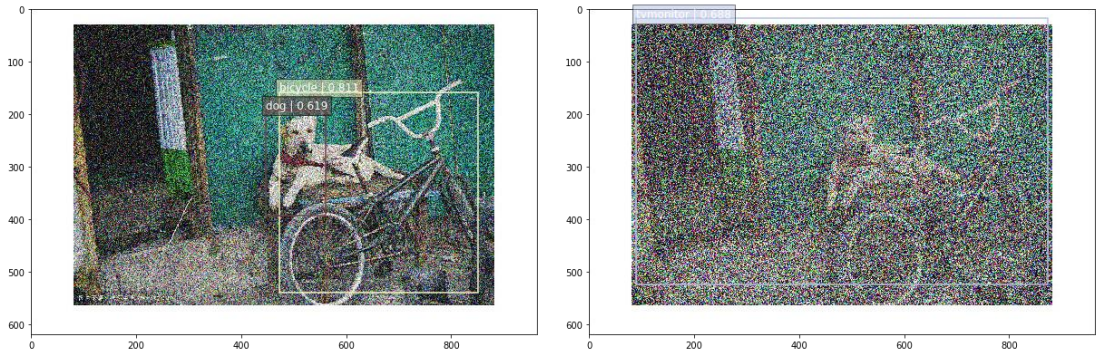
4.3 噪声

为了探究噪声对目标识别的影响，我们选取了网络中的一张图片。在现实生活中，常见的噪声是高斯噪声，因此，我们将图片分别加上不同的高斯噪声，再使用本文模型对加噪后的图片进行目标识别，其识别效果如图 15 所示，具体的目标识别率见表 4 .

(a)方差为 0（原图）



(b)方差为 0.01



(c)方差为 0.1　　(d)方差为 1

图 15 不同的高斯噪声(mean=0)图像的目标识别效果

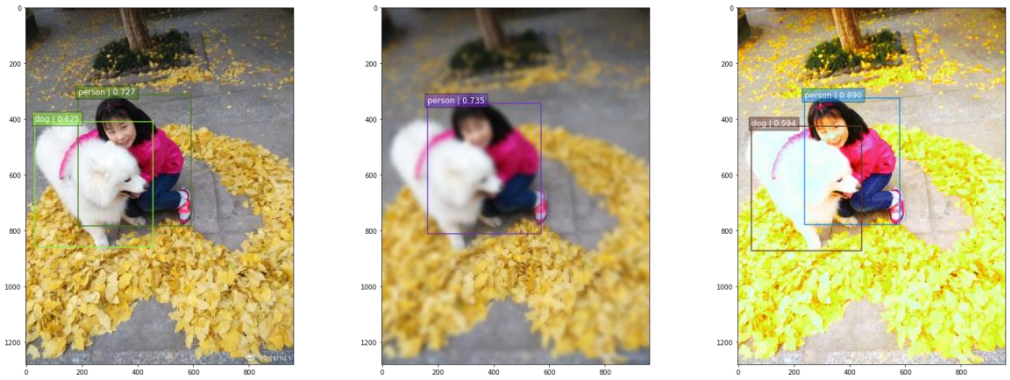
由表 4 和图 15 可以看出，高斯噪声图像，随着高斯噪声方差的增大，对图片中目标(DOG 和 BICYCLE)的正确识别率逐步下降，值得一提的是，由于方差为 1 的高斯噪声对图像的影响类似于电视中雪花效应，因而本文的模型将均值为 0、方差为 1 的高斯噪声图像识别为一个电视机，无法识别图像中的 DOG 和 BICYCLE。

表 4 不同的高斯噪声(mean=0)图像的目标识别率

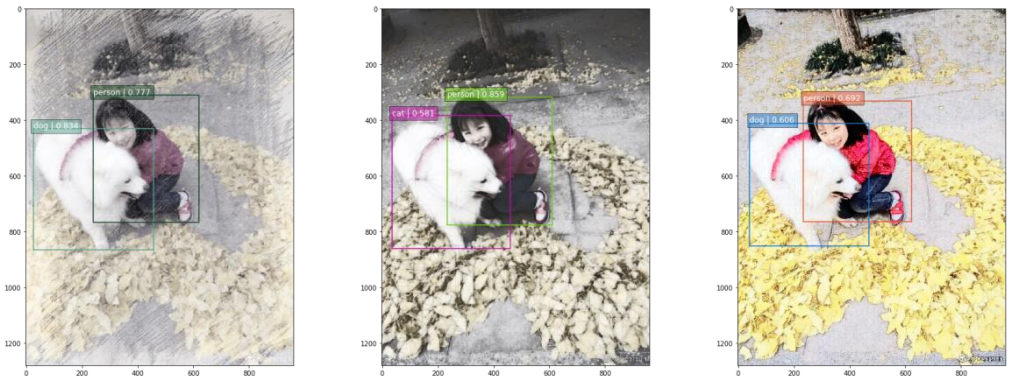
噪声的方差	0(无噪声)	0.01	0.1	1
DOG	0.954	0.853	0.619	0
BICYCLE	0.951	0.974	0.811	0

4.4 失真

为了探究图片失真对目标识别的影响，我们选取了网络中的一张图片。将图片进行失真处理，我们对图片进行虚化、色调调整等处理。其识别效果如图 16 所示，具体的目标识别率如表 5 所示。



(a)原图　　(b)虚化　　(c)色调 1



(d)色调 2　　(e)色调 3　　(f)色调 4

图 16：失真图像的目标识别效果

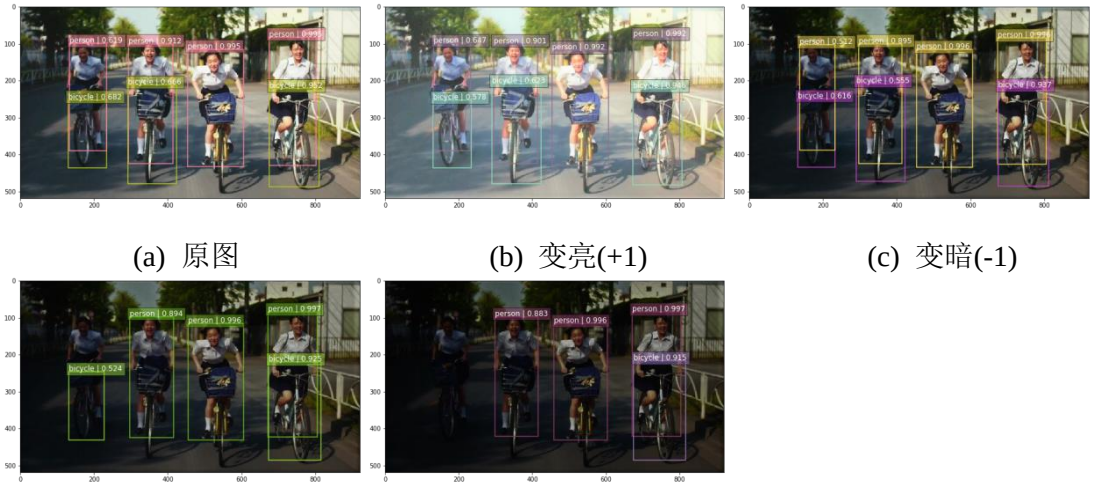
由表 5 和图 16 可以看出，虚化的处理使得模型失去了对 DOG 的识别，但是对人像的识别还是效果较佳的。在色调调整之后，对 DOG 与 PERSON 两个目标的识别没有太大影响。

表 5：失真图像的目标正确识别率

失真处理	无	虚化	色调 1	色调 2	色调 3	色调 4
PERSON	0.727	0.7235	0.89	0.777	0.859	0.692
DOG	0.625	0	0.594	0.834	0.581	0.606

4.5 亮度

为了探究光照对目标识别的影响，我们选取了网络中的一张图片。将图片的亮度进行调整，我们对图片进行调亮（+1），调暗（-1，-2）处理。其识别效果如图 17 所示，具体的目标识别率如表 6 所示。



(a) 原图　　(b) 变亮(+1)　　(c) 变暗(-1)

(d) 变暗(-2)　　(e) 变暗(-3)

图 17 不同亮度的图像的目标识别效果

由表 6 和图 17 可以看出，不同亮度的图像，随着图片亮度的增大，对图片中目标(PERSON 与 BICYCLE)的正确识别率改变不明显。随着图片亮度的降低，对于图片中目标(PERSON 与 BICYCLE)的正确识别率均有降低。值得注意的是，由于这张图片原图中这个部分的亮度受到了阳光的影响，所以存在偏差。因为在 PERSON1，BICYCLE1，

PERSON2，BICYCLE2（图片左边部分）在亮度很低的时候，几乎已经与图片背景融为一体，失去了物体的特征，模型已经失去了判别出它们的能力。PERSON3，PERSON4（图片右边部分）正确识别率改变不明显。

表 6：不同亮度的图像的目标正确识别率

Objects	原图	调亮(+1)	调暗(-1)	调暗(-2)	调暗(-3)
PERSON1	0.619	0.647	0.512	0.0	0
BICYCLE1	0.682	0.578	0.616	0.524	0
PERSON2	0.912	0.901	0.895	0.894	0.883
BICYCLE2	0.666	0.623	0.555	0	0
PERSON3	0.995	0.992	0.996	0.996	0.996
BICYCLE3	0	0	0	0	0
PERSON4	0.995	0.992	0.996	0.997	0.997
BICYCLE4	0.952	0.946	0.937	0.925	0.915

4.6 密集

为了探究图片中目标实体的密集程度，我们选取了平时常见的公路街景图片，对于同一幅场景，我们手动的去设置了不同的目标物(Car)的数据量，然后去去对比识别率。

图 18 不同数量目标物的识别效果[6]

表 7 不同目标物密度的检测识别效果

[6] 图片来源：http://b0.att.hudong.com/52/29/30000035084112889229770271 6_1024.jpg

	图 18-A	图 18-B	图 18-C	图 18-D
识别正确率	4/4	5/6	6/10	6/23
平均概率	0.8765	0.885	0.829	0.793

从图 18 和表 7 中可以看到，随着待识别目标的增多，识别的正确率在不断下降。从两方面分析这个问题：一是随着目标的越来越多，SSD 算法所需要设置的 default boxes 重叠的越来越多，目标物之间的特征会相互干扰，导致了识别率的下降；另一方面，本例的情况来说，后增的目标物的像素密度越来越小，对于算法提取其特征也有所影响，导致越小越模糊的目标，相对不容易被识别，这一点在前文压缩部分就初见端倪。

5 想法与讨论

本文主要记录了我们小组成员学习目前基于深度学习的多目标识别算法的过程。文中，大部分是我们对于目前一些主流算法的理解，并着重的对 SSD 算法做了研究，实现了 SSD(2016)算法在多种“有损”图片中的目标检测。测试过程中的图片均来自组员采集的学校及周边的场景，在原有图片的基础上，我们做了压缩、旋转、加噪、失真、亮度和目标物密度等多种情况的处理，去对比测试了模型的检测识别效果，我们发现，所有被处理过得图片，若能较大程度的保留其原始特征信息（如缩放、旋转、亮度等），测试得到了准确率依然较高；而在变换的过程中丢失了其原有特征信息的（如加噪、失真等），测试识别的准确率就较低。

本文的有待改进的地方是，要做一些横向比较。对各种不同的模型（YOLO，R-CNN 等）之间做一些测试，这样也更有利于理解各个算法各自的特点和核心。

6 参考文献

[1] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016, October). Ssd: Single shot multibox detector. In *European conference on computer vision* (pp. 21-37). Springer, Cham.

[2] Neubeck, A., & Van Gool, L. (2006, August). Efficient non-maximum suppression. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on* (Vol. 3, pp. 850-855). IEEE.

[3] https://research.googleblog.com/2017/06/supercharge-your-computer-vision-models.html

[4] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... & Darrell, T. (2014, November). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (pp. 675-678). ACM.

[5] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems* (pp. 91-99).

[6] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 779-788).

[7] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

附录

项目地址：https://github.com/weiliu89/caffe/tree/ssd

部分关键代码及注解：

```
----------------------------------------------------------------------------------------------------
"""
    Implementation of the SSD VGG-based 300 network.
    The default features layers with 300x300 image input are:
      conv4 ==> 38 x 38
      conv7 ==> 19 x 19
      conv8 ==> 10 x 10
      conv9 ==> 5 x 5
      conv10 ==> 3 x 3
      conv11 ==> 1 x 1
    The default image size used to train this network is 300x300.
"""
    default_params = SSDParams(
        img_shape=(300, 300),#输入尺寸
        num_classes=21,#预测类别 20+1=21（20 类加背景）
        #获取 feature map 层
        feat_layers=['block4', 'block7', 'block8', 'block9', 'block10', 'block11'],
        feat_shapes=[(38, 38), (19, 19), (10, 10), (5, 5), (3, 3), (1, 1)],

        anchor_size_bounds=[0.15, 0.90],
        #anchor boxes 的大小
        anchor_sizes=[(21., 45.),
                      (45., 99.),
                      (99., 153.),
                      (153., 207.),
                      (207., 261.),
                      (261., 315.)],
        #anchor boxes 的 aspect ratios
        anchor_ratios=[[2, .5],
                       [2, .5, 3, 1./3],
                       [2, .5, 3, 1./3],
                       [2, .5, 3, 1./3],
                       [2, .5],
                       [2, .5]],
```

```
        anchor_steps=[8, 16, 32, 64, 100, 300],#anchor 的层
        anchor_offset=0.5,#补偿阀值 0.5
        normalizations=[20, -1, -1, -1, -1, -1],#该特征层是否正则，大于零即正则；小于零则否
        prior_scaling=[0.1, 0.1, 0.2, 0.2]
        )
```

```
#建立 ssd 网络函数
def ssd_net(inputs,
                num_classes=21,
                feat_layers=SSDNet.default_params.feat_layers,
                anchor_sizes=SSDNet.default_params.anchor_sizes,
                anchor_ratios=SSDNet.default_params.anchor_ratios,
                normalizations=SSDNet.default_params.normalizations,
                is_training=True,
                dropout_keep_prob=0.5,
                prediction_fn=slim.softmax,
                reuse=None,
                scope='ssd_300_vgg'):
    """SSD net definition.
    """
    # End_points collect relevant activations for external use.
    #用于收集每一层输出结果
    end_points = {}
    #采用 slim 建立 vgg 网络,网络结构参考文章内的结构图
    with tf.variable_scope(scope, 'ssd_300_vgg', [inputs], reuse=reuse):
        # Original VGG-16 blocks.
        net = slim.repeat(inputs, 2, slim.conv2d, 64, [3, 3], scope='conv1')
        end_points['block1'] = net
        net = slim.max_pool2d(net, [2, 2], scope='pool1')
        # Block 2.
        net = slim.repeat(net, 2, slim.conv2d, 128, [3, 3], scope='conv2')
        end_points['block2'] = net
        net = slim.max_pool2d(net, [2, 2], scope='pool2')
        # Block 3.
        net = slim.repeat(net, 3, slim.conv2d, 256, [3, 3], scope='conv3')
```

```
        end_points['block3'] = net
        net = slim.max_pool2d(net, [2, 2], scope='pool3')
        # Block 4.
        net = slim.repeat(net, 3, slim.conv2d, 512, [3, 3], scope='conv4')
        end_points['block4'] = net
        net = slim.max_pool2d(net, [2, 2], scope='pool4')
        # Block 5.
        net = slim.repeat(net, 3, slim.conv2d, 512, [3, 3], scope='conv5')
        end_points['block5'] = net
        net = slim.max_pool2d(net, [3, 3], 1, scope='pool5')#max pool
-------------------------------------------------------------------------------------------
        #外加的 SSD 层
        # Additional SSD blocks.
        # Block 6: let's dilate the hell out of it!
        #输出 shape 为 19×19×1024
        net = slim.conv2d(net, 1024, [3, 3], rate=6, scope='conv6')
        end_points['block6'] = net
        # Block 7: 1x1 conv. Because the fuck.
        #卷积核为 1×1
        net = slim.conv2d(net, 1024, [1, 1], scope='conv7')
        end_points['block7'] = net
        # Block 8/9/10/11: 1x1 and 3x3 convolutions stride 2 (except lasts).
        end_point = 'block8'
        with tf.variable_scope(end_point):
            net = slim.conv2d(net, 256, [1, 1], scope='conv1x1')
            net = slim.conv2d(net, 512, [3, 3], stride=2, scope='conv3x3')
        end_points[end_point] = net
        end_point = 'block9'
        with tf.variable_scope(end_point):
            net = slim.conv2d(net, 128, [1, 1], scope='conv1x1')
            net = slim.conv2d(net, 256, [3, 3], stride=2, scope='conv3x3')
        end_points[end_point] = net
        end_point = 'block10'
        with tf.variable_scope(end_point):
            net = slim.conv2d(net, 128, [1, 1], scope='conv1x1')
            net = slim.conv2d(net, 256, [3, 3], scope='conv3x3', padding='VALID')
        end_points[end_point] = net
        end_point = 'block11'
```

```
            with tf.variable_scope(end_point):
                net = slim.conv2d(net, 128, [1, 1], scope='conv1x1')
                net = slim.conv2d(net, 256, [3, 3], scope='conv3x3', padding='VALID')
            end_points[end_point] = net
            # Prediction and localisations layers.
            #预测和定位
            predictions = []
            logits = []
            localisations = []
            for i, layer in enumerate(feat_layers):
                with tf.variable_scope(layer + '_box'):
                    #接受特征层的输出，生成类别和位置预测
                    p, l = ssd_multibox_layer(end_points[layer],
                                              num_classes,
                                              anchor_sizes[i],
                                              anchor_ratios[i],
                                              normalizations[i])
                #把每一层的预测收集
                predictions.append(prediction_fn(p))#prediction_fn 为 softmax，预测类别
                logits.append(p)#概率
                localisations.append(l)#预测位置信息
            return predictions, localisations, logits, end_points
-----------------------------------------------------------------------------------------------
#生成一层的 anchor boxes
-----------------------------------------------------------------------------------------------
def ssd_anchor_one_layer(img_shape,#原始图像 shape
                         feat_shape,#特征图 shape
                         sizes,#预设的 box size
                         ratios,#aspect 比例
                         step,#anchor 的层
                         offset=0.5,
                         dtype=np.float32):
    """Computer SSD default anchor boxes for one feature layer.
    Determine the relative position grid of the centers, and the relative
    width and height.
    Arguments:
      feat_shape: Feature shape, used for computing relative position grids;
      size: Absolute reference sizes;
```

```
    ratios: Ratios to use on these features;
    img_shape: Image shape, used for computing height, width relatively to the
      former;
    offset: Grid offset.
Return:
    y, x, h, w: Relative x and y grids, and height and width.
"""
# Compute the position grid: simple way.
# y, x = np.mgrid[0:feat_shape[0], 0:feat_shape[1]]
# y = (y.astype(dtype) + offset) / feat_shape[0]
# x = (x.astype(dtype) + offset) / feat_shape[1]
# Weird SSD-Caffe computation using steps values...
"""
#测试中，参数如下
feat_shapes=[(38, 38), (19, 19), (10, 10), (5, 5), (3, 3), (1, 1)]
anchor_sizes=[(21., 45.),
                          (45., 99.),
                          (99., 153.),
                          (153., 207.),
                          (207., 261.),
                          (261., 315.)]
anchor_ratios=[[2, .5],
                           [2, .5, 3, 1./3],
                           [2, .5, 3, 1./3],
                           [2, .5, 3, 1./3],
                           [2, .5],
                           [2, .5]]
anchor_steps=[8, 16, 32, 64, 100, 300]
offset=0.5
dtype=np.float32
feat_shape=feat_shapes[0]
step=anchor_steps[0]
"""
#测试中，y 和 x 的 shape 为（38,38）（38,38）
#y 的值为
#array([[ 0,    0,    0, ...,    0,    0,    0],
 #    [ 1,    1,    1, ...,    1,    1,    1],
# [ 2,    2,    2, ...,    2,    2,    2],
```

```
#     ...,
 #    [35, 35, 35, ..., 35, 35, 35],
#    [36, 36, 36, ..., 36, 36, 36],
 #    [37, 37, 37, ..., 37, 37, 37]])
y, x = np.mgrid[0:feat_shape[0], 0:feat_shape[1]]
#测试中 y=(y+0.5)×8/300,x=(x+0.5)×8/300
y = (y.astype(dtype) + offset) * step / img_shape[0]
x = (x.astype(dtype) + offset) * step / img_shape[1]

#扩展维度，维度为（38,38,1）
# Expand dims to support easy broadcasting.
y = np.expand_dims(y, axis=-1)
x = np.expand_dims(x, axis=-1)

# Compute relative height and width.
# Tries to follow the original implementation of SSD for the order.
#数值为 2+2
num_anchors = len(sizes) + len(ratios)
#shape 为（4,）
h = np.zeros((num_anchors, ), dtype=dtype)
w = np.zeros((num_anchors, ), dtype=dtype)
# Add first anchor boxes with ratio=1.
#测试中，h[0]=21/300,w[0]=21/300?
h[0] = sizes[0] / img_shape[0]
w[0] = sizes[0] / img_shape[1]
di = 1
if len(sizes) > 1:
    #h[1]=sqrt(21*45)/300
    h[1] = math.sqrt(sizes[0] * sizes[1]) / img_shape[0]
    w[1] = math.sqrt(sizes[0] * sizes[1]) / img_shape[1]
    di += 1
for i, r in enumerate(ratios):
    h[i+di] = sizes[0] / img_shape[0] / math.sqrt(r)
    w[i+di] = sizes[0] / img_shape[1] * math.sqrt(r)
#测试中，y 和 x shape 为（38,38,1）
#h 和 w 的 shape 为（4,）
return y, x, h, w
```

```
#
========================================================
=================== #
# SSD loss function.
#
========================================================
=================== #
def ssd_losses(logits, #预测类别
                       localisations,#预测位置
                       gclasses, #ground truth 类别
                       glocalisations, #ground truth 位置
                       gscores,#ground truth 分数
                       match_threshold=0.5,
                       negative_ratio=3.,
                       alpha=1.,
                       label_smoothing=0.,
                       scope='ssd_losses'):
    """Loss functions for training the SSD 300 VGG network.
    This function defines the different loss components of the SSD, and
    adds them to the TF loss collection.
    Arguments:
      logits: (list of) predictions logits Tensors;
      localisations: (list of) localisations Tensors;
      gclasses: (list of) groundtruth labels Tensors;
      glocalisations: (list of) groundtruth localisations Tensors;
      gscores: (list of) groundtruth score Tensors;
    """
    # Some debugging...
    # for i in range(len(gclasses)):
    #       print(localisations[i].get_shape())
    #       print(logits[i].get_shape())
    #       print(gclasses[i].get_shape())
    #       print(glocalisations[i].get_shape())
    #       print()
    with tf.name_scope(scope):
        l_cross = []
        l_loc = []
```

```
        for i in range(len(logits)):
            with tf.name_scope('block_%i' % i):
                # Determine weights Tensor.
                pmask = tf.cast(gclasses[i] > 0, logits[i].dtype)
                n_positives = tf.reduce_sum(pmask)#正样本数目

                #np.prod 函数 Return the product of array elements over a given
axis
                n_entries = np.prod(gclasses[i].get_shape().as_list())
                # r_positive = n_positives / n_entries
                # Select some random negative entries.
                r_negative = negative_ratio * n_positives / (n_entries - n_positives)#
负样本数
                nmask = tf.random_uniform(gclasses[i].get_shape(),
                                          dtype=logits[i].dtype)
                nmask = nmask * (1. - pmask)
                nmask = tf.cast(nmask > 1. - r_negative, logits[i].dtype)

                #cross_entropy loss
                # Add cross-entropy loss.
                with tf.name_scope('cross_entropy'):
                    # Weights Tensor: positive mask + random negative.
                    weights = pmask + nmask
                    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits[i],
gclasses[i])
                    loss = tf.contrib.losses.compute_weighted_loss(loss, weights)
                    l_cross.append(loss)

                #smooth loss
                # Add localization loss: smooth L1, L2, ...
                with tf.name_scope('localization'):
                    # Weights Tensor: positive mask + random negative.
                    weights = alpha * pmask
                    loss = custom_layers.abs_smooth(localisations[i] -
glocalisations[i])
                    loss = tf.contrib.losses.compute_weighted_loss(loss, weights)
                    l_loc.append(loss)
```

```
        # Total losses in summaries...
        with tf.name_scope('total'):
            tf.summary.scalar('cross_entropy', tf.add_n(l_cross))
            tf.summary.scalar('localization', tf.add_n(l_loc))
```
