南 阳 理 工 学 院

本科生毕业设计(论文)

学院(系)： 软件学院

专 业： 软件工程

学 生： 暴印行

指导教师： 靳冰

完成日期 2018 年 03 月

南阳理工学院本科生毕业设计（论文）

基于YOLOv2的左右手识别  
优化研究与实现

Optimization and Implementation of  
Left and Right Hand Recognition Based on YOLOv2

总 计：毕业设计(论文) 32页

表 格： 1个

图 片： 11个

南 阳 理 工 学 院 本 科 毕 业 设 计(论文)

基于YOLOv2的左右手识别  
优化研究与实现

Optimization and Implementation of  
Left and Right Hand Recognition Based on YOLOv2

学 院(系)： 软件学院

专 业： 软件工程

学 生 姓 名： 暴印行

学 号： 1415925239

指导教师(职称)： 靳冰 讲师

评 阅 教 师：

完 成 日 期： 2018年03月30日

南阳理工学院

Nanyang Institute of Technology

基于YOLOv2的左右手识别  
优化研究与实现

软件工程 暴印行

1. 目前全球最先进的一些物体检测方法都需要设置一个有目的性的分类器来执行检测，而YOLO（You Only Look Once）是一种全新的实时物体检测方法,将边界框和类别概率关联到一个整体，设计成一个简单的回归问题。YOLOv2是YOLO的升级版本，使用多种分辨率数据输入的训练方法，在精度和速度之间做了一个平衡，从而使其性能达到最优。要实现基于YOLOv2的左右手识别，首先需要组建一个网络模型，然后采集并标注大量应用场景数据，进行模型训练及优化，得到权重文件，最后运用这个训练好的权重文件把图片中的左右手识别出来。基于YOLOv2的左右手识别速度和精度都相当高。基础模型速度可以达到45帧每秒，同时Mean AP (平均精度)达到78.6%。与目前的最先进的方法相比，虽然YOLOv2的位置检测误差较大，对小目标检测效果不理想，但是对于背景它有更低的误检，同时，YOLOv2能够轻松学习到更泛化的特征。由于在触控行业，触控要做到实时响应，并且应用场景中摄像头捕获的图片中手的像素宽高均不小于78px，不算是小目标，所以YOLOv2完全可以胜任这项工作。
2. YOLO；深度学习；优化；实现

Optimization and Implementation of  
Left and Right Hand Recognition Based on YOLOv2

Software Engineering Major Bao Yinhang

1. At present, some advanced object detection methods need to set up a purposeful classifier to perform the detection, and YOLO (You Only Look Once) is a brand-new real-time object detection method that associates the bounding box and class probability to a whole, designed as a simple regression problem. YOLOv2 is an upgraded version of YOLO, it uses multiple resolution data training methods to achieve a balance between accuracy and speed, thereby do the best performance. To realize left and right hands recognition based on YOLOv2, we first need to set up a network model, then collect and label a large number of application scene data, run model training and optimization, then get a weight file, and finally use this trained weight file to identify the left and right hands in the image. The speed and accuracy of Right and left hands recognition based on YOLOv2 are quite high. The basic model speed can reach 45 frames per second, while Mean AP (Mean Average Precision) reaches 78.6%. Compared with the current state-of-the-art methods, although YOLOv2's position detection error is larger and the detection effect on small targets is not ideal, but it has lower misdetection on the background, and at the same time, YOLOv2 can easily learn more generalized features. In the touch industry, touch control needs real-time response, and the pixel width of the hand captured by the camera in the application scene is not less than 78px, which is not a small target. Therefore, YOLOv2 is very suitable for this task.
2. YOLO; deep learning; optimization; realization

目 录

[1 绪 论 1](#_Toc512891066)

[1.1 课题的背景、目的和意义 1](#_Toc512891067)

[1.2 要完成的目标 1](#_Toc512891068)

[1.3 主要内容 1](#_Toc512891069)

[1.4 论文的组织结构 2](#_Toc512891070)

[2 理论基础 2](#_Toc512891071)

[2.1 YOLO框架介绍 2](#_Toc512891072)

[2.2 算法相关和相似技术的介绍 4](#_Toc512891073)

[3 算法描述 5](#_Toc512891074)

[3.1 神经网络框架相关的概念 5](#_Toc512891075)

[3.1.1 激活函数 5](#_Toc512891076)

[3.1.2 池化层 5](#_Toc512891077)

[3.1.3 Dropout 5](#_Toc512891078)

[3.1.4 损失函数 5](#_Toc512891079)

[3.1.5 全链接层 5](#_Toc512891080)

[3.2 输入、输出与流程 5](#_Toc512891081)

[4 设计与实现 7](#_Toc512891082)

[4.1 复杂度与核心的程序流程 7](#_Toc512891083)

[4.2 核心功能的主要代码及说明 7](#_Toc512891084)

[4.2.1 主函数 7](#_Toc512891085)

[4.2.2 卷积层 9](#_Toc512891086)

[4.2.3 池化层 14](#_Toc512891087)

[4.2.4 全链接层 16](#_Toc512891088)

[5 优化 18](#_Toc512891089)

[5.1 批正则化 18](#_Toc512891090)

[5.2 高分辨率分类器 19](#_Toc512891091)

[5.3 加锚点框的卷积层 19](#_Toc512891092)

[5.4 维度聚类 20](#_Toc512891093)

[5.5 直接定位预测 21](#_Toc512891094)

[5.6 细粒度特征 22](#_Toc512891095)

[5.7 多尺度训练 23](#_Toc512891096)

[5.8 交叉熵 24](#_Toc512891097)

[6 结果与结论 25](#_Toc512891098)

[6.1 测试与运行效果 25](#_Toc512891099)

[6.2 性能或者效果与同类算法的比较 25](#_Toc512891100)

[结束语 27](#_Toc512891101)

[参考文献 28](#_Toc512891102)

[附录 30](#_Toc512891103)

[致谢 32](#_Toc512891104)

# 绪 论

## 课题的背景、目的和意义

目前全世界的触控屏按照工作原理及传输信息的介质，可将其分成四类：电阻式、电容感应式、红外线式和表面声波式。而把计算机视觉和深度学习用于触控产品上目前还是全球首例。

最近现有的深度学习方法，如R-CNN运用了区域检测方法，首先在图片中生成大量的可能存在的边界框，然后以这些边界框为基础设计分类器。分类后，再剔除误检测的边界框，消除有重复的检测，每种类别在小范围内只留一个，并根据场景中的其它目标从新定位框的位置。这些非常复杂的计算难以优化而且很慢，因为每个独立的组件都必须单独训练。再比如SSD在神经网络最后一层加入一个带有滑动式窗口算法算法的全链接层，它对最后的特征图每个位置都进行了评估，方式是非常经典的，但是将会付出沉重的时间复杂度以及空间复杂度代价。

YOLO相比较以前的方法就显得更加精小，它将目标检测从新设计为单个回归问题，直接将图片划分为7\*7的网格，评估以网格中心为中心的两个边界框，也就是总共98个待处理评估。YOLO只需要在图片上运行一遍神经网络，就能从图片像素直接得出边界框坐标与类的概率。

## 要完成的目标

触控产品需要做到实时响应触控点，这对深度模型的效率要求很高，触控产品还需要做到精准的触控点位，这对深度模型的精度要求也很高，需要一个综合性能很强的神经网络框架才能实现。

左右手识别要做到实时检测，并且Mean AP要高于75%，还要让误检率尽可能少的。

## 主要内容

解决左右手识别问题主要要做两件事：

1.采集数据。需要构建自己的数据集，为了防止过拟合，先在不同环境下拍摄左右手同时触摸、左手单独触摸、右手单独触摸的视频，还要考虑到不同的光照环境、广泛的用户、高低对比度、多噪声少噪声等环境因素。然后把视频按帧拆分成图片，用标注工具一张一张的标注左右手位置信息，最后把这些数据按照一定格式（VOC）整合到一个数据集中。

2.设计网络模型与训练优化。编写网络模型配置文件，将YOLOv2原有的平方差损失函数替换成交叉熵损失函数，然后根据训练结果调整超参数，使网络模型达到最优状态，保存权重文件，有网络模型配置文件与权重文件就可以用于左右手识别了

## 论文的组织结构

论文中主要讲解一下YOLOv2的工作基本思路以及优化工作。先简单的讲解一下算法的全过程，然后再细分技术块讲解，最后讲一下网络模型的测试与运行效果。

# 理论基础

## YOLO框架介绍

这里先大概陈述一下YOLO的检测基本思路，下边再细分模块讲解每一步的基本思路。

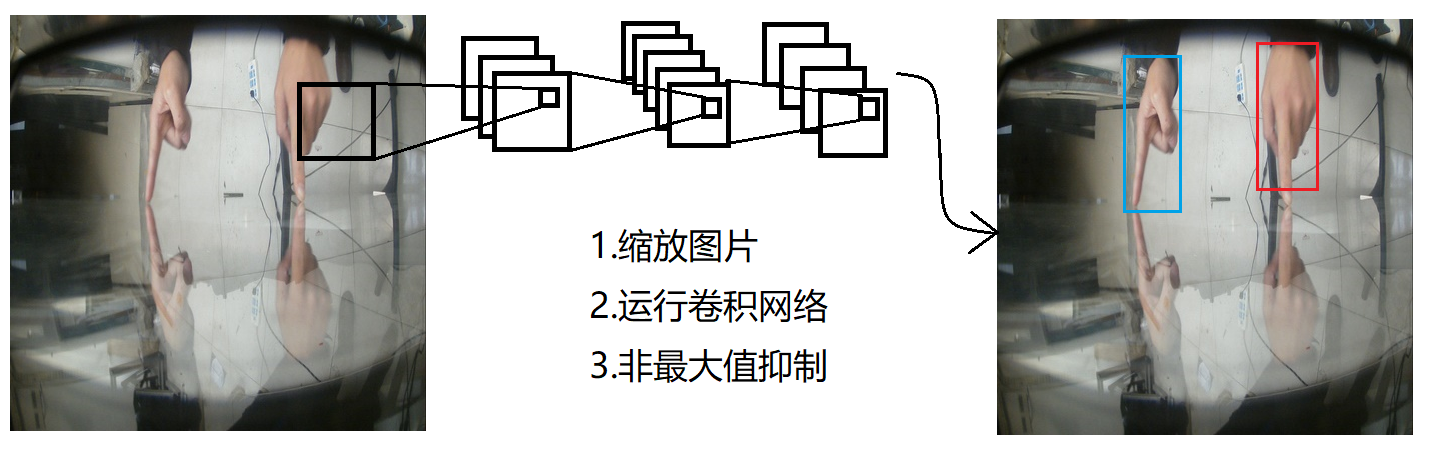


图2-1：YOLOv2的检测系统。

YOLO很精巧。如图2-1中所示，用YOLO处理图片非常简单而且直接。YOLO将输入图片大小调整为448px\*448px，在输入图片上运行单个卷积网络，以及用边界框的置信度与网络阈值对比，留下最置信度最高的边界框。另外YOLO卷积网络可以同时预测所有格子的多个边界框与类别的概率。

YOLO用全图片进行训练而且可以直接优化检测性能。这种联合边界框和类概率的模型与传统的物体检测方法相比大致有三大优点与一个缺点：

YOLO速度非常快。由于它将检测视为简单的回归问题，所以不再需要复杂计算。测试的时候，它只需要在一张新图片上运行神经网络来预测检测结果。没有批处理的情况下，在Titan X Ti GPU上，它的基础网络运行速度为每秒四十五帧，这意味着它可以在不到二十五毫秒每帧的效率实时地处理视频。另外，YOLO的平均精度达到了其他实时目标检测系统两倍还多。

YOLO替换掉效率低下的滑动式窗口算法。YOLO进行预测时，和基于滑动式窗口算法和RPN(区域提议网络)的技术不同，YOLO在训练和测试期间能覆盖到整个图片，所以它能够隐式地编码分类的上下文信息以及它们的表层特征。Fast R-CNN是一种顶尖的检测方法，但是它看不到上下文信息，所以在图片中出现目标的背景补丁的错误。和Fast R-CNN相比，YOLO的错误背景数量少了一半。

YOLO学习物体的一般化表示。在对自然图片进行训练并在验证集上进行测试时，YOLO已经远超DPM和R-CNN等顶尖检测方法。由于YOLO具有高度的泛化性能，降低了YOLO在应用于新的应用场景时发生故障的概率。

YOLO对小目标检测效果差。YOLO的精度仍然落后于最先进的目标检测系统。虽然它可以迅速识别图片中的物体，但是它不能准确的把小物体边界用边界框圈住。因为每个格子中只能留一个类别，一张图片只有7\*7个格子，比较稀疏，所以当一个格子中有多个目标物体的时候，只会留可信度最高的那个，可信度较小的那个就会被过滤掉。

YOLO将目标检测的传统的零散组件集成到单个神经网络中。YOLO运用整个图片的特征来预测每个边界框。它还同时预测了图片中所有类的所有边界框。这意味着YOLO涵盖了整个图片和图片中的所有的目标。这使它保持了较高的平均精度并且还能实时检测和端到端训练。

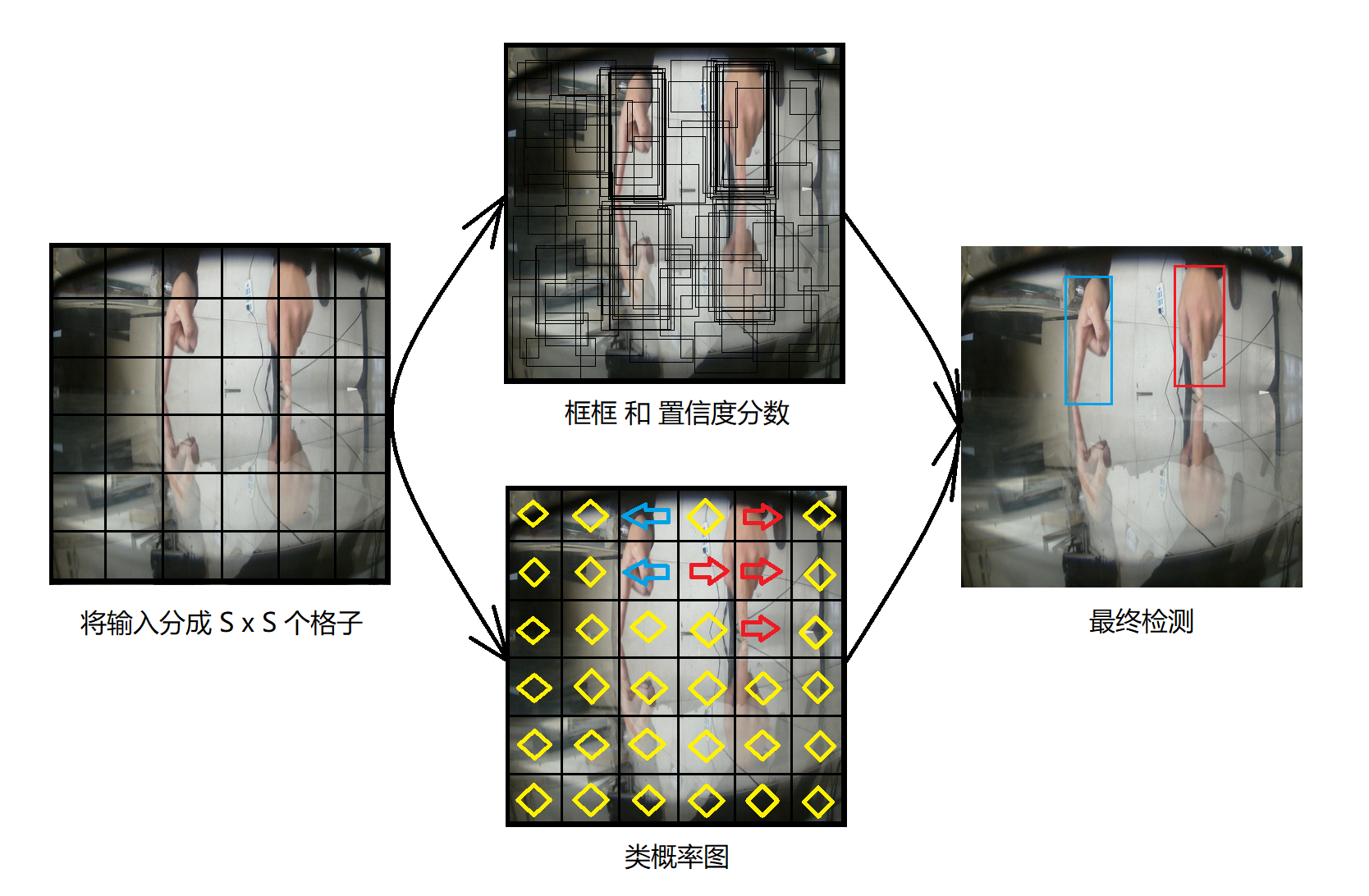


图2-2：YOLO模型将框和类概率关联在一个模型中

YOLO先将输入图片划分为S\*S的格子。 假如目标的中心位置出现在一个格子中，则该格子就会检测到该目标。 每个格子预测B 个（默认两个）bounding box、一个置信度分数和C个类的概率，这些预测将会生成为一个大小的S\*S\*（B\*5＋C）的张量作为输出。

置信度分数反映了该网络模型认为格子中涵盖某个目标的概率。通常， 将置信度分数定义为,其中Pr（Object）={0,1}。假如该格子中不存在任何目标，则置信度分数就为零。置信度分数等于预测框和标注框之间的交集和并集的比值（IOU）。



每个bounding box由五个属性值组成：confidence（置信度分数）和x，y，w，h。 （x，y）坐标表示框的正中心。宽度和高度是相对于整个图片百分比的数值。每个格子还预测C个类中每个类的概率Pr（Class i|Object）。这些概率取决于存在目标的格子。YOLO只预测了每个格子的类别概率最高的一组，跟bounding box B的数量没关系。

## 算法相关和相似技术的介绍

将YOLO检测系统与目前几种顶尖检测系统比较运行效率，简单描述一下关键的差别与相似性。

可调整模块模型。可调整模块模型（Deformable Part Model，DPM）运用可滑动的窗口的方法进行物体的检测。 DPM采用了改进后的方向梯度直方图（Histogram of Oriented Gradient, HOG）特征、获取静态特征、对区域进行分类、预测高评估区域的边界框等。YOLO用单个卷积神经网络融合了所有功能。网络可以同时执行特征获取、边界框预测、非最大抑制与上下文信息推理。YOLO可以在线执行训练功能与优化检测任务。这种统一型架构造就了YOLO的快、准、狠。

循环卷积网络。循环卷积网络（Regions with CNN features，R-CNN）及其变体则运用区域提议而不是滑动式窗口算法来查找图片中的目标。用选择性搜索方法生成可能存在的边界框，用卷积网络获取特征，用支持向量机(Support Vector Machine, SVM)对框进行评估，用线性模型调整边界框，用非最大抑制法消除重复检测。这个复杂流水线的每个阶段都必须独立地进行精确调整，并且是循环网络，所以得到的系统非常缓慢，在测试时间每个图片至少需要40秒。YOLO与R-CNN有一些相似的地方，每个格子预测可能存在的边界框并运用卷积网络对这些框进行评估，除此之外，YOLO对格子检测结果设置限制，这有效的避免了对同一目标的多次检测。YOLO只需要极少的边界框就够了，每张图片只有九十八个，而选择性搜索则约有两千个。

最后，YOLO将各个独立的组件组合成一个单个、可共同优化的模型。其它快速目标检测器如Fast R-CNN把注意力放在共享权值上并运用神经网络来提取区域特征，而不是采用选择性搜索加速R-CNN框架。虽然它们俩都得到了比R-CNN更快的速度与更高的精度的改进，但两者仍然做不到实时检测。

# 算法描述

## 神经网络框架相关的概念

### 激活函数

激活函数有很多，但常用的有四个：sigmoid、tahn、relu、leaky relu。大体功能都很相似，都是把数据一个映射到一个小的区间，但有微小差别:sigmoid 把数据映射到[0,1]；tahn把数据映射到[-1,1]；relu负数全部映射到0，但损失一些特征；leaky relu负数乘以一个小于1的正数值。

### 池化层

池化层通常放在卷积层的后面，一般卷积层输出的特征图比较冗余，池化层的目标是消减卷积层输出的信息，从而节省一些内存开销，减小权值的冗余，更具体一点就是，池化层可以把卷积层的输出作为它的输入并且输出一个更加紧凑的特征向量传递给下层网络。

### Dropout

Dropout是为防止模型过拟合的而生的一种手段。它的基本思想是，随机屏蔽某块区域的特征图，使其暂时不参与本轮的运算过程。这样做，模型可以更加的鲁棒，因为这样做，模型才不会过度的以来某些局部特征使最终的训练方向跑偏。到了下一轮的训练中，这块区域还是有可能参与运算的，因为这个过程是随机的，它只是阻止本轮的这块区域参与运算。

### 损失函数

损失函数（loss function）是用来评估网络模型的预测值与真实值的差异程度，然后通过反向传播用这个误差更新权重。它的输出通常是一个不小于零的实数,损失函数越小，模型的稳定性就越好。平均精度越高，模型的性能就越好。损失函数是反向传播的核心部分。

### 全链接层

全链接层（fully connected layers，FC）在整个卷积的神经网络中饰演着 “分类器”的角色。如果说池化层和卷积层等层结构是将输入数据映射到零散特征图的话，全链接层就起到将前边卷积层学到的零散的特征映射到联系紧凑的特征图的作用。然后根据这个紧凑的特征图就可以把目标分类。

## 输入、输出与流程

网络在检测的时候，只需要输入图片（输入视频的时候，网络会把视频拆分成图片），网络经过卷积、池化、全连接计算，输出一个tensor作为检测结果，将tensor内容表示在图片中。

用图片测试：

./yolo\_windows detector test ./doubleHands.data ./tiny-yolo.cfg ./backup/tiny-youlo\_31000.weights ./0001.jpg

用视频测试：

./yolo\_windows detector demo ./doubleHands.data ./tiny-yolo.cfg ./backup/tiny-youlo\_31000.weights ./0001.avi

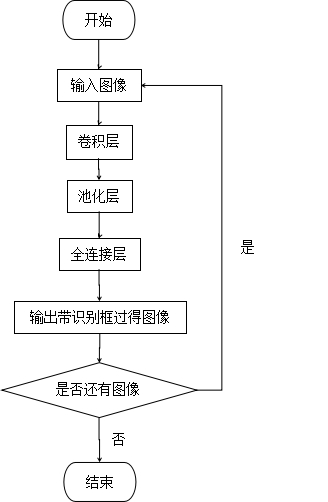


图3-2：YOLO测试流程图

# 设计与实现

## 复杂度与核心的程序流程

YOLO的网络架构的思想来自于GoogLeNet图片分类模型。可实现实时检测和端到端训练，同时还能保持较高的平均精度。YOLO有二十四个卷积层，后接两个全链接层。与GoogLeNet运用的初始模块不同的是，YOLO简单地运用了1\*1的缩放层，然后是3\*3的卷积层,完整的网络如图4-1所示：

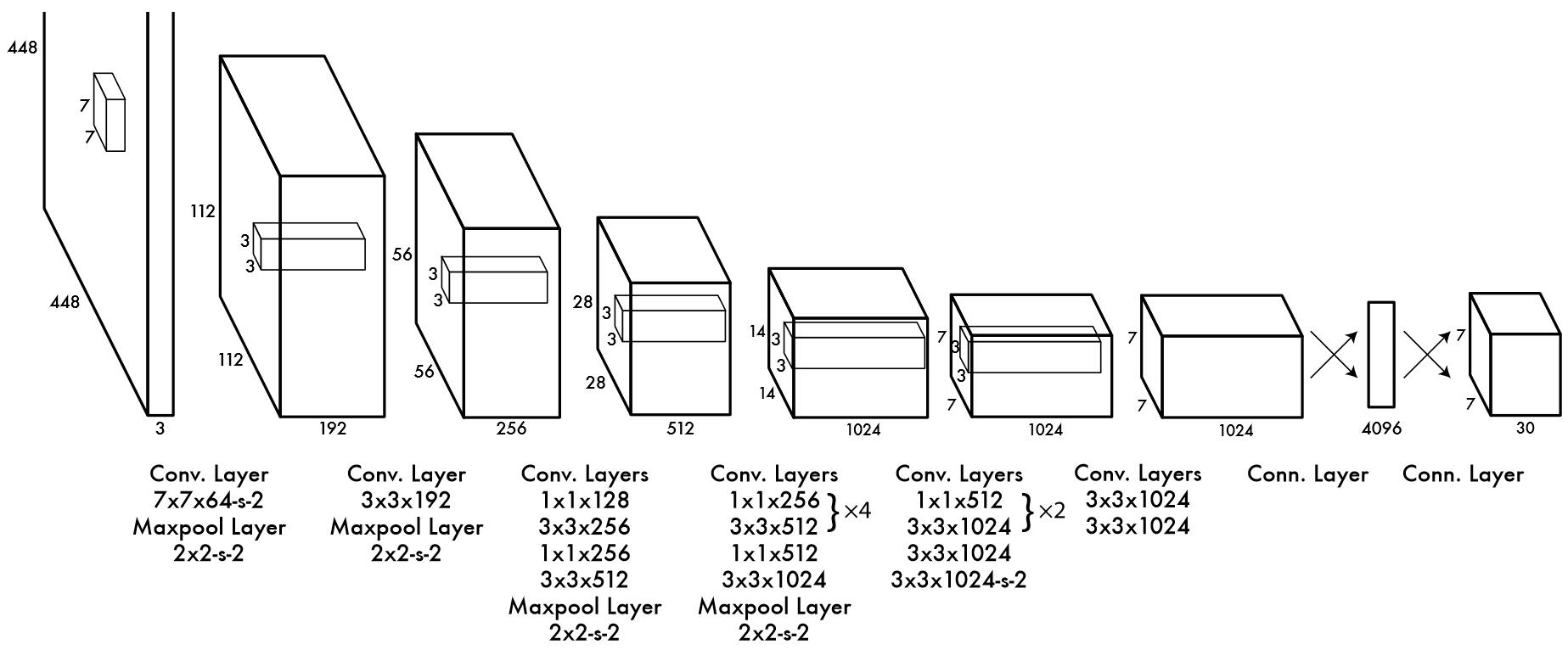


图4-1：YOLO网络结构

YOLO有二十四个卷积层，其次是两个全链接层。YOLO的卷积层从图片中一步步获取越来越深的特征，全链接层预测输出的类的概率与位置，最终输出的是7\*7\*30的特征张量。

## 核心功能的主要代码及说明

### 主函数

主函数是整个模型的入口，它定义了很多功能分支入口，比如训练的时候传入train关键字，用图片测试时传入test关键字，用视频测试时用demo关键字。

int main(int argc, char \*\*argv)

{

//test\_resize("data/hand.jpg");

//test\_box();

//test\_convolutional\_layer();

if(argc < 2){

fprintf(stderr, "usage: %s <function>\n", argv[0]);

return 0;

}

gpu\_index = find\_int\_arg(argc, argv, "-i", 0);

if(find\_arg(argc, argv, "-nogpu")) {

gpu\_index = -1;

}

#ifndef GPU //GPU优化

gpu\_index = -1;

#else

if(gpu\_index >= 0){

cuda\_set\_device(gpu\_index);

}

#endif

if (0 == strcmp(argv[1], "average")){ //采用均值方法

average(argc, argv);

} else if (0 == strcmp(argv[1], "yolo")){

run\_yolo(argc, argv);

} else if (0 == strcmp(argv[1], "voxel")){

run\_voxel(argc, argv);

} else if (0 == strcmp(argv[1], "super")){

run\_super(argc, argv);

} else if (0 == strcmp(argv[1], "detector")){//采用YOLOv2网络框架

run\_detector(argc, argv);

} else if (0 == strcmp(argv[1], "detect")){//采用YOLOv1网络框架

float thresh = find\_float\_arg(argc, argv, "-thresh", .24);//设置阈值，用于过滤可信度过低的分类

char \*filename = (argc > 4) ? argv[4]: 0;

test\_detector("cfg/coco.data", argv[2], argv[3], filename, thresh);

} else if (0 == strcmp(argv[1], "cifar")){

run\_cifar(argc, argv);

} else if (0 == strcmp(argv[1], "go")){

run\_go(argc, argv);

} else if (0 == strcmp(argv[1], "rnn")){

run\_char\_rnn(argc, argv);

} else if (0 == strcmp(argv[1], "vid")){

run\_vid\_rnn(argc, argv);

} else if (0 == strcmp(argv[1], "coco")){

run\_coco(argc, argv);

} else if (0 == strcmp(argv[1], "classify")){

predict\_classifier("cfg/imagenet1k.data", argv[2], argv[3], argv[4], 5);

} else if (0 == strcmp(argv[1], "classifier")){

run\_classifier(argc, argv);

} else if (0 == strcmp(argv[1], "art")){

run\_art(argc, argv);

} else if (0 == strcmp(argv[1], "tag")){

run\_tag(argc, argv);

} else if (0 == strcmp(argv[1], "compare")){

run\_compare(argc, argv);

} else if (0 == strcmp(argv[1], "dice")){

run\_dice(argc, argv);

} else if (0 == strcmp(argv[1], "writing")){

run\_writing(argc, argv);

} else if (0 == strcmp(argv[1], "3d")){

composite\_3d(argv[2], argv[3], argv[4], (argc > 5) ? atof(argv[5]) : 0);

} else if (0 == strcmp(argv[1], "test")){//用于测试模型

test\_resize(argv[2]);

} else if (0 == strcmp(argv[1], "captcha")){

run\_captcha(argc, argv);

} else if (0 == strcmp(argv[1], "nightmare")){

run\_nightmare(argc, argv);

} else if (0 == strcmp(argv[1], "rgbgr")){

rgbgr\_net(argv[2], argv[3], argv[4]);

} else if (0 == strcmp(argv[1], "reset")){

reset\_normalize\_net(argv[2], argv[3], argv[4]);

} else if (0 == strcmp(argv[1], "denormalize")){

denormalize\_net(argv[2], argv[3], argv[4]);

} else if (0 == strcmp(argv[1], "statistics")){

statistics\_net(argv[2], argv[3]);

} else if (0 == strcmp(argv[1], "normalize")){

normalize\_net(argv[2], argv[3], argv[4]);

} else if (0 == strcmp(argv[1], "rescale")){

rescale\_net(argv[2], argv[3], argv[4]);

} else if (0 == strcmp(argv[1], "ops")){

operations(argv[2]);

} else if (0 == strcmp(argv[1], "speed")){

speed(argv[2], (argc > 3 && argv[3]) ? atoi(argv[3]) : 0);

} else if (0 == strcmp(argv[1], "oneoff")){

oneoff(argv[2], argv[3], argv[4]);

} else if (0 == strcmp(argv[1], "partial")){

partial(argv[2], argv[3], argv[4], atoi(argv[5]));

} else if (0 == strcmp(argv[1], "average")){

average(argc, argv);

} else if (0 == strcmp(argv[1], "visualize")){

visualize(argv[2], (argc > 3) ? argv[3] : 0);

} else if (0 == strcmp(argv[1], "imtest")){

test\_resize(argv[2]);

} else {

fprintf(stderr, "Not an option: %s\n", argv[1]);

}

return 0;

}

### 卷积层

简单来说，卷积层是获取特征的网络层，卷积层越深，获取的特征越详细，随之而来的就是臃肿的权重文件。所以说卷积层不是越少越好、越多越好，而是要找到详细与臃肿平衡的那个点。

输入是前边那一层输出的特征图，输出是从这个特征图中卷积出来的更加详细的特征图，这是正向传播。而卷积核将在反向传播中根据损失函数调整更新，它将会保存在权重文件中，这个权重文件就是类似人类大脑的记忆。当训练次数增多时，权重就更接近真相。当训练次数更多的时候，模型就会产生过拟合。

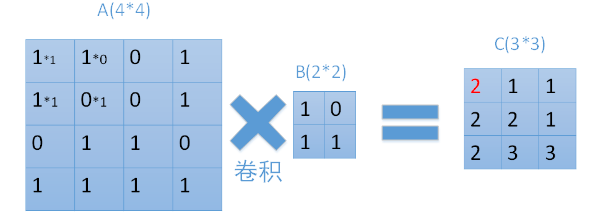


图4-2-1：卷积层工作原理

其中：A:网络输入，B:卷积核，C:网络输出（特征值）

创建卷积层

convolutional\_layer make\_convolutional\_layer(int batch, int h, int w, int c, int n, int size, int stride, int padding, ACTIVATION activation, int batch\_normalize, int binary, int xnor, int adam)

{

int i;

convolutional\_layer l = {0};

l.type = CONVOLUTIONAL;

l.h = h;

l.w = w;

l.c = c;

l.n = n;

//传递超参数

l.binary = binary;

l.xnor = xnor;

l.batch = batch;

l.stride = stride;

l.size = size;

l.pad = padding;

l.batch\_normalize = batch\_normalize;

l.weights = calloc(c\*n\*size\*size, sizeof(float));

l.weight\_updates = calloc(c\*n\*size\*size, sizeof(float));

l.biases = calloc(n, sizeof(float));

l.bias\_updates = calloc(n, sizeof(float));

// float scale = 1./sqrt(size\*size\*c);

float scale = sqrt(2./(size\*size\*c));

for(i = 0; i < c\*n\*size\*size; ++i) l.weights[i] = scale\*rand\_uniform(-1, 1);

int out\_h = convolutional\_out\_height(l);

int out\_w = convolutional\_out\_width(l);

l.out\_h = out\_h;

l.out\_w = out\_w;

l.out\_c = n;

l.outputs = l.out\_h \* l.out\_w \* l.out\_c;

l.inputs = l.w \* l.h \* l.c;

l.output = calloc(l.batch\*l.outputs, sizeof(float));

l.delta = calloc(l.batch\*l.outputs, sizeof(float));

l.forward = forward\_convolutional\_layer;

l.backward = backward\_convolutional\_layer;

l.update = update\_convolutional\_layer;

if(binary){

l.binary\_weights = calloc(c\*n\*size\*size, sizeof(float));

l.cweights = calloc(c\*n\*size\*size, sizeof(char));

l.scales = calloc(n, sizeof(float));

}

if(xnor){

l.binary\_weights = calloc(c\*n\*size\*size, sizeof(float));

l.binary\_input = calloc(l.inputs\*l.batch, sizeof(float));

}

//为批正则化申请内存

if(batch\_normalize){

l.scales = calloc(n, sizeof(float));

l.scale\_updates = calloc(n, sizeof(float));

for(i = 0; i < n; ++i){

l.scales[i] = 1;

}

l.mean = calloc(n, sizeof(float));

l.variance = calloc(n, sizeof(float));

l.mean\_delta = calloc(n, sizeof(float));

l.variance\_delta = calloc(n, sizeof(float));

l.rolling\_mean = calloc(n, sizeof(float));

l.rolling\_variance = calloc(n, sizeof(float));

l.x = calloc(l.batch\*l.outputs, sizeof(float));

l.x\_norm = calloc(l.batch\*l.outputs, sizeof(float));

}

if(adam){

l.adam = 1;

l.m = calloc(c\*n\*size\*size, sizeof(float));

l.v = calloc(c\*n\*size\*size, sizeof(float));

}

#ifdef GPU //GPU加速

l.forward\_gpu = forward\_convolutional\_layer\_gpu;

l.backward\_gpu = backward\_convolutional\_layer\_gpu;

l.update\_gpu = update\_convolutional\_layer\_gpu;

if(gpu\_index >= 0){

if (adam) {

l.m\_gpu = cuda\_make\_array(l.m, c\*n\*size\*size);

l.v\_gpu = cuda\_make\_array(l.v, c\*n\*size\*size);

}

l.weights\_gpu = cuda\_make\_array(l.weights, c\*n\*size\*size);

l.weight\_updates\_gpu = cuda\_make\_array(l.weight\_updates, c\*n\*size\*size);

l.biases\_gpu = cuda\_make\_array(l.biases, n);

l.bias\_updates\_gpu = cuda\_make\_array(l.bias\_updates, n);

l.delta\_gpu = cuda\_make\_array(l.delta, l.batch\*out\_h\*out\_w\*n);

l.output\_gpu = cuda\_make\_array(l.output, l.batch\*out\_h\*out\_w\*n);

if(binary){

l.binary\_weights\_gpu = cuda\_make\_array(l.weights, c\*n\*size\*size);

}

if(xnor){

l.binary\_weights\_gpu = cuda\_make\_array(l.weights, c\*n\*size\*size);

l.binary\_input\_gpu = cuda\_make\_array(0, l.inputs\*l.batch);

}

if(batch\_normalize){

l.mean\_gpu = cuda\_make\_array(l.mean, n);

l.variance\_gpu = cuda\_make\_array(l.variance, n);

l.rolling\_mean\_gpu = cuda\_make\_array(l.mean, n);

l.rolling\_variance\_gpu = cuda\_make\_array(l.variance, n);

l.mean\_delta\_gpu = cuda\_make\_array(l.mean, n);

l.variance\_delta\_gpu = cuda\_make\_array(l.variance, n);

l.scales\_gpu = cuda\_make\_array(l.scales, n);

l.scale\_updates\_gpu = cuda\_make\_array(l.scale\_updates, n);

l.x\_gpu = cuda\_make\_array(l.output, l.batch\*out\_h\*out\_w\*n);

l.x\_norm\_gpu = cuda\_make\_array(l.output, l.batch\*out\_h\*out\_w\*n);

}

#ifdef CUDNN

cudnnCreateTensorDescriptor(&l.srcTensorDesc);

cudnnCreateTensorDescriptor(&l.dstTensorDesc);

cudnnCreateFilterDescriptor(&l.weightDesc);

cudnnCreateTensorDescriptor(&l.dsrcTensorDesc);

cudnnCreateTensorDescriptor(&l.ddstTensorDesc);

cudnnCreateFilterDescriptor(&l.dweightDesc);

cudnnCreateConvolutionDescriptor(&l.convDesc);

cudnn\_convolutional\_setup(&l);

#endif

}

#endif

l.workspace\_size = get\_workspace\_size(l);

l.activation = activation;

fprintf(stderr, "conv %5d %2d x%2d /%2d %4d x%4d x%4d -> %4d x%4d x%4d\n", n, size, size, stride, w, h, c, l.out\_w, l.out\_h, l.out\_c);

return l;

}

前向传播，从输入提取特征：

void forward\_convolutional\_layer(convolutional\_layer l, network\_state state)

{

int out\_h = convolutional\_out\_height(l);

int out\_w = convolutional\_out\_width(l);

int i;

fill\_cpu(l.outputs\*l.batch, 0, l.output, 1);

if(l.xnor){

binarize\_weights(l.weights, l.n, l.c\*l.size\*l.size, l.binary\_weights);

swap\_binary(&l);

binarize\_cpu(state.input, l.c\*l.h\*l.w\*l.batch, l.binary\_input);

state.input = l.binary\_input;

}

int m = l.n;

int k = l.size\*l.size\*l.c;

int n = out\_h\*out\_w;

float \*a = l.weights;

float \*b = state.workspace;

float \*c = l.output;

for(i = 0; i < l.batch; ++i){

im2col\_cpu(state.input, l.c, l.h, l.w,

l.size, l.stride, l.pad, b);

gemm(0,0,m,n,k,1,a,k,b,n,1,c,n);

c += n\*m;

state.input += l.c\*l.h\*l.w;

}

if(l.batch\_normalize){

forward\_batchnorm\_layer(l, state);

}

add\_bias(l.output, l.biases, l.batch, l.n, out\_h\*out\_w);

activate\_array(l.output, m\*n\*l.batch, l.activation);

if(l.binary || l.xnor) swap\_binary(&l);

}

反向传播，返回loss调节卷积层权重：

void backward\_convolutional\_layer(convolutional\_layer l, network\_state state)

{

int i;

int m = l.n;

int n = l.size\*l.size\*l.c;

int k = convolutional\_out\_height(l)\*

convolutional\_out\_width(l);

gradient\_array(l.output, m\*k\*l.batch, l.activation, l.delta);

backward\_bias(l.bias\_updates, l.delta, l.batch, l.n, k);

if(l.batch\_normalize){

backward\_batchnorm\_layer(l, state);

}

for(i = 0; i < l.batch; ++i){

float \*a = l.delta + i\*m\*k;

float \*b = state.workspace;

float \*c = l.weight\_updates;

float \*im = state.input+i\*l.c\*l.h\*l.w;

im2col\_cpu(im, l.c, l.h, l.w,

l.size, l.stride, l.pad, b);

gemm(0,1,m,n,k,1,a,k,b,k,1,c,n);

if(state.delta){

a = l.weights;

b = l.delta + i\*m\*k;

c = state.workspace;

gemm(1,0,n,k,m,1,a,n,b,k,0,c,k);

col2im\_cpu(state.workspace, l.c, l.h, l.w, l.size, l.stride, l.pad, state.delta+i\*l.c\*l.h\*l.w);

}

}

}

### 池化层

主流的池化层有Max pooling与Mean pooling，无论哪一个，都没有需要保存的卷积核，也就是权重。所以，在卷积神经网络的训练中，Pooling层需要做的仅仅是将特征图压缩传递到上一层，而没有梯度的计算。

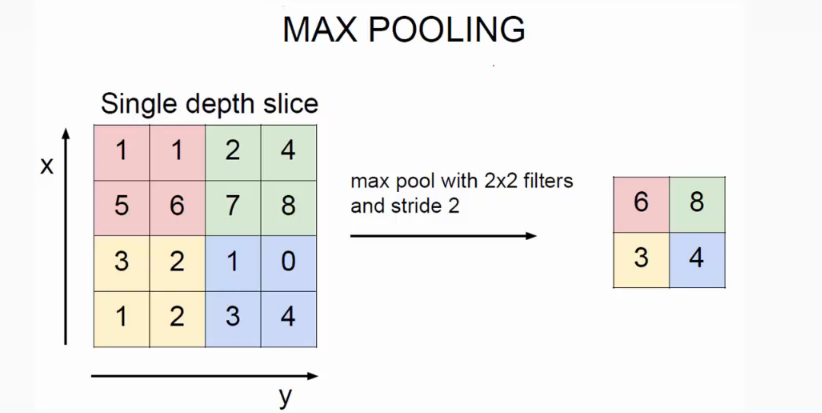


图4-2：max pooling

（1）Max pooling层：在输入从特征图上，以2\*2的筛选区域范围找到最大值输出，筛选区域挪动的步长为2。

（2）Mean pooling层：在输入从特征图上，以2\*2的筛选区域范围求平均值输出，筛选区域挪动的步长为2。

以上筛选区域大小和步长都是可以通过超参数调节的。

以学术界前辈的经验来看，Max pooling要比 Mean pooling更有效。

YOLO中采用的就是Max pooling，基本思路如图4-2所示。

创建池化层

maxpool\_layer make\_maxpool\_layer(int batch, int h, int w, int c, int size, int stride, int padding)

{

maxpool\_layer l = {0};

l.type = MAXPOOL;

l.batch = batch;

l.h = h;

l.w = w;

l.c = c;

l.pad = padding;

l.out\_w = (w + 2\*padding)/stride;

l.out\_h = (h + 2\*padding)/stride;

l.out\_c = c;

l.outputs = l.out\_h \* l.out\_w \* l.out\_c;

l.inputs = h\*w\*c;

l.size = size;

l.stride = stride;

int output\_size = l.out\_h \* l.out\_w \* l.out\_c \* batch;

l.indexes = calloc(output\_size, sizeof(int));

l.output = calloc(output\_size, sizeof(float));

l.delta = calloc(output\_size, sizeof(float));

l.forward = forward\_maxpool\_layer;

l.backward = backward\_maxpool\_layer;

#ifdef GPU

l.forward\_gpu = forward\_maxpool\_layer\_gpu;

l.backward\_gpu = backward\_maxpool\_layer\_gpu;

l.indexes\_gpu = cuda\_make\_int\_array(output\_size);

l.output\_gpu = cuda\_make\_array(l.output, output\_size);

l.delta\_gpu = cuda\_make\_array(l.delta, output\_size);

#endif

fprintf(stderr, "max %d x %d / %d %4d x%4d x%4d -> %4d x%4d x%4d\n", size, size, stride, w, h, c, l.out\_w, l.out\_h, l.out\_c);

return l;

}

前向传播，从输入特征提炼理想特征，丢弃不理想特征，结果会丢弃75%的特征：

void forward\_maxpool\_layer(const maxpool\_layer l, network\_state state)

{

int b,i,j,k,m,n;

int w\_offset = -l.pad;

int h\_offset = -l.pad;

int h = l.out\_h;

int w = l.out\_w;

int c = l.c;

for(b = 0; b < l.batch; ++b){

for(k = 0; k < c; ++k){

for(i = 0; i < h; ++i){

for(j = 0; j < w; ++j){

int out\_index = j + w\*(i + h\*(k + c\*b));

float max = -FLT\_MAX;

int max\_i = -1;

for(n = 0; n < l.size; ++n){

for(m = 0; m < l.size; ++m){

int cur\_h = h\_offset + i\*l.stride + n;

int cur\_w = w\_offset + j\*l.stride + m;

int index = cur\_w + l.w\*(cur\_h + l.h\*(k + b\*l.c));

int valid = (cur\_h >= 0 && cur\_h < l.h &&

cur\_w >= 0 && cur\_w < l.w);

float val = (valid != 0) ? state.input[index] : -FLT\_MAX;

max\_i = (val > max) ? index : max\_i;

max = (val > max) ? val : max;

}

}

l.output[out\_index] = max;

l.indexes[out\_index] = max\_i;

}

}

}

}

}

反向传播，返回loss调节池化层权重：

void backward\_maxpool\_layer(const maxpool\_layer l, network\_state state)

{

int i;

int h = l.out\_h;

int w = l.out\_w;

int c = l.c;

for(i = 0; i < h\*w\*c\*l.batch; ++i){

int index = l.indexes[i];

state.delta[index] += l.delta[i];

}

}

缩放池化层，使其适应它之前各种大小的卷积层。

void resize\_maxpool\_layer(maxpool\_layer \*l, int w, int h)

{

l->h = h;

l->w = w;

l->inputs = h\*w\*l->c;

l->out\_w = (w + 2\*l->pad)/l->stride;

l->out\_h = (h + 2\*l->pad)/l->stride;

l->outputs = l->out\_w \* l->out\_h \* l->c;

int output\_size = l->outputs \* l->batch;

l->indexes = realloc(l->indexes, output\_size \* sizeof(int));

l->output = realloc(l->output, output\_size \* sizeof(float));

l->delta = realloc(l->delta, output\_size \* sizeof(float));

#ifdef GPU

cuda\_free((float \*)l->indexes\_gpu);

cuda\_free(l->output\_gpu);

cuda\_free(l->delta\_gpu);

l->indexes\_gpu = cuda\_make\_int\_array(output\_size);

l->output\_gpu = cuda\_make\_array(l->output, output\_size);

l->delta\_gpu = cuda\_make\_array(l->delta, output\_size);

#endif

}

### 全链接层

前向传播，把前边提炼出来的零散特征进行整合，映射到一个联系紧凑的特征向量里，这个联系紧凑的特征向量可以解析成展现在图片中的边界框以及对应的概率：

void forward\_connected\_layer(connected\_layer l, network\_state state)

{

int i;

fill\_cpu(l.outputs\*l.batch, 0, l.output, 1);

int m = l.batch;

int k = l.inputs;

int n = l.outputs;

float \*a = state.input;

float \*b = l.weights;

float \*c = l.output;

gemm(0,1,m,n,k,1,a,k,b,k,1,c,n);

if(l.batch\_normalize){

if(state.train){

mean\_cpu(l.output, l.batch, l.outputs, 1, l.mean);

variance\_cpu(l.output, l.mean, l.batch, l.outputs, 1, l.variance);

scal\_cpu(l.outputs, .95, l.rolling\_mean, 1);

axpy\_cpu(l.outputs, .05, l.mean, 1, l.rolling\_mean, 1);

scal\_cpu(l.outputs, .95, l.rolling\_variance, 1);

axpy\_cpu(l.outputs, .05, l.variance, 1, l.rolling\_variance, 1);

copy\_cpu(l.outputs\*l.batch, l.output, 1, l.x, 1);

normalize\_cpu(l.output, l.mean, l.variance, l.batch, l.outputs, 1);

copy\_cpu(l.outputs\*l.batch, l.output, 1, l.x\_norm, 1);

} else {

normalize\_cpu(l.output, l.rolling\_mean, l.rolling\_variance, l.batch, l.outputs, 1);

}

scale\_bias(l.output, l.scales, l.batch, l.outputs, 1);

}

for(i = 0; i < l.batch; ++i){

axpy\_cpu(l.outputs, 1, l.biases, 1, l.output + i\*l.outputs, 1);

}

activate\_array(l.output, l.outputs\*l.batch, l.activation);

}

反向传播，返回loss调节全链接层权重：

void backward\_connected\_layer(connected\_layer l, network\_state state)

{

int i;

gradient\_array(l.output, l.outputs\*l.batch, l.activation, l.delta);

for(i = 0; i < l.batch; ++i){

axpy\_cpu(l.outputs, 1, l.delta + i\*l.outputs, 1, l.bias\_updates, 1);

}

if(l.batch\_normalize){

backward\_scale\_cpu(l.x\_norm, l.delta, l.batch, l.outputs, 1, l.scale\_updates);

scale\_bias(l.delta, l.scales, l.batch, l.outputs, 1);

mean\_delta\_cpu(l.delta, l.variance, l.batch, l.outputs, 1, l.mean\_delta);

variance\_delta\_cpu(l.x, l.delta, l.mean, l.variance, l.batch, l.outputs, 1, l.variance\_delta);

normalize\_delta\_cpu(l.x, l.mean, l.variance, l.mean\_delta, l.variance\_delta, l.batch, l.outputs, 1, l.delta);

}

int m = l.outputs;

int k = l.batch;

int n = l.inputs;

float \*a = l.delta;

float \*b = state.input;

float \*c = l.weight\_updates;

gemm(1,0,m,n,k,1,a,m,b,n,1,c,n);

m = l.batch;

k = l.outputs;

n = l.inputs;

a = l.delta;

b = l.weights;

c = state.delta;

if(c) gemm(0,0,m,n,k,1,a,k,b,n,1,c,n);

}

由于全链接层权重冗余严重，仅全链接层权重就可以占据整个网络权重五分之四左右的比重，最近一些性能优越的网络模型如GoogLeNet与ResNet等都采用全局平均池化（Global Average Pooling，GAP）取代全链接层来结合卷积网络学到的深度特征，最后仍然使用softmax等损失函数作为网络惩罚函数。以YOLOv2为例，对416\*416\*3的输入，最后一层卷积可得输出为7\*7\*30的向量输出，如果全链接层含4096个参数，则可用卷积核为7\*7\*30\*4096的全局卷积来替代这一全连接运算过程。

# 优化

## 批正则化

批正则化（Batch Normalization，BN）可以加快模型收敛的速度，减少过拟合发生。 在所有卷积层前应用了Batch Normalization，使结果提升了2%。同时，应用了Batch Normalization，即使去除了dropout，也不会过拟合。

layer make\_normalization\_layer(int batch, int w, int h, int c, int size, float alpha, float beta, float kappa)

{

fprintf(stderr, "Local Response Normalization Layer: %d x %d x %d image, %d size\n", w,h,c,size);

layer layer = {0};

layer.type = NORMALIZATION;

layer.batch = batch;

layer.h = layer.out\_h = h;

layer.w = layer.out\_w = w;

layer.c = layer.out\_c = c;

layer.kappa = kappa;

layer.size = size;

layer.alpha = alpha;

layer.beta = beta;

layer.output = calloc(h \* w \* c \* batch, sizeof(float));

layer.delta = calloc(h \* w \* c \* batch, sizeof(float));

layer.squared = calloc(h \* w \* c \* batch, sizeof(float));

layer.norms = calloc(h \* w \* c \* batch, sizeof(float));

layer.inputs = w\*h\*c;

layer.outputs = layer.inputs;

layer.forward = forward\_normalization\_layer;

layer.backward = backward\_normalization\_layer;

#ifdef GPU

layer.forward\_gpu = forward\_normalization\_layer\_gpu;

layer.backward\_gpu = backward\_normalization\_layer\_gpu;

layer.output\_gpu = cuda\_make\_array(layer.output, h \* w \* c \* batch);

layer.delta\_gpu = cuda\_make\_array(layer.delta, h \* w \* c \* batch);

layer.squared\_gpu = cuda\_make\_array(layer.squared, h \* w \* c \* batch);

layer.norms\_gpu = cuda\_make\_array(layer.norms, h \* w \* c \* batch);

#endif

return layer;

}

## 高分辨率分类器

高分辨率分类器（High Resolution Classifier），目前表现很好的图片分类器均采用基于ImageNet数据集的预训练模型。大部分类器的输入图片尺寸都小于256px \* 256px。在YOLOv2中，首先采用448px \* 448px分辨率的ImageNet数据，然后微调，使网络模型适应高分辨率图片输入，最终高分辨率输入使模型Mean AP提升了四个点。

缩放卷积层大小，有了这个接口，网络就可以随意缩放卷积层的输入输出大小。

void resize\_convolutional\_layer(convolutional\_layer \*l, int w, int h)

{

l->w = w;

l->h = h;

int out\_w = convolutional\_out\_width(\*l);

int out\_h = convolutional\_out\_height(\*l);

l->out\_w = out\_w;

l->out\_h = out\_h;

l->outputs = l->out\_h \* l->out\_w \* l->out\_c;

l->inputs = l->w \* l->h \* l->c;

l->output = realloc(l->output, l->batch\*l->outputs\*sizeof(float));

l->delta = realloc(l->delta, l->batch\*l->outputs\*sizeof(float));

if(l->batch\_normalize){

l->x = realloc(l->x, l->batch\*l->outputs\*sizeof(float));

l->x\_norm = realloc(l->x\_norm, l->batch\*l->outputs\*sizeof(float));

}

#ifdef GPU

cuda\_free(l->delta\_gpu);

cuda\_free(l->output\_gpu);

l->delta\_gpu = cuda\_make\_array(l->delta, l->batch\*l->outputs);

l->output\_gpu = cuda\_make\_array(l->output, l->batch\*l->outputs);

if(l->batch\_normalize){

cuda\_free(l->x\_gpu);

cuda\_free(l->x\_norm\_gpu);

l->x\_gpu = cuda\_make\_array(l->output, l->batch\*l->outputs);

l->x\_norm\_gpu = cuda\_make\_array(l->output, l->batch\*l->outputs);

}

#ifdef CUDNN

cudnn\_convolutional\_setup(l);

#endif

#endif

l->workspace\_size = get\_workspace\_size(\*l);

}

## 加锚点框的卷积层

加锚点框的卷积层（Convolutional With 锚点框），YOLO采用全链接层来直接预测边界框，而Fast R-CNN采用人工选择的边界框。Fast R-CNN中的 region proposal network仅采用卷积层来预测固定的锚点框的偏移与置信度。

YOLOv2去除了全链接层，采用锚点框来预测边界框。首先，去除了一个池化层来提高卷积层输出分辨率。然后，修改网络输入尺寸：由448px \* 448px 改为416px \* 416 px，使特征图只有一个中心。大的物品更有可能出现在图片的中心。YOLO的卷积层下采样率为32，所以输入尺寸变为416px \* 416px,输出尺寸为13\*13。

采用锚点框，提升了精确度。YOLO每张图片预测九十八个预测框，但是采用锚点框，每张图片可以预测超过一千个预测框。采用锚点框方法后，Mean AP值下降了一点点，可以忽略不计，而recall（召回率）增加了六个点。

box encode\_box(box b, box anchor)

{

box encode;

encode.x = (b.x - anchor.x) / anchor.w;

encode.y = (b.y - anchor.y) / anchor.h;

encode.w = log2(b.w / anchor.w);

encode.h = log2(b.h / anchor.h);

return encode;

}

box decode\_box(box b, box anchor)

{

box decode;

decode.x = b.x \* anchor.w + anchor.x;

decode.y = b.y \* anchor.h + anchor.y;

decode.w = pow(2., b.w) \* anchor.w;

decode.h = pow(2., b.h) \* anchor.h;

return decode;

}

## 维度聚类

维度聚类（Dimension Clusters），在YOLO模型上采用锚点框有两个问题。第一，框的维度需要手动选择。模型可以简单的学习适当的调整预测框，但是假如人工事先选择先验框喂给网络模型，可以让模型更容易学习。 采用K-means聚类方法来自动选择最佳的先验框，这个过程可以通过一个python脚本，对训练集数据进行分析，最后得出5个锚点框作为先验框，手动选择的先验框可以有效的提高IOU分数。

这个操作是对数据集的训练前的一个评估预测，没有在网络框架中体现，它只需要生成5个可数据集内标框相近的anchor就行，保存在一个txt文件就完成任务了，网络框架可以直接读取这个文件，所有 另写了一个python脚本来实现这个功能。

def kmeans(X,centroids,eps,anchor\_file):

D=[]

old\_D = []

iterations = 0

diff = 1e5

c,dim = centroids.shape

while True:

iterations+=1

D = []

for i in range(X.shape[0]):

d = 1 - IOU(X[i],centroids)

D.append(d)

D = np.array(D)

if len(old\_D)>0:

diff = np.sum(np.abs(D-old\_D))

print 'diff = %f'%diff

if diff<eps or iterations>100:

print "Number of iterations took = %d"%(iterations)

print "Centroids = ",centroids

write\_anchors\_to\_file(centroids,X,anchor\_file)

return

#assign samples to centroids

belonging\_centroids = np.argmin(D,axis=1)

print belonging\_centroids

#calculate the new centroids

centroid\_sums=np.zeros((c,dim),np.float)

for i in range(belonging\_centroids.shape[0]):

centroid\_sums[belonging\_centroids[i]]+=X[i]

for j in range(c):

print '#annotations in centroid[%d] is %d'%(j,np.sum(belonging\_centroids==j))

centroids[j] = centroid\_sums[j]/np.sum(belonging\_centroids==j)

print 'new centroids = ',centroids

old\_D = D.copy()

print D

## 直接定位预测

直接定位预测（Direct Location Prediction），在YOLO模型上采用锚点框的原因就是网络模型不稳定，特别是在前几轮训练。大部分不稳定因素出自于预测框中心位置（x，y）偏差严重，移量范围限制在YOLO的格子的宽和高的范围，将预测值限定在格子框宽高的[0，1]比率范围内，增强稳定性。网络对特征矩阵中的每个格子预测五个边界框。对每一个边界框，模型预测5个匹配性值中心横纵坐标、框的宽高、置信度。采用聚类方法选择boxes维度与直接预测边界框中心位置将精确率提高将近5%。

def IOU(x,centroids):

dists = []

for centroid in centroids:

c\_w,c\_h = centroid

w,h = x

if c\_w>=w and c\_h>=h:

dist = w\*h/(c\_w\*c\_h)

elif c\_w>=w and c\_h<=h:

dist = w\*c\_h/(w\*h + (c\_w-w)\*c\_h)

elif c\_w<=w and c\_h>=h:

dist = c\_w\*h/(w\*h + c\_w\*(c\_h-h))

else: #means both w,h are bigger than c\_w and c\_h respectively

dist = (c\_w\*c\_h)/(w\*h)

dists.append(dist)

return np.array(dists)

def avg\_IOU(X,centroids):

n,d = X.shape

sum = 0.

for i in range(X.shape[0]):

#note IOU() will return array which contains IoU for each centroid and X[i] // slightly ineffective, but I am too lazy

sum+= max(IOU(X[i],centroids))

return sum/n

def write\_anchors\_to\_file(centroids,X,anchor\_file):

f = open(anchor\_file,'w')

#anchors = centroids\*416/32

anchors = centroids

#print 'Anchors = ', centroids\*416/32

num\_anchors = anchors.shape[0]

for i in range(num\_anchors):

anchors[i][0] = anchors[i][0]\*512/32

anchors[i][1] = anchors[i][1]\*200/32

for i in range(num\_anchors-1):

f.write('%f,%f, '%(anchors[i][0],anchors[i][1]))

#there should not be comma after last anchor, that's why

f.write('%f,%f\n'%(anchors[num\_anchors-1][0],anchors[num\_anchors-1][1]))

f.write('%f\n'%(avg\_IOU(X,centroids)))

## 细粒度特征

细粒度特征（Fine-Grained Features），改进后的YOLO对13\*13的格子进行目标检测。更精确的特征（finer grained features）可以有效的提高对于小目标的检测精度。向网络加入passtrough层以增加特征。Passthrough层类似于ResNet，将高分辨率特征与低分辨率特征结合，使26\*26\*512的特征图转化为13\*13\*2048的特征图。该改进使模型的性能增加了一个点。

这个只需要改参就行了，没有代码。

## 多尺度训练

多尺度训练（Multi-Scale Training），最初的YOLO输入尺寸为448px \* 448px，加入锚点框后，输入尺寸为416px \* 416px。模型只涵盖卷积层与pooling 层，两者输入输出都很灵活，所以可以方便的改变输入尺寸。

YOLO在训练时，每间隔一定批次就要改变模型输入的尺寸，以使模型对不同尺寸图片具有稳定的识别能力。每隔十批次，模型选择一种新的输入图片尺寸（320,352,...608，等），这些参数都是32的倍数，因为模型的下采样数量为32，缩放的大小是随机的，重新缩放输入的图片，继续训练。

该训练的规则能让模型自己去适应不同的输入图片的分辨率。模型对于小的尺寸的图片输入有更快的处理速度，但是精确度低，模型对于大的尺寸的图片输入有更慢的处理速度，但是精确度高，所以要想调节速度与精确率，YOLOv2可以按照任意的需求更改输入的分辨率。在低分辨率的情况下（288px \* 288px），YOLOv2可以与Fast R-CNN的精确率持平情况下，处理的速度可以达到九十帧每秒。

layer make\_normalization\_layer(int batch, int w, int h, int c, int size, float alpha, float beta, float kappa)

{

fprintf(stderr, "Local Response Normalization Layer: %d x %d x %d image, %d size\n", w,h,c,size);

layer layer = {0};

layer.type = NORMALIZATION;

layer.batch = batch;

layer.h = layer.out\_h = h;

layer.w = layer.out\_w = w;

layer.c = layer.out\_c = c;

layer.kappa = kappa;

layer.size = size;

layer.alpha = alpha;

layer.beta = beta;

layer.output = calloc(h \* w \* c \* batch, sizeof(float));

layer.delta = calloc(h \* w \* c \* batch, sizeof(float));

layer.squared = calloc(h \* w \* c \* batch, sizeof(float));

layer.norms = calloc(h \* w \* c \* batch, sizeof(float));

layer.inputs = w\*h\*c;

layer.outputs = layer.inputs;

layer.forward = forward\_normalization\_layer;

layer.backward = backward\_normalization\_layer;

#ifdef GPU

layer.forward\_gpu = forward\_normalization\_layer\_gpu;

layer.backward\_gpu = backward\_normalization\_layer\_gpu;

layer.output\_gpu = cuda\_make\_array(layer.output, h \* w \* c \* batch);

layer.delta\_gpu = cuda\_make\_array(layer.delta, h \* w \* c \* batch);

layer.squared\_gpu = cuda\_make\_array(layer.squared, h \* w \* c \* batch);

layer.norms\_gpu = cuda\_make\_array(layer.norms, h \* w \* c \* batch);

#endif

return layer;

}

## 交叉熵

用于模型训练的loss函数多种多样，比如，Leaky ReLU, ReLU, MSE, CE,其中交叉熵(Cross-Entropy，CE)作为loss有很多应用的场景，它最大的优点是训练时可以避免函数的梯度消失。

当分类器为二分类时:



当分类器为非二分器时：



其偏导函数是;



（详细推导过程请见附录）

均方差（Mean Squared Error，MSE）：是最基础的最简单的损失函数计算方法，但缺点是在多层网络中存在梯度消失的问题。

交叉熵（Cross-Entropy，CE）：用于度量两个概率分布之间的相似性。为了解决参数更新效率下降这一问题，代码中， 用交叉熵的损失函数替代了网络框架中的平方差的损失函数。如果预测值与实际值的误差越大，那么在反向传播的过程中权重调整的幅度就要越大，从而能够使训练加速收敛。

把均方差改成交叉熵：

float delta\_region\_box(box truth, float \*x, float \*biases, int n, int index, int i, int j, int w, int h, float \*delta, float scale)

{

box pred = get\_region\_box(x, biases, n, index, i, j, w, h);

float iou = box\_iou(pred, truth);

float tx = (truth.x\*w - i);

float ty = (truth.y\*h - j);

float tw = log(truth.w\*w / biases[2 \* n]);

float th = log(truth.h\*h / biases[2 \* n + 1]);

/\*delta[index + 0] = scale \* (tx - logistic\_activate(x[index + 0])) \* logistic\_gradient(logistic\_activate(x[index + 0]));//MSE(Mean Squared Error）

delta[index + 1] = scale \* (ty - logistic\_activate(x[index + 1])) \* logistic\_gradient(logistic\_activate(x[index + 1]));//MSE(Mean Squared Error）\*/

if (epoch >= 70000)scale = scale\*(2 - truth.w\*truth.h);

delta[index + 0] = scale \* (tx - logistic\_activate(x[index + 0]));// Cross-Entropy

delta[index + 1] = scale \* (ty - logistic\_activate(x[index + 1]));// Cross-Entropy

delta[index + 2] = scale \* (tw - x[index + 2]);

delta[index + 3] = scale \* (th - x[index + 3]);

return iou;

}

# 结果与结论

## 测试与运行效果

优化后的YOLOv2表现很好，在 自己的测试集上Mean AP可以跑到78.6%。

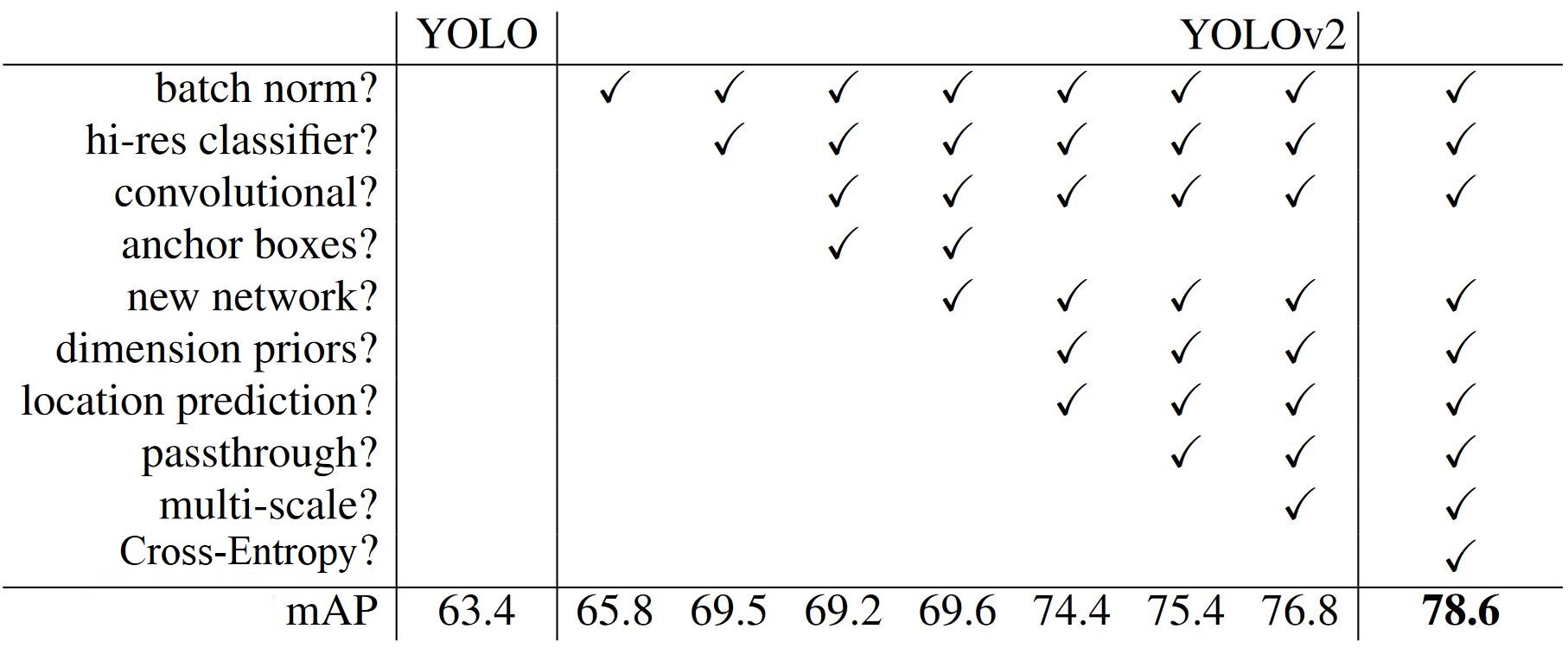


图6-1：YOLOv2加入优化的对比

YOLOv2相对于YOLO增加了批正则化、高分辨率分类、全链接层换成卷积层、用新网络消减计算量、维度聚类、本地特征预测、直通层和多图像尺寸训练，平均精度提升了13.4个点，在YOLOv2的基础上又加上交叉熵，平均精度又提升了1.8个点。

## 性能或者效果与同类算法的比较

|  |  |  |
| --- | --- | --- |
| Detectiom Framewoks | Mean AP | FPS |
| Fast R-CNN | 70.0 | 0.5 |
| R-CNN VGG-16 | 73.2 | 7 |
| Fast R-CNN ResNet | 76.4 | 5 |
| YOLO | 63.4 | 45 |
| SSD300 | 74.3 | 46 |
| SSD500 | 76.8 | 19 |
| YOLOv2 288\*288 | 69.0 | 91 |
| YOLOv2 352\*352 | 73.7 | 81 |
| YOLOv2 416\*416 | 76.8 | 67 |
| YOLOv2 480\*480 | 77.8 | 59 |
| YOLOv2 544\*544 | 78.6 | 40 |

图6-2：YOLOv2与其它优秀的模型对比

该对比实验采用的事单一变量法，用相同的训练数据、验证集与测试集训练出来的其它网络模型与优化后的YOLOv2进行了对比实验。每一组数据都存在一组与其仅差一个变量的数据与之对比，其中FPS是指模型每秒可以处理多少帧图片，mAP是指模型对测试集的准确预测的平均精度，Fast R-CNN,Fast R-CNN VGG-16, Fast R-CNN ResNet,YOLO,SSD,YOLOv2这些网络都是一些结构相似的神经网络框架，如表格6-2所示。

Fast R-CNN最差，Mean AP达到70%并且每两秒才能处理一帧也是就一张图片。Faster R-CNN VGG-16次之，Mean AP达到达到73.2%每秒可以处理七帧，提升了14倍。Faster R-CNN ResNet比Faster R-CNN VGG-16表现略好，但更慢一些。SSD500表现很出色，但是还是无法做到试试检测，因为人眼的效率在每秒二十四帧左右，到不了这个值，用户就会觉得卡顿，就会影响产品的品质。以上几种均不能达到 的期望，YOLO(YOLOv1)虽然可以很好地做到实时检测，但是它的Mean AP太差。那 优化过的YOLOv2就有很好的表现了，完全达到实时监测，并且Mean AP也比前几个模型中最好的还要高出1.8%，并且输入的图片分辨率越高模型获取到的特征越多，Mean AP跑分越高，但是带来的问题是效率会变慢，当输入图片到544px \* 544px的时候达到最优。

结束语

基于YOLOv2的左右手实时识别系统，在触控应用场景检测数据集中是最先进的，并且比其他目标检测系统更快。此外，它可以在各种图像尺寸下运行，对速度和准确性之间的做一个很好的折中。YOLOv2在深度学习领域中是个优秀的神经网络框架，像它多图像尺寸训练这样的训练技术可以为各种视觉任务提供很好的服务。

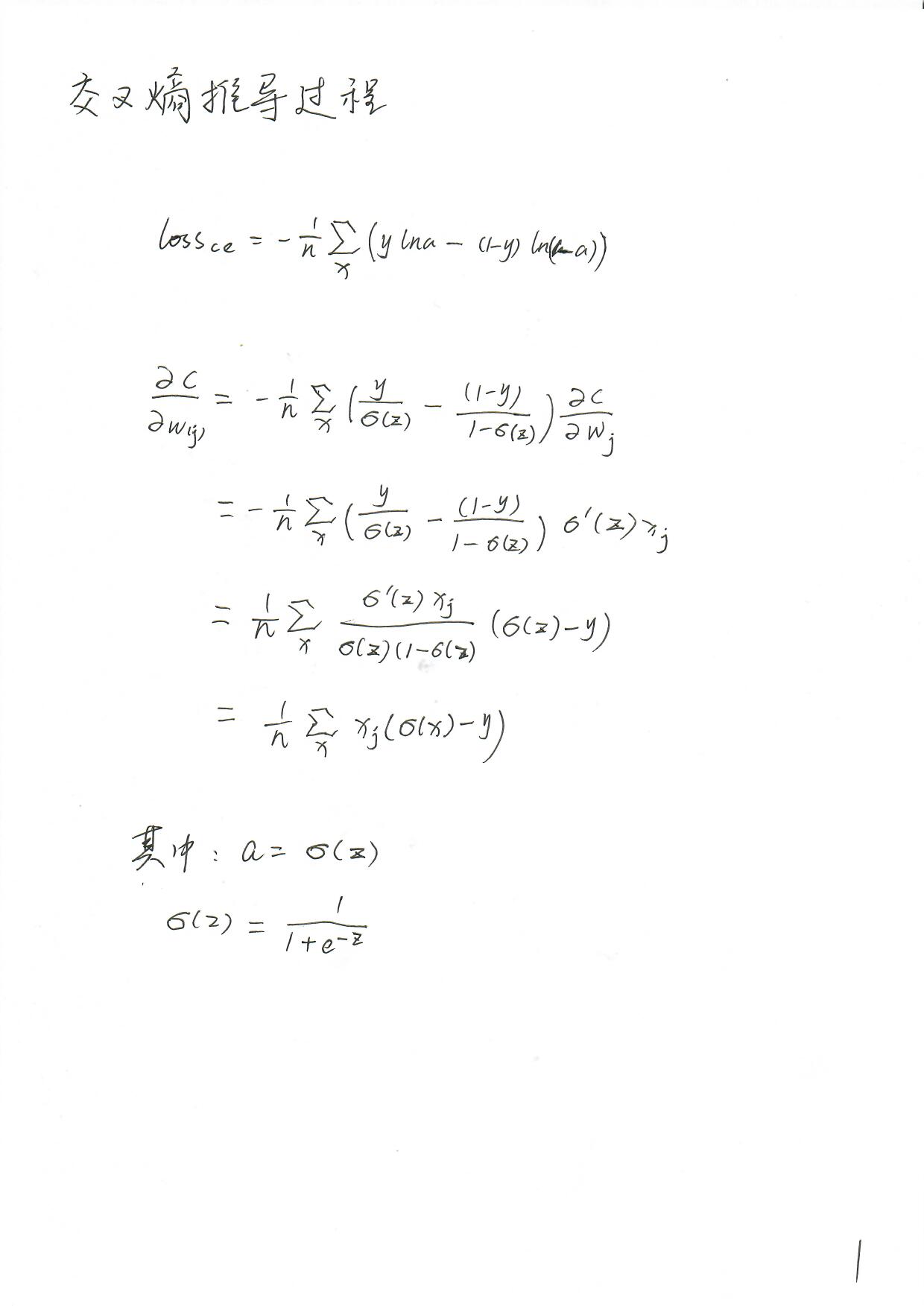
交叉熵的引进，对YOLOv2的框架的训练过程有一个极大的改进，有效地收敛模型，加速了模型训练速度。在同样的训练时间里，可以使模型进行更多的迭代，测试效果得以优化。

深度学习是机器学习里边最简单的一个小分支，它的泛化性很好，暂且可以把它当成一个黑盒来用，慢慢感受深度网络的“性格与能力”。深度学习的能力还是有限的，也需要大量的算力来实现它，花销很大，有必要继续学习更深一点的机器学习。机器学习的方向很多种，比如增强学习，深度学习等等，它需要高等数学作为基础，对概率论、微积分等要求很高。即使将来离开母校也要继续保持学习状态，对将来的技术之旅报以热忱的心态。

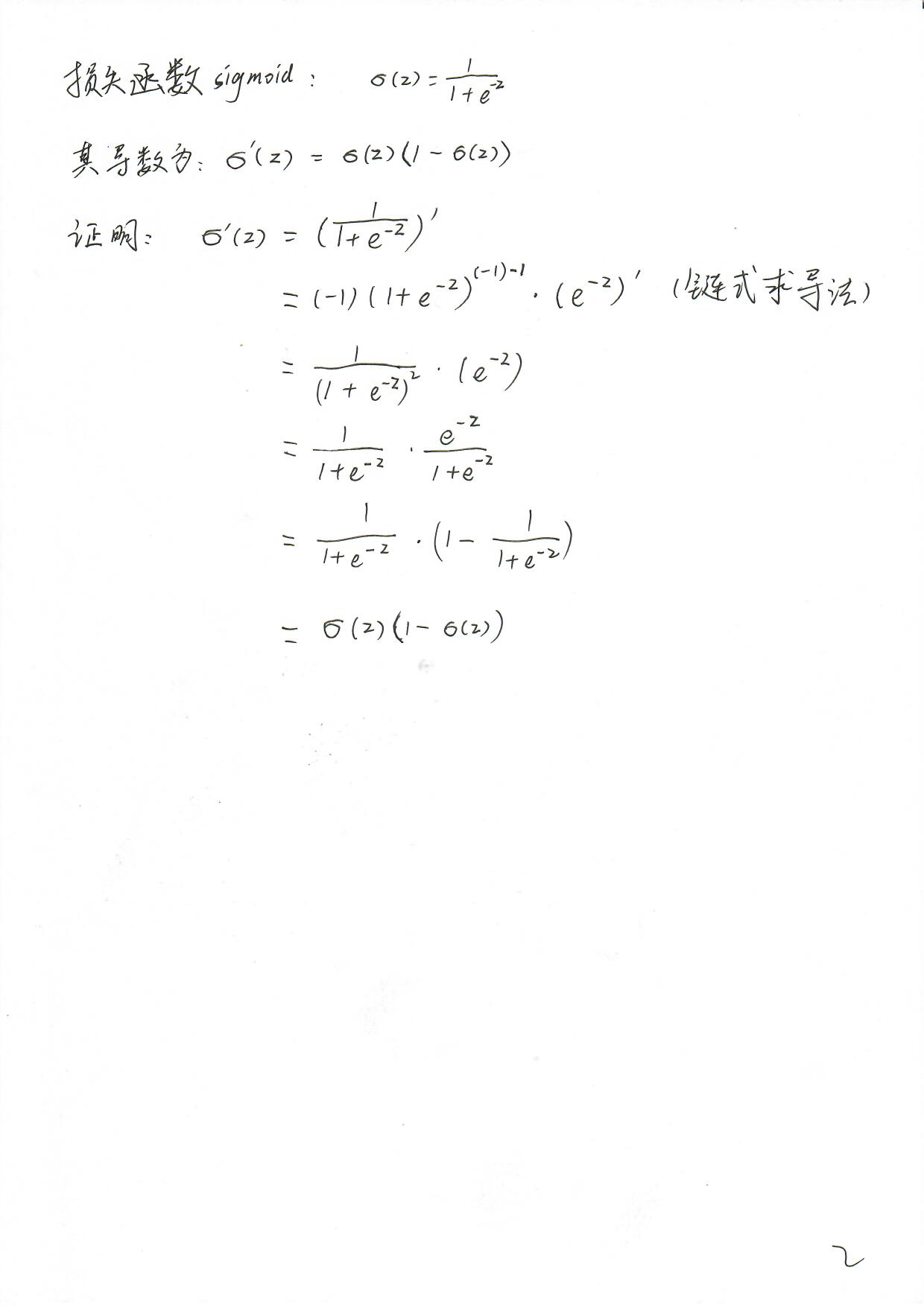
参考文献

1. Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi.You Only Look Once:Unified, Real-Time Object Detection[D].arXiv:1506.02640v5 ,2016,1～10
2. Joseph Redmon, Ali Farhadi.YOLO9000:Better, Faster, Stronger[D]. arXiv:1612.08242v1.2016,1～9
3. lan Goodfellow,Yoshua Bengio,Aaron Courville.DEEP LEARNING[M].第一版.人民邮电出版社.2017年8月.201～226
4. 周志华.机器学习[M].第一版.清华大学出版社.2016年1月.97～246
5. 李航.统计学习方法[M].第一版.清华大学出版社.2012年3月.1～53
6. J. Dong, Q. Chen, S. Yan, and A. Yuille. Towards unified object detection and semantic segmentation. In Computer Vision–ECCV 2014, 2014.7 299～314
7. K. Lenc and A. Vedaldi. R-cnn minus r.arXiv preprint arXiv:1506.06981, 2015. 5, 6
8. C. Szegedy, S. Ioffe, and V. Vanhoucke.Inception-v4,inception-resnet and the impact of residual connections on learning.CoRR, abs/1602.07261, 2016. 2
9. 谢剑斌等.视觉机器学习20讲[M].第一版.清华大学出版社.2015年6月.59～164
10. A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems,2012. 2 1097～1105
11. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh,S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein,A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge.International Journal of Computer Vision (IJCV), 2015. 3
12. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition[D].arXiv preprint arXiv:1409.1556, 2014. 2～5
13. B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni,D.Poland, D.Borth, and L.-J. Li. Yfcc100m: The new data in multimedia research[D]. Communications of the ACM,59(2):64–73, 2016.1
14. S. Bell, C.L. Zitnick, K. Bala, and R. Girshick. Inside-outside net:Detecting objects in context with skip pooling and recurrent neural networks[D]. arXiv preprint arXiv:1512.04143, 2015.1～6
15. S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift[D]. arXiv preprint arXiv:1502.03167, 2015.2～5
16. K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition[D]. arXiv preprint arXiv:1512.03385,2015. 2～5
17. K. Lenc and A. Vedaldi. R-cnn minusr[D].arXiv preprint arXiv:1506.06981, 2015. 5～6
18. S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks[D].arXiv preprint arXiv:1506.01497, 2015. 5～7
19. H. Cai, Q. Wu, T. Corradi, and P. Hall.The cross-depiction problem: Computer vision algorithms for recognising objects in artwork and in photographs[D]. arXiv preprintarXiv:1505.00110, 2015. 7
20. J. Yan, Z. Lei, L. Wen, and S. Z. Li. The fastest deformable part model for object detection[D]. In Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, pages2497–2504. IEEE, 2014. 5～6
21. J. Redmon and A. Angelova. Real-time grasp detection using convolutional neural networks[D].CoRR, abs/1412.3128, 2014.5

附录



参数w的梯度推导过程，其中，x表示样本，n表示样本的总数。



Sigmoid函数的导函数推导过程

致谢

首先要感谢的是我的论文指导老师，南阳理工学院软件学院的靳冰老师，靳冰老师在论文的写作中投入了很多精力和心血，对我的论文研究方向做了指导性的推荐和建议，及时的对我遇到的困惑予以悉心的点播，并且提出了许多关键的改善型建议。在这里，我对靳冰老师对我的关怀和鼓励表示诚挚的感谢。

同时还要感谢南阳理工学院所有软件工程授课的老师们，您将软件工程的课程深入浅出的传授给我，夯实了我的软件工程基础。感谢软件学院的所有同学们，大家在一起互相督促共同进步，共同探讨毕业设计的困难与解决方案，共同度过了一个充实的大学生活。

此外，我还要感谢上海品奇数码科技有限公司的CEO陆飞的大力支持。陆飞是我实习公司的老板，更是一位技术大牛，非常支持我学习深度学习，这让我有机会接触到了YOLO,有了陆飞老板的帮助我对人工智能行业有了深入的了解，有机会做梦想中的工作！

另外，我要感谢我的父母，是他们不辞辛苦，全力以赴地支持我上大学，有了父母的支持与鼓励，我才得以顺利地完成论文的撰写。

最后，感谢南阳理工学院的支持与创新，使得我们学生可以在异地答辩，这是一个壮举，谢谢你们谢谢论文评阅老师们的辛苦工作。