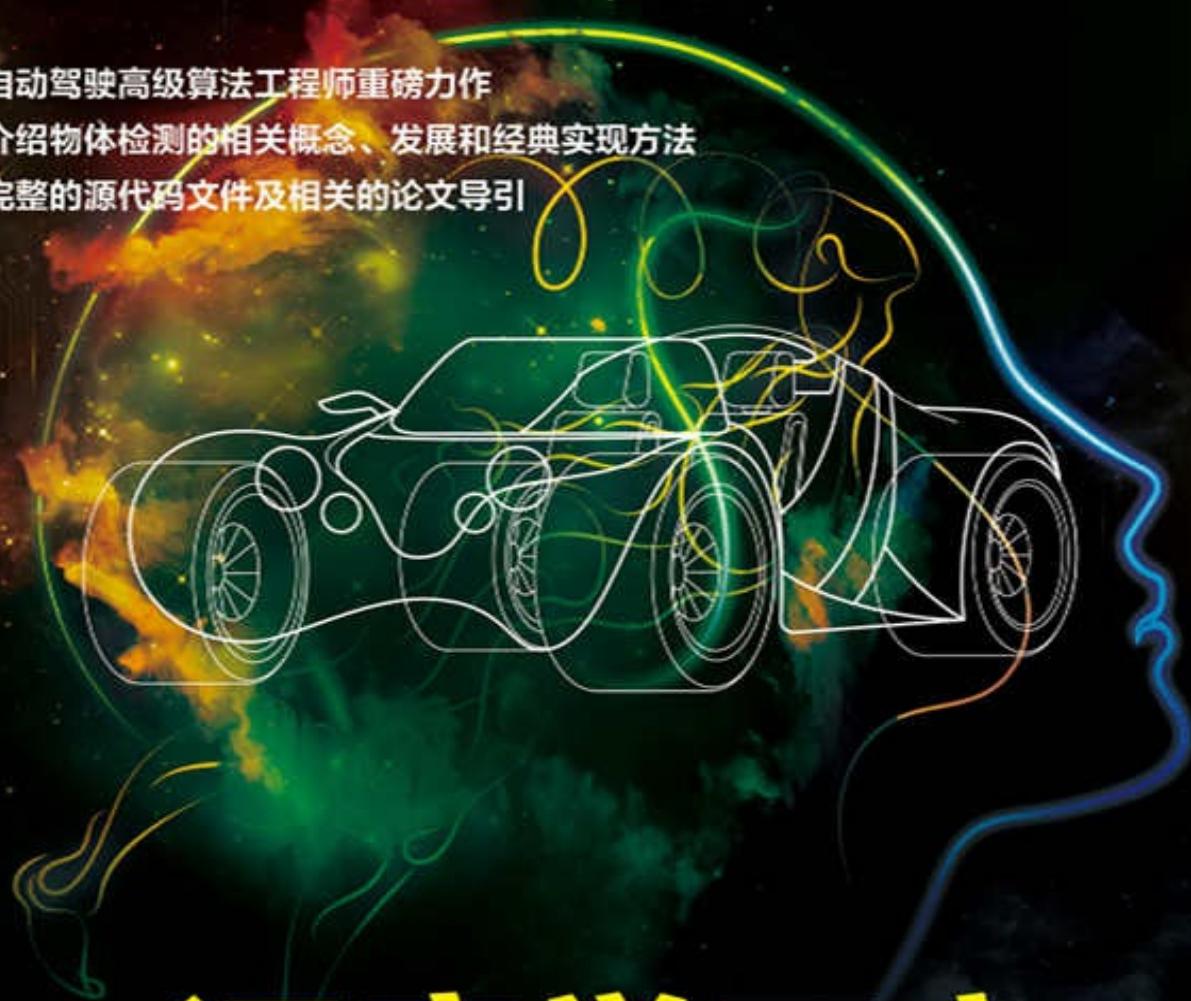


董洪义〇编著

百度自动驾驶高级算法工程师重磅力作  
系统介绍物体检测的相关概念、发展和经典实现方法  
提供完整的源代码文件及相关的论文导引



# 深度学习之 PyTorch物体检测实战

Object Detection by Deep Learning: Core Technologies and Practices

用PyTorch从代码角度详解Faster RCNN、SSD和YOLO三个经典检测器

详细介绍物体检测的轻量化网络、细节处理、难点问题及未来的发展趋势

王田苗  
教育部长江学者特聘教授

陶吉  
百度自动驾驶技术总监

夏添  
百度自动驾驶前事业部主任架构师

武伟  
商汤科技研发总监

丁宁  
义柏资本技术专家

朱本金  
旷视研究院研究员

唐家声  
阿里巴巴达摩院视觉工程师

共同推荐



机械工业出版社  
China Machine Press

# 深度学习之PyTorch物体检测实战

董洪义 编著

ISBN：978-7-111-64174-2

本书纸版由机械工业出版社于2019年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

# 目录

前言

第1篇 物体检测基础知识

    第1章 浅谈物体检测与PyTorch

        1.1 深度学习与计算机视觉

            1.1.1 发展历史

            1.1.2 计算机视觉

        1.2 物体检测技术

            1.2.1 发展历程

            1.2.2 技术应用领域

            1.2.3 评价指标

        1.3 PyTorch简介

            1.3.1 诞生与特点

            1.3.2 各大深度学习框架对比

            1.3.3 为什么选择PyTorch

            1.3.4 安装方法

        1.4 基础知识准备

            1.4.1 Linux基础

            1.4.2 Python基础

            1.4.3 高效开发工具

        1.5 总结

第2章 PyTorch基础

    2.1 基本数据：Tensor

        2.1.1 Tensor数据类型

        2.1.2 Tensor的创建与维度查看

        2.1.3 Tensor的组合与分块

        2.1.4 Tensor的索引与变形

        2.1.5 Tensor的排序与取极值

2.1.6 Tensor的自动广播机制与向量化

2.1.7 Tensor的内存共享

## 2.2 Autograd与计算图

2.2.1 Tensor的自动求导: Autograd

2.2.2 计算图

2.2.3 Autograd注意事项

## 2.3 神经网络工具箱torch.nn

2.3.1 nn.Module类

2.3.2 损失函数

2.3.3 优化器nn.optim

## 2.4 模型处理

2.4.1 网络模型库: torchvision.models

2.4.2 加载预训练模型

2.4.3 模型保存

## 2.5 数据处理

2.5.1 主流公开数据集

2.5.2 数据加载

2.5.3 GPU加速

2.5.4 数据可视化

## 2.6 总结

# 第3章 网络骨架: Backbone

## 3.1 神经网络基本组成

3.1.1 卷积层

3.1.2 激活函数层

3.1.3 池化层

3.1.4 Dropout层

3.1.5 BN层

3.1.6 全连接层

- 3.1.7 深入理解感受野
- 3.1.8 详解空洞卷积（Dilated Convolution）
- 3.2 走向深度：VGGNet
- 3.3 纵横交错：Inception
- 3.4 里程碑：ResNet
- 3.5 继往开来：DenseNet
- 3.6 特征金字塔：FPN
- 3.7 为检测而生：DetNet
- 3.8 总结

## 第2篇 物体检测经典框架

- 第4章 两阶经典检测器：Faster RCNN
  - 4.1 RCNN系列发展历程
    - 4.1.1 开山之作：RCNN
    - 4.1.2 端到端：Fast RCNN
    - 4.1.3 走向实时：Faster RCNN
  - 4.2 准备工作
  - 4.3 Faster RCNN总览
  - 4.4 详解RPN
    - 4.4.1 理解Anchor
    - 4.4.2 RPN的真值与预测量
    - 4.4.3 RPN卷积网络
    - 4.4.4 RPN真值的求取
    - 4.4.5 损失函数设计
    - 4.4.6 NMS与生成Proposal
    - 4.4.7 筛选Proposal得到RoI
  - 4.5 RoI Pooling层
  - 4.6 全连接RCNN模块
    - 4.6.1 RCNN全连接网络

#### 4.6.2 损失函数设计

### 4.7 Faster RCNN的改进算法

#### 4.7.1 审视Faster RCNN

#### 4.7.2 特征融合: HyperNet

#### 4.7.3 实例分割: Mask RCNN

#### 4.7.4 全卷积网络: R-FCN

#### 4.7.5 级联网络: Cascade RCNN

### 4.8 总结

## 第5章 单阶多层检测器: SSD

### 5.1 SSD总览

#### 5.1.1 SSD的算法流程

#### 5.1.2 代码准备工作

### 5.2 数据预处理

#### 5.2.1 加载PASCAL数据集

#### 5.2.2 数据增强

### 5.3 网络架构

#### 5.3.1 基础VGG结构

#### 5.3.2 深度卷积层

#### 5.3.3 PriorBox与边框特征提取网络

#### 5.3.4 总体网络计算过程

### 5.4 匹配与损失求解

#### 5.4.1 预选框与真实框的匹配

#### 5.4.2 定位损失的计算

#### 5.4.3 难样本挖掘

#### 5.4.4 类别损失计算

### 5.5 SSD的改进算法

#### 5.5.1 审视SSD

#### 5.5.2 特征融合: DSSD

- 5.5.3 彩虹网络：RSSD
- 5.5.4 基于SSD的两阶：RefineDet
- 5.5.5 多感受野融合：RFBNet

## 5.6 总结

# 第6章 单阶段经典检测器：YOLO

- 6.1 无锚框预测：YOLO v1
  - 6.1.1 网络结构
  - 6.1.2 特征图的意义
  - 6.1.3 损失计算
- 6.2 依赖锚框：YOLO v2
  - 6.2.1 网络结构的改善
  - 6.2.2 先验框的设计
  - 6.2.3 正、负样本与损失函数
  - 6.2.4 正、负样本选取代码示例
  - 6.2.5 工程技巧
- 6.3 多尺度与特征融合：YOLO v3
  - 6.3.1 新网络结构DarkNet-53
  - 6.3.2 多尺度预测
  - 6.3.3 Softmax改为Logistic

## 6.4 总结

# 第3篇 物体检测的难点与发展

# 第7章 模型加速之轻量化网络

- 7.1 压缩再扩展：SqueezeNet
  - 7.1.1 SqueezeNet网络结构
  - 7.1.2 SqueezeNet总结
- 7.2 深度可分离：MobileNet
  - 7.2.1 标准卷积
  - 7.2.2 深度可分离卷积

7.2.3 MobileNet v1结构

7.2.4 MobileNet v1总结

7.2.5 MobileNet v2

### 7.3 通道混洗: ShuffleNet

7.3.1 通道混洗

7.3.2 网络结构

7.3.3 ShuffleNet v2

### 7.4 总结

## 第8章 物体检测细节处理

### 8.1 非极大值抑制: NMS

8.1.1 NMS基本过程

8.1.2 抑制得分: Soft NMS

8.1.3 加权平均: Softer NMS

8.1.4 定位置信度: IoU-Net

### 8.2 样本不均衡问题

8.2.1 不均衡问题分析

8.2.2 在线难样本挖掘: OHEM

8.2.3 专注难样本: Focal Loss

### 8.3 模型过拟合

8.3.1 数据增强

8.3.2 L1与L2正则化

### 8.4 总结

## 第9章 物体检测难点

### 9.1 多尺度检测

9.1.1 多尺度问题

9.1.2 降低下采样率与空洞卷积

9.1.3 Anchor设计

9.1.4 多尺度训练

9.1.5 特征融合

9.1.6 尺度归一化: SNIP

9.1.7 三叉戟: TridentNet

## 9.2 拥挤与遮挡

9.2.1 遮挡背景

9.2.2 排斥损失: Repulsion Loss

9.2.3 OR-CNN

## 9.3 总结

# 第10章 物体检测的未来发展

## 10.1 重新思考物体检测

10.1.1 精度与速度的权衡

10.1.2 卷积网络的可解释性与稳定性

10.1.3 训练: 微调还是随机初始化

10.1.4 考虑物体间关系的检测

10.1.5 优化卷积方式

10.1.6 神经架构搜索: NAS

10.1.7 与产业结合的创新

## 10.2 摆脱锚框: Anchor-Free

10.2.1 重新思考Anchor

10.2.2 基于角点的检测: CornerNet

10.2.3 检测中心点: CenterNet

10.2.4 锚框自学习: Guided Anchoring

## 10.3 总结

# 前言

随着深度学习的飞速发展，计算机视觉技术取得了令人瞩目的成果，尤其是物体检测这一基础又核心的分支，诞生了众多经典算法，在自动驾驶、智能医疗、智能安防及搜索娱乐等多个领域都得到了广泛应用。与此同时，诞生于2017年的PyTorch框架，凭借其简洁优雅、灵活易上手等优点，给开发人员留下了深刻的印象。

目前，国内图书市场上已经出版了几本PyTorch方面的图书，但大多数图书停留在浅层的概念与简单示例的讲解上，缺乏实用性，而且也没有一本系统讲解PyTorch物体检测方面的图书。因此，图书市场上迫切需要一本系统介绍PyTorch物体检测技术的书籍。这便是笔者写作本书的初衷。

本书是国内原创图书市场上首部系统介绍物体检测技术的图书。书中利用PyTorch深度学习框架，从代码层面讲解了Faster RCNN、SSD及YOLO这三大经典框架的相关知识，并进一步介绍了物体检测的细节与难点问题，让读者可以全面、深入、透彻地理解物体检测的种种细节，并能真正提升实战能力，从而将这些技术灵活地应用到实际开发中，享受深度学习带来的快乐。

## 本书特色

### 1. 系统介绍了PyTorch物体检测技术

本书深入物体检测这一基础又核心的技术，从其诞生背景、主流算法、难点问题、发展趋势等多个角度详细介绍了物体检测知识，并结合代码给出了多个算法的实现。

### 2. 从代码角度详细介绍了物体检测的三大算法

本书介绍了Faster RCNN、SSD及YOLO这三个影响深远的检测算法，从代码层面详细介绍了它们所实现的每一个细节与难点，并进行了优缺点分析，而且给出了多种优化算法。

### 3.涵盖所有主流的物体检测算法

本书几乎涵盖所有主流的物体检测算法，包括VGGNet、ResNet、FPN、DenseNet和DetNet等卷积基础网络，以及从Faster RCNN、HyperNet、Mask RCNN、SSD、RefineDet、YOLO v1到YOLO v3、RetinaNet、CornerNet和CenterNet等物体检测算法，呈现给读者一个完整的知识体系。

### 4.给出了多个实际的物体检测实例，有很强的实用性

本书对PyTorch的知识体系进行了较为精炼的介绍，还结合物体检测算法重点介绍了PyTorch实现的多个物体检测实例。因此本书不仅是一本很好的PyTorch框架学习书籍，更是一本PyTorch物体检测实战宝典。

### 5.对物体检测技术常见的细节、难点及发展做了详细分析

本书不仅对物体检测技术的热门话题做了详细分析，例如非极大值抑制、样本不均衡、模型过拟合、多尺度检测、物体拥挤与遮挡等，而且对各种细节与常见问题做了详细分析，并给出了多种解决方法。

## 本书内容

### 第1篇 物体检测基础知识

本篇涵盖第1~3章，介绍了物体检测技术与PyTorch框架的背景知识与必备的基础知识。主要内容包括物体检测技术的背景与发展；物体检

测的多种有效工具；PyTorch背景知识与基础知识；多种基础卷积神经网络的相关知识与具体实现等。掌握本篇内容，可以为读者进一步学习物体检测技术奠定基础。

## 第2篇 物体检测经典框架

本篇涵盖第4~6章，介绍了Faster RCNN、SSD与YOLO三大经典算法的思想与实现。主要内容包括Faster RCNN两阶算法的思想；锚框Anchor的意义与实现；Faster RCNN的多种改进算法；SSD单阶算法的思想与实现；SSD的数据增强方法及多种改进算法；YOLO单阶算法的三个版本演变过程及具体实现等。掌握本篇内容，可以让读者从代码角度学习物体检测的种种细节。

## 第3篇 物体检测的难点与发展

本篇涵盖第7~10章，介绍了物体检测技术的细节、难点及未来发展。主要内容包括针对模型加速的多种轻量化网络思想与实现；非极大值抑制；样本不均衡及模型过拟合等物体检测细节问题的背景知识与解决方法；多尺度、拥挤与遮挡等物体检测难点问题的背景知识与解决方法；多种摆脱锚框的检测算法；物体检测的未来发展趋势等。掌握本篇内容，可以让读者更加深入地学习物体检测的相关技术。

## 本书读者对象

- 需要全面学习物体检测技术的人员；
- PyTorch框架爱好者和研究者；
- 计算机视觉从业人员与研究者；
- 深度学习从业人员与爱好者；

- 自动驾驶、智能安防等领域的开发人员；
- 人工智能相关产业的从业人员；
- 计算机、机器人等专业的高校学生。

## 阅读建议

- 没有物体检测与PyTorch基础的读者，建议从第1章顺次阅读并演练每一个实例。
- 有一定PyTorch与物体检测基础的读者，可以根据实际情况有重点地选择阅读各个算法的细节。
- 对于每一个检测算法，建议读者先阅读一下原论文，多思考算法设计的动机与目的，并重点思考如何用代码实现，这会加深读者对检测算法的理解。原论文的下载地址和本书源代码文件一起提供。
- 多思考各种物体检测算法的优缺点、相互之间的联系与区别，以及可以优化和改进的细节等，形成完整的知识体系树，这样会进一步加深读者对知识的理解。

## 配书资源获取方式

本书涉及的全部源代码都放在了GitHub上，需要读者自行下载。下载地址如下：

<https://github.com/dongdonghy/Detection-PyTorch-Notebook>

有些章节的代码较多，但在书中仅给出了重要的片段代码，完整代码以GitHub上的代码为准。

另外，读者也可以登录华章公司的网站[www.hzbook.com](http://www.hzbook.com)，搜索到本书，然后单击“资料下载”按钮，即可在本书页面上找到相关的下载链接。

## 致谢

本书的编写得到了许多人的帮助。可以说，本书是多人共同努力的结晶。感谢北京源智天下科技有限公司的王蕾，她在稿件整理方面帮做了大量的工作！感谢王田苗教授、陶吉博士、夏添博士、侯涛刚博士、严德培、单增光、王策、鄂俊光、李成、丁宁、付航、高鹏、朱本金、彭强、王粟瑶、张腾、王兆玮、黄彬效和拓万琛等人，他们对本书提出了许多宝贵的意见和建议！感谢我的女朋友及家人，他们一直以来都对我鼓励有加，给我写作本书以最大的动力！感谢为本书付出辛勤工作的每一位编辑，他们认真、细致的工作让本书质量提高不少！

由于本书涉及的知识点较多，难免有错漏与不当之处，敬请各位读者指正。如有疑问，请随时通过电子邮件与笔者联系，笔者将不胜感激。联系邮箱：[hzbook2017@163.com](mailto:hzbook2017@163.com)。

董洪义

# 第1篇 物体检测基础知识

·第1章 浅谈物体检测与PyTorch

·第2章 PyTorch基础

·第3章 网络骨架：Backbone

## 第1章 浅谈物体检测与PyTorch

在2012年的ImageNet图像识别竞赛中，Hinton率领的团队利用卷积神经网络构建的AlexNet一举夺得了冠军，从此点燃了学术界、工业界等对于深度学习、卷积网络的热情。短短六七年的时间，在图像分类、物体检测、图像分割、人体姿态估计等一系列计算机视觉领域，深度学习都收获了极大的成功。

作为本书的开篇，为了使读者有一个全面的认知，本章将专注于介绍物体检测的基础知识、发展历史，以及为什么选择PyTorch作为深度学习的框架。

## 1.1 深度学习与计算机视觉

近年来，人工智能、机器学习、深度学习、物体检测这样的名词经常会出现在大家的眼前，但大家对这些技术之间的区别与联系却经常混淆。基于此，本节将会介绍每一个技术的基本概念与发展历史。

### 1.1.1 发展历史

当前的计算机无论是处理多复杂的计算任务，其基本逻辑始终是0与1的位运算，凭借其每秒亿万的计算次数，计算机可以快速完成复杂的数学运算，这是人类无法比拟的。然而，涉及高语义的理解、判断，甚至是情感、意识（如判断一张图像中有几个儿童）这种对于人类轻而易举的任务，对计算机而言却很难处理，因为这种问题无法建立明确的数学规则。

为了赋予计算机以人类的理解能力与逻辑思维，诞生了人工智能（Artificial Intelligence, AI）这一学科。在实现人工智能的众多算法中，机器学习是发展较为快速的一支。机器学习的思想是让机器自动地从大量的数据中学习出规律，并利用该规律对未知的数据做出预测。在机器学习的算法中，深度学习是特指利用深度神经网络的结构完成训练和预测的算法。

人工智能、机器学习、深度学习这三者的关系如图1.1所示。可以看出，机器学习是实现人工智能的途径之一，而深度学习则是机器学习的算法之一。如果把人工智能比喻成人类的大脑，机器学习则是人类通过大量数据来认知学习的过程，而深度学习则是学习过程中非常高效的一种算法。

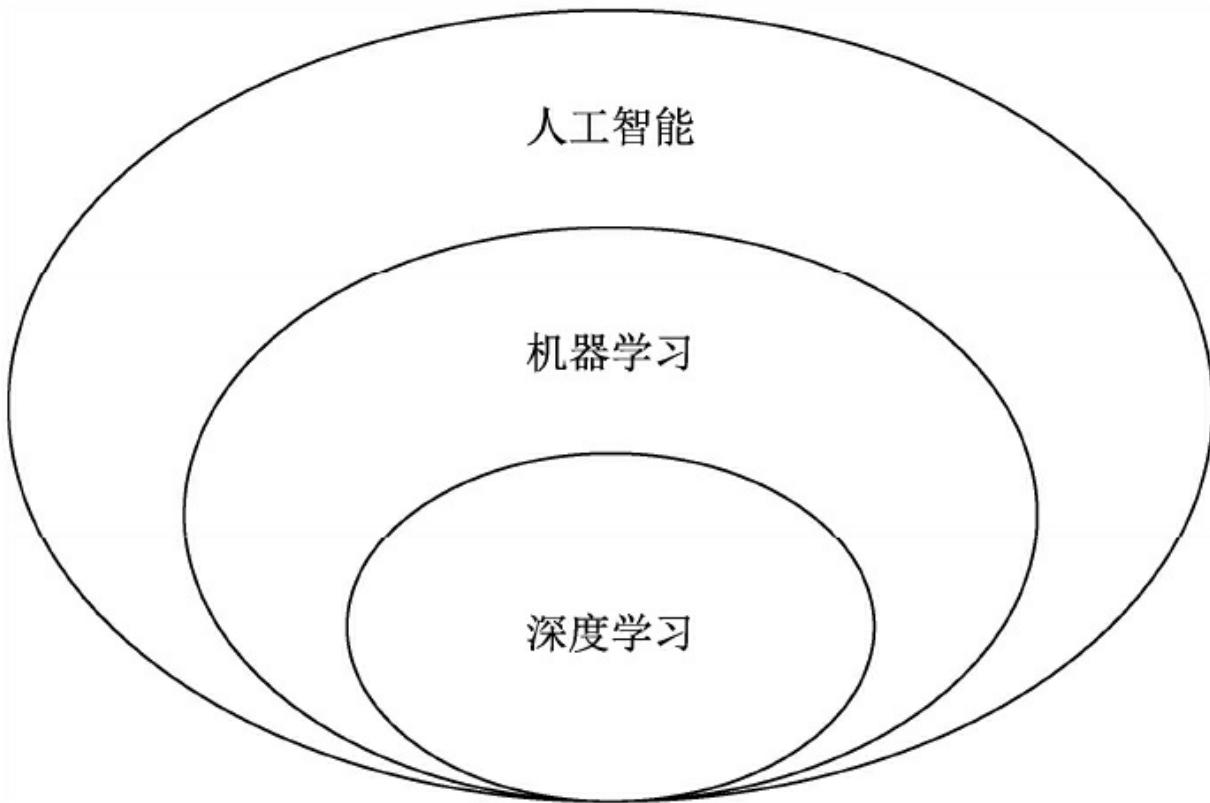


图1.1 人工智能、机器学习与深度学习三者间的关系

下面详细介绍这三者的发展历程。

## 1. 人工智能

人工智能的概念最早来自于1956年的计算机达特茅斯会议，其本质是希望机器能够像人类的大脑一样思考，并作出反应。由于极具难度与吸引力，人工智能从诞生至今，吸引了无数的科学家与爱好者投入研究。搭载人工智能的载体可以是近年来火热的机器人、自动驾驶车辆，甚至是一个部署在云端的智能大脑。

根据人工智能实现的水平，我们可以进一步分为3种人工智能，如图1.2所示。

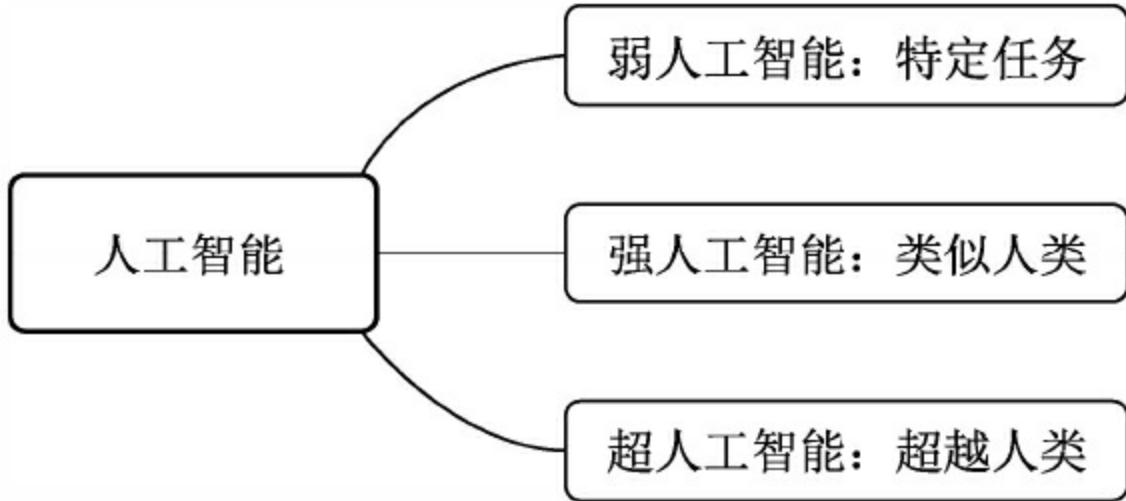


图1.2 人工智能分类

弱人工智能（Artificial Narrow Intelligence, ANI）：擅长某个特定任务的智能。例如语言处理领域的谷歌翻译，让该系统去判断一张图片中是猫还是狗，就无能无力了。再比如垃圾邮件的自动分类、自动驾驶车辆、手机上的人脸识别等，当前的人工智能大多是弱人工智能。

强人工智能：在人工智能概念诞生之初，人们期望能够通过打造复杂的计算机，实现与人一样的复杂智能，这被称做强人工智能，也可以称之为通用人工智能（Artificial General Intelligence, AGI）。这种智能要求机器像人一样，听、说、读、写样样精通。目前的发展技术尚未达到通用人工智能的水平，但已经有众多研究机构展开了研究。

超人工智能（Artificial Super Intelligence, ASI）：在强人工智能之上，是超人工智能，其定义是在几乎所有领域都比人类大脑聪明的智能，包括创新、社交、思维等。人工智能科学家Aaron Saenz曾有一个有趣的比喻，现在的弱人工智能就好比地球早期的氨基酸，可能突然之间就会产生生命。超人工智能不会永远停留在想象之中。

人类追求人工智能的脚步永不会停止，例如鼎鼎有名的智能围棋学习系统AlphaGo，由谷歌大脑团队开发，在2016年3月15日以4：1的总比

分战胜了韩国围棋选手李世石，引发了全世界巨大的反响。谷歌在时隔一年后推出了AlphaGo Zero，可以在不使用人类棋谱经验的前提下成为一个围棋高手，具有划时代的意义。我们也有理由相信，实现更高级的人工智能指日可待。

## 2. 机器学习

机器学习是实现人工智能的重要途径，也是最早发展起来的人工智能算法。与传统的基于规则设计的算法不同，机器学习的关键在于从大量的数据中找出规律，自动地学习出算法所需的参数。

机器学习最早可见于1783年的贝叶斯定理中。贝叶斯定理是机器学习的一种，根据类似事件的历史数据得出发生的可能性。在1997年，IBM开发的深蓝（Deep Blue）象棋电脑程序击败了世界冠军。当然，最令人振奋的成就还当属2016年打败李世石的AlphaGo。

机器学习算法中最重要的就是数据，根据使用的数据形式，可以分为三大类：监督学习（Supervised Learning）、无监督学习（Unsupervised Learning）与强化学习（Reinforcement Learning），如图1.3所示。

·**监督学习：**通常包括训练与预测阶段。在训练时利用带有人工标注标签的数据对模型进行训练，在预测时则根据训练好的模型对输入进行预测。监督学习是相对成熟的机器学习算法。监督学习通常分为分类与回归两个问题，常见算法有决策树（Decision Tree, DT）、支持向量机（Support Vector Machine, SVM）和神经网络等。

·**无监督学习：**输入的数据没有标签信息，也就无法对模型进行明确的惩罚。无监督学习常见的思路是采用某种形式的回报来激励模型做出一定的决策，常见的算法有K-Means聚类与主成分分析（Principal

Component Analysis, PCA)。

·强化学习：让模型在一定的环境中学习，每次行动会有对应的奖励，目标是使奖励最大化，被认为是走向通用人工智能的学习方法。常见的强化学习有基于价值、策略与模型3种方法。

机器学习涉及概率、统计、凸优化等多个学科，目前已在数据挖掘、计算机视觉、自然语言处理、医学诊断等多个领域有了相当成熟的应用。

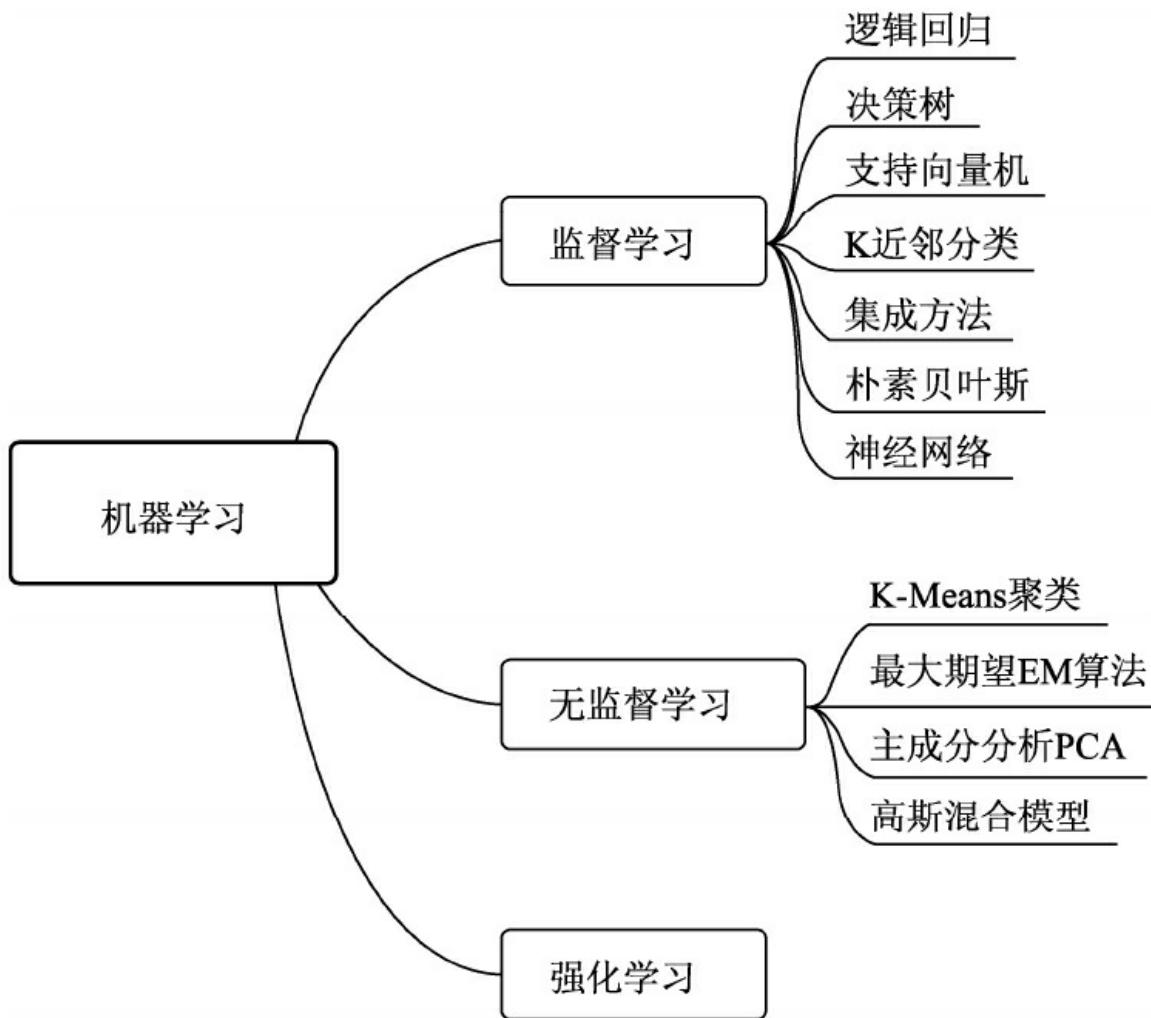


图1.3 机器学习算法总览

### 3. 深度学习

深度学习是机器学习的技术分支之一，主要是通过搭建深层的人工神经网络（Artificial Neural Network）来进行知识的学习，输入数据通常较为复杂、规模大、维度高。深度学习可以说是机器学习问世以来最大的突破之一。

深度学习的发展经历了一番波折，具体历程如图1.4所示。

最早的神经网络可以追溯到1943年的MCP（McCulloch and Pitts）人工神经元网络，希望使用简单的加权求和与激活函数来模拟人类的神经元过程。在此基础上，1958年的感知器（Perception）模型使用了梯度下降算法来学习多维的训练数据，成功地实现了二分类问题，也掀起了深度学习的第一次热潮。图1.5代表了一个最简单的单层感知器，输入有3个量，通过简单的权重相加，再作用于一个激活函数，最后得到了输出y。

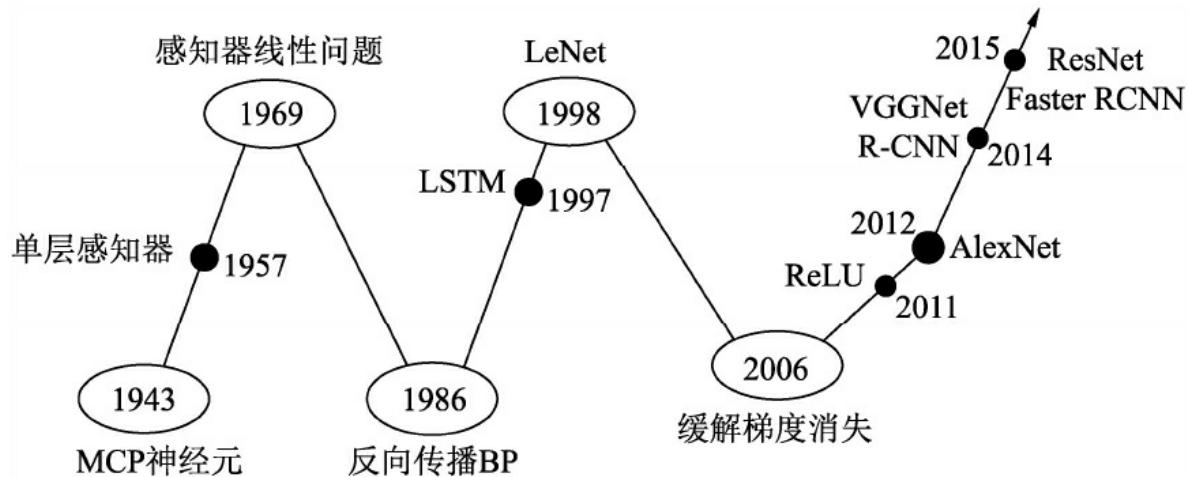


图1.4 深度学习发展历程

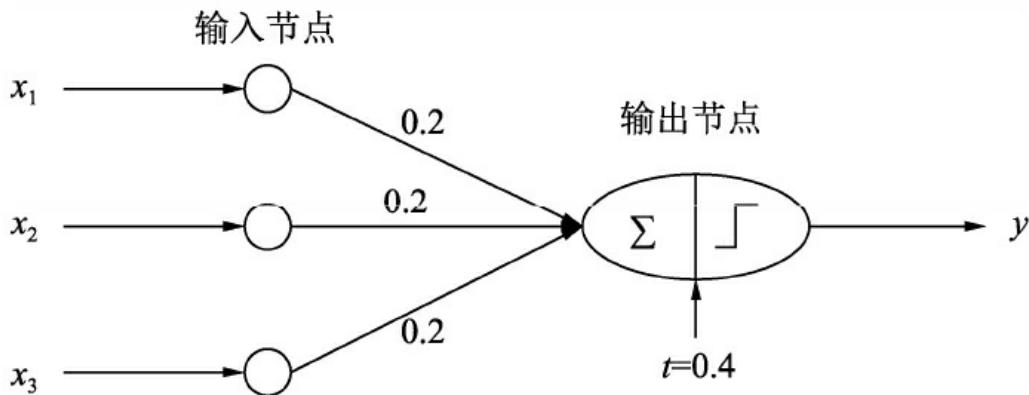


图1.5 单层感知器

然而，1969年，Minsky证明了感知器仅仅是一种线性模型，对简单的与或判断都无能为力，而生活中的大部分问题都是非线性的，这直接让学者研究神经网络的热情难以持续，造成了深度学习长达20年的停滞不前。

1986年，深度学习领域“三驾马车”之一的Geoffrey Hinton创造性地将非线性的Sigmoid函数应用到了多层感知器中，并利用反向传播（Backpropagation）算法进行模型学习，使得模型能够有效地处理非线性问题。1998年，“三驾马车”中的卷积神经网络之父Yann LeCun发明了卷积神经网络LeNet模型，可有效解决图像数字识别问题，被认为是卷积神经网络的鼻祖。

然而在此之后的多年时间里，深度学习并没有代表性的算法问世，并且神经网络存在两个致命问题：一是Sigmoid在函数两端具有饱和效应，会带来梯度消失问题；另一个是随着神经网络的加深，训练时参数容易陷入局部最优解。这两个弊端导致深度学习陷入了第二次低谷。在这段时间内，反倒是传统的机器学习算法，如支持向量机、随机森林等算法获得了快速的发展。

2006年，Hinton提出了利用无监督的初始化与有监督的微调缓解了

局部最优解问题，再次挽救了深度学习，这一年也被称为深度学习元年。2011年诞生的ReLU激活函数有效地缓解了梯度消失现象。

真正让深度学习迎来爆发式发展的当属2012年的AlexNet网络，其在ImageNet图像分类任务中以“碾压”第二名算法的姿态取得了冠军。深度学习从此一发不可收拾，VGGNet、ResNet等优秀的网络接连问世，并且在分类、物体检测、图像分割等领域渐渐地展现出深度学习的实力，大大超过了传统算法的水平。

当然，深度学习的发展离不开大数据、GPU及模型这3个因素，如图1.6所示。

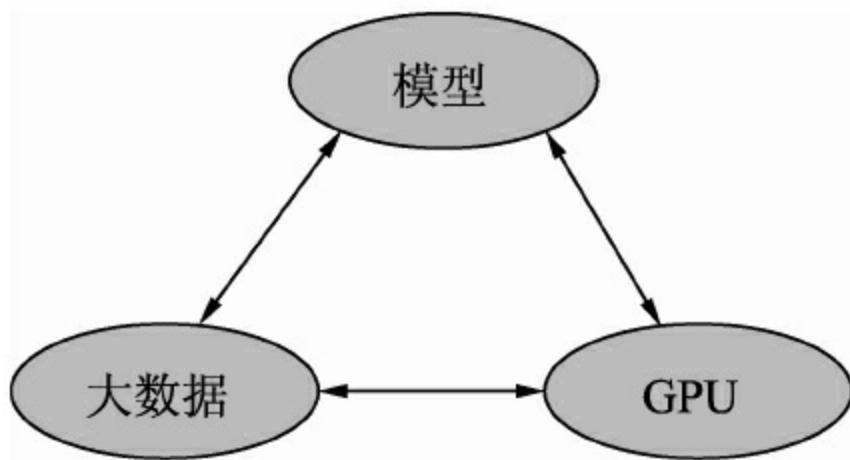


图1.6 深度学习中的核心因素

·**大数据**：当前大部分的深度学习模型是有监督学习，依赖于数据的有效标注。例如，要做一个高性能的物体检测模型，通常需要使用上万甚至是几十万的标注数据。数据的积累也是一个公司深度学习能力雄厚的标志之一，没有数据，再优秀的模型也会面对无米之炊的尴尬。

·**GPU**：当前深度学习如此“火热”的一个很重要的原因就是硬件的发展，尤其是GPU为深度学习模型的快速训练提供了可能。深度学习模

型通常有数以千万计的参数，存在大规模的并行计算，传统的以逻辑运算能力著称的CPU面对这种并行计算会异常缓慢，GPU以及CUDA计算库专注于数据的并行计算，为模型训练提供了强有力的工具。

·模型：在大数据与GPU的强有力支撑下，无数研究学者的奇思妙想，催生出了VGGNet、ResNet和FPN等一系列优秀的深度学习模型，并且在学习任务的精度、速度等指标上取得了显著的进步。

根据网络结构的不同，深度学习模型可以分为卷积神经网络（Convolutional Neural Network, CNN）、循环神经网络（Recurrent Neural Network, RNN）及生成式对抗网络（Generative Adversarial Network, GAN）。

## 1.1.2 计算机视觉

视觉是人类最为重要的感知系统，大脑皮层中近一半的神经元与视觉有关系。计算机视觉则是研究如何使机器学会“看”的学科，最早起源于20世纪50年代，当时主要专注于光学字符识别、航空图片的分析等特定任务。在20世纪90年代，计算机视觉在多视几何、三维重建、相机标定等多个领域取得了众多成果，也走向了繁荣发展的阶段。

然而在很长的一段时间里，计算机视觉的发展都是基于规则与人工设定的模板，很难有鲁棒的高语义理解。真正将计算机视觉的发展推向高峰的，当属深度学习的爆发。由于视觉图像丰富的语义性与图像的结构性，计算机视觉也是当前人工智能发展最为迅速的领域之一。

进入深度学习发展阶段后，计算机视觉在多个领域都取得了令人瞩目的成就，如图1.7所示。

·**图像成像**：成像是计算机视觉较为底层的技术，深度学习在此发挥的空间更多的是成像后的应用，如修复图像的DCGAN网络，图像风格迁移的CycleGAN，这些任务中GAN有着广阔的空间。此外，在医学成像、卫星成像等领域中，超分辨率也至关重要，例如SRCNN（Super-Resolution CNN）。

·**2.5D空间**：我们通常将涉及2D运动或者视差的任务定义为2.5D空间问题，因为其任务跳出了单纯的2D图像，但又缺乏3D空间的信息。这里包含的任务有光流的估计、单目的深度估计及双目的深度估计。

·**3D空间**：3D空间的任务通常应用于机器人或者自动驾驶领域，将2D图像检测与3D空间进行结合。这其中，主要任务有相机标定（Camera Calibration）、视觉里程计（Visual Odometry, VO）及

SLAM (Simultaneous Localization and Mapping) 等。

·环境理解：环境的高语义理解是深度学习在计算机视觉中的主战场，相比传统算法其优势更为明显。主要任务有图像分类（Classification）、物体检测（Object Detection）、图像分割（Segmentation）、物体跟踪（Tracking）及关键点检测。其中，图像分割又可以细分为语义分割（Semantic Segmentation）与实例分割（Instance Segmentation）。

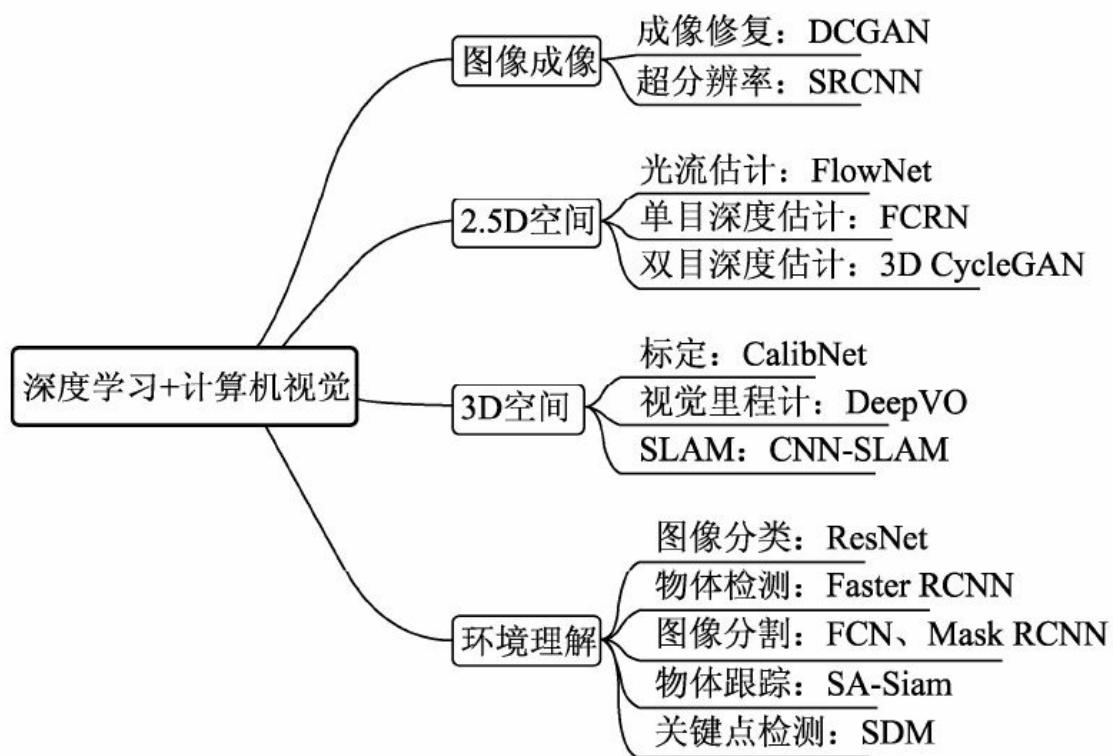


图1.7 深度学习在计算机视觉中的应用

## 1.2 物体检测技术

在计算机视觉众多的技术领域中，物体检测是一项非常基础的任务，图像分割、物体追踪、关键点检测等通常都要依赖于物体检测。此外，由于每张图像中物体的数量、大小及姿态各不相同，也就是非结构化的输出，这是与图像分类非常不同的一点，并且物体时常会有遮挡截断，物体检测技术也极富挑战性，从诞生以来始终是研究学者最为关注的焦点领域之一。

物体检测技术，通常是指在一张图像中检测出物体出现的位置及对应的类别。对于图1.8中的人，我们要求检测器输出5个量：物体类别、 $x_{min}$ 、 $y_{min}$ 、 $x_{max}$ 与 $y_{max}$ 。当然，对于一个边框，检测器也可以输出中心点与宽高的形式，这两者是等价的。



图1.8 物体检测示例

在计算机视觉中，图像分类、物体检测与图像分割是最基础、也是目前发展最为迅速的3个领域。图1.9列出了这3个任务之间的区别。



图1.9 图像分类、物体检测与图像分割的区别

·**图像分类**：输入图像往往仅包含一个物体，目的是判断每张图像是什么物体，是图像级别的任务，相对简单，发展也最快。

·**物体检测**：输入图像中往往有很多物体，目的是判断出物体出现的位置与类别，是计算机视觉中非常核心的一个任务。

·**图像分割**：输入与物体检测类似，但是要判断出每一个像素属于哪一个类别，属于像素级的分类。图像分割与物体检测任务之间有很多联系，模型也可以相互借鉴。

### 1.2.1 发展历程

在利用深度学习做物体检测之前，传统算法对于物体的检测通常分为区域选取、特征提取与特征分类这3个阶段，如图1.10所示。

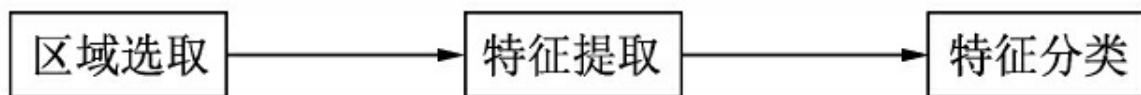


图1.10 传统物体检测算法思路

·区域选取：首先选取图像中可能出现物体的位置，由于物体位置、大小都不固定，因此传统算法通常使用滑动窗口（Sliding Windows）算法，但这种算法会存在大量的冗余框，并且计算复杂度高。

·特征提取：在得到物体位置后，通常使用人工精心设计的提取器进行特征提取，如SIFT和HOG等。由于提取器包含的参数较少，并且人工设计的鲁棒性较低，因此特征提取的质量并不高。

·特征分类：最后，对上一步得到的特征进行分类，通常使用如SVM、AdaBoost的分类器。

深度学习时代的物体检测发展过程如图1.11所示。深度神经网络大量的参数可以提取出鲁棒性和语义性更好的特征，并且分类器性能也更优越。2014年的RCNN（Regions with CNN features）算是使用深度学习实现物体检测的经典之作，从此拉开了深度学习做物体检测的序幕。

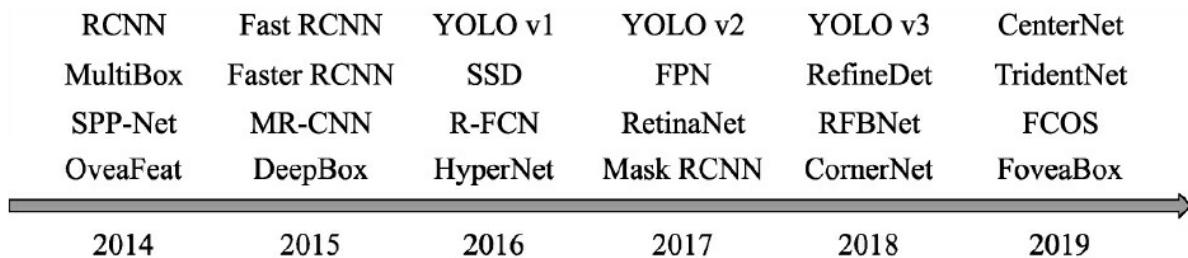


图1.11 深度学习检测器的发展历程

在RCNN基础上，2015年的Fast RCNN实现了端到端的检测与卷积共享，Faster RCNN提出了锚框（Anchor）这一划时代的思想，将物体检测推向了第一个高峰。在2016年，YOLO v1实现了无锚框（Anchor-Free）的一阶检测，SSD实现了多特征图的一阶检测，这两种算法对随后的物体检测也产生了深远的影响，在本书中将分别用一章的篇幅详细介绍。

在2017年，FPN利用特征金字塔实现了更优秀的特征提取网络，Mask RCNN则在实现了实例分割的同时，也提升了物体检测的性能。进入2018年后，物体检测的算法更为多样，如使用角点做检测的CornerNet、使用多个感受野分支的TridentNet、使用中心点做检测的CenterNet等。

在物体检测算法中，物体边框从无到有，边框变化的过程在一定程度上体现了检测是一阶的还是两阶的。

- 两阶：两阶的算法通常在第一阶段专注于找出物体出现的位置，得到建议框，保证足够的准召率，然后在第二个阶段专注于对建议框进行分类，寻找更精确的位置，典型算法如Faster RCNN。两阶的算法通常精度准更高，但速度较慢。当然，还存在例如Cascade RCNN这样更多阶的算法。

- 一阶：一阶的算法将二阶算法的两个阶段合二为一，在一个阶段

里完成寻找物体出现位置与类别的预测，方法通常更为简单，依赖于特征融合、Focal Loss等优秀的网络经验，速度一般比两阶网络更快，但精度会有所损失，典型算法如SSD、YOLO系列等。

Anchor是一个划时代的思想，最早出现在Faster RCNN中，其本质上是一系列大小宽高不等的先验框，均匀地分布在特征图上，利用特征去预测这些Anchors的类别，以及与真实物体边框存在的偏移。Anchor相当于给物体检测提供了一个梯子，使得检测器不至于直接从无到有地预测物体，精度往往较高，常见算法有Faster RCNN和SSD等。

当然，还有一部分无锚框的算法，思路更为多样，有直接通过特征预测边框位置的方法，如YOLO v1等。最近也出现了众多依靠关键点来检测物体的算法，如CornerNet和CenterNet等。

## 1.2.2 技术应用领域

由于检测性能的迅速提升，物体检测也是深度学习在工业界取得大规模应用的领域之一。以下列举了5个广泛应用的领域。

·**安防：**受深度学习的影响，安防领域近年来取得了快速的发展与落地。例如广为人知的人脸识别技术，在交通卡口、车站等已有了成熟的应用。此外，在智慧城市的安防中，行人与车辆的检测也是尤为重要的一个环节。在安防领域中，有很大的趋势是将检测技术融入到摄像头中，形成智能摄像头，以海康威视、地平线等多家公司最为知名。

·**自动驾驶：**自动驾驶的感知任务中，行人、车辆等障碍物的检测尤为重要。由于涉及驾驶的安全性，自动驾驶对于检测器的性能要求极高，尤其是召回率这个指标，自动驾驶也堪称人工智能应用的“珠穆朗玛峰”。此外，由于车辆需要获取障碍物相对于其自身的三维位置，因此通常在检测器后还需要增加很多的后处理感知模块。

·**机器人：**工业机器人自动分拣中，系统需要识别出要分拣的各种部件，这是极为典型的机器人应用领域。此外，移动智能机器人需要时刻检测出环境中的各种障碍物，以实现安全的避障与导航。从广泛意义来看，自动驾驶车辆也可以看做是机器人的一种形式。

·**搜索推荐：**在互联网公司的各大应用平台中，物体检测无处不在。例如，对于包含特定物体的图像过滤、筛选、推荐和水印处理等，在人脸、行人检测的基础上增加更加丰富的应用，如抖音等产品。

·**医疗诊断：**基于人工智能与大数据，医疗诊断也迎来了新的春天，利用物体检测技术，我们可以更准确、迅速地对CT、MR等医疗图像中特定的关节和病症进行诊断。

### 1.2.3 评价指标

对于一个检测器，我们需要制定一定的规则来评价其好坏，从而选择需要的检测器。对于图像分类任务来讲，由于其输出是很简单的图像类别，因此很容易通过判断分类正确的图像数量来进行衡量。

物体检测模型的输出是非结构化的，事先并无法得知输出物体的数量、位置、大小等，因此物体检测的评价算法就稍微复杂一些。对于具体的某个物体来讲，我们可以从预测框与真实框的贴合程度来判断检测的质量，通常使用IoU（Intersection of Union）来量化贴合程度。

IoU的计算方式如图1.12所示，使用两个边框的交集与并集的比值，就可以得到IoU，公式如式（1-1）所示。显而易见，IoU的取值区间是[0,1]，IoU值越大，表明两个框重合越好。

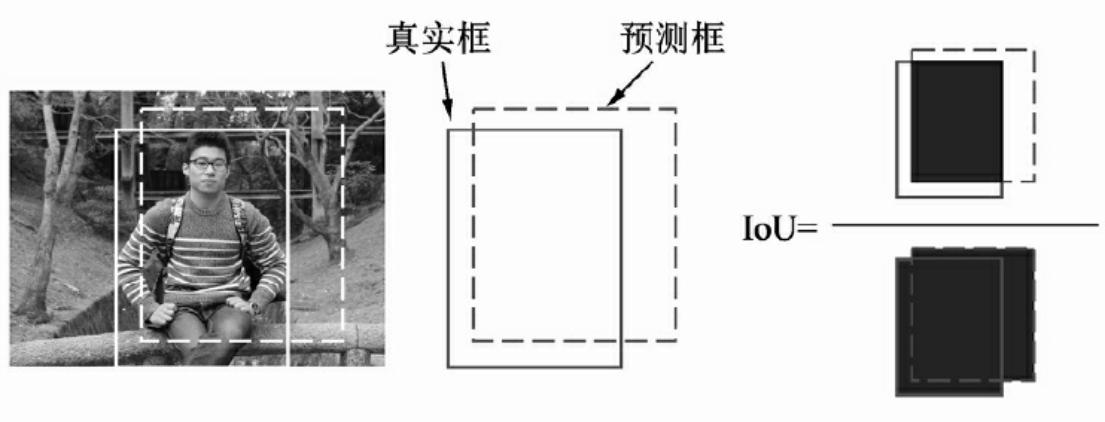


图1.12 IoU的计算过程

$$\text{IoU}_{A,B} = \frac{S_A \cap S_B}{S_A \cup S_B} \quad (1-1)$$

利用Python可以很方便地实现IoU的计算，代码如下：

```
def iou(boxA, boxB):
    # 计算重合部分的上、下、左、右4个边的值，注意最大最小函数的使用
    left_max = max(boxA[0], boxB[0])
    top_max = max(boxA[1], boxB[1])
    right_min = min(boxA[2], boxB[2])
    bottom_min = min(boxA[3], boxB[3])
    # 计算重合部分的面积
    inter = max(0, (right_min-left_max))* max(0, (bottom_min-top_max))
    Sa = (boxA[2]-boxA[0])*(boxA[3]-boxA[1])
    Sb = (boxB[2]-boxB[0])*(boxB[3]-boxB[1])
    # 计算所有区域的面积并计算iou，如果是Python 2，则要增加浮点化操作
    union = Sa+Sb-inter
    iou = inter/union
    return iou
```

对于IoU而言，我们通常会选取一个阈值，如0.5，来确定预测框是正确的还是错误的。当两个框的IoU大于0.5时，我们认为是一个有效的检测，否则属于无效的匹配。如图1.13中有两个杯子的标签，模型产生了两个预测框。

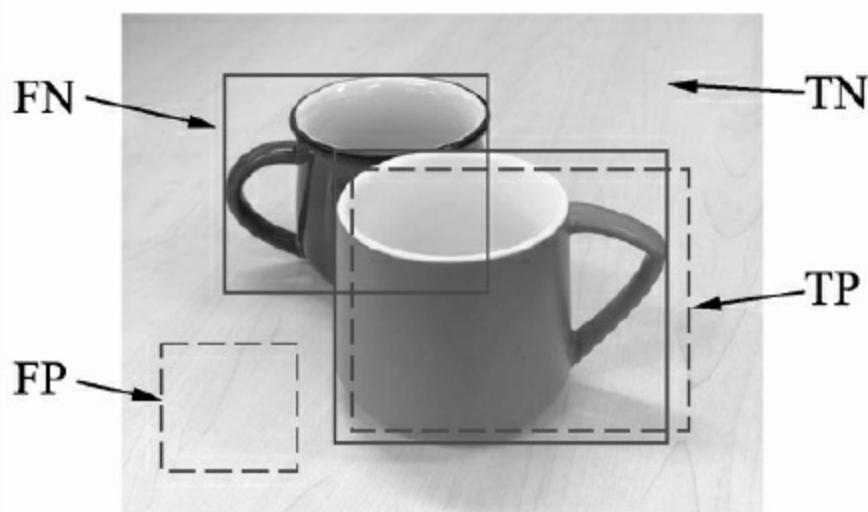


图1.13 正、负样本判别示例

由于图像中存在背景与物体两种标签，预测框也分为正确与错误，因此在评测时会产生以下4种样本。

·正确检测框TP（True Positive）：预测框正确地与标签框匹配了，两者间的IoU大于0.5，如图1.13中右下方的检测框。

·误检框FP（False Positive）：将背景预测成了物体，如图1.13中左下方的检测框，通常这种框与图中所有标签的IoU都不会超过0.5。

·漏检框FN（False Negative）：本来需要模型检测出的物体，模型没有检测出，如图1.13中左上方的杯子。

·正确背景（True Negative）：本身是背景，模型也没有检测出来，这种情况在物体检测中通常不需要考虑。

有了上述基础知识，我们就可以开始进行检测模型的评测。对于一个检测器，通常使用mAP（mean Average Precision）这一指标来评价一个模型的好坏，这里的AP指的是一个类别的检测精度，mAP则是多个类别的平均精度。评测需要每张图片的预测值与标签值，对于某一个实例，二者包含的内容分别如下：

·预测值（Dets）：物体类别、边框位置的4个预测值、该物体的得分。

·标签值（GTs）：物体类别、边框位置的4个真值。

在预测值与标签值的基础上，AP的具体计算过程如图1.14所示。我们首先将所有的预测框按照得分从高到低进行排序（因为得分越高的边框其对于真实物体的概率往往越大），然后从高到低遍历预测框。

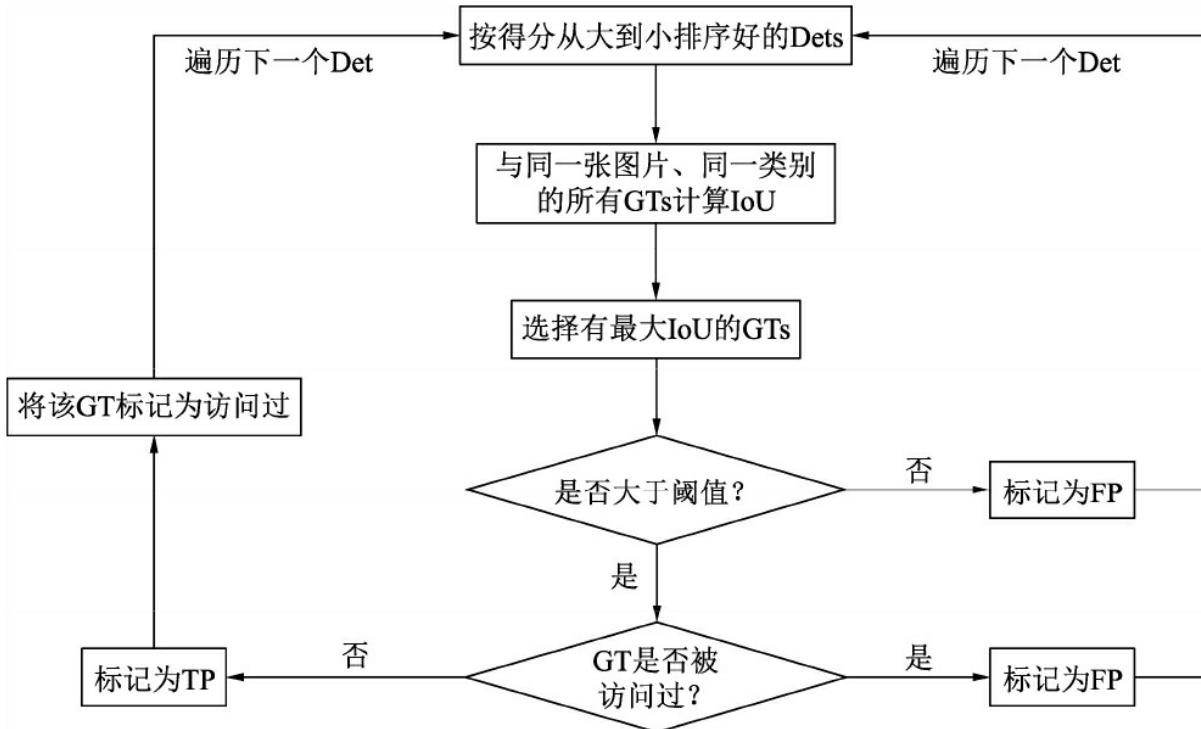


图1.14 AP的计算过程

对于遍历中的某一个预测框，计算其与该图中同一类别的所有标签框GTs的IoU，并选取拥有最大IoU的GT作为当前预测框的匹配对象。如果该IoU小于阈值，则将当前的预测框标记为误检框FP。

如果该IoU大于阈值，还要看对应的标签框GT是否被访问过。如果前面已经有得分更高的预测框与该标签框对应了，即使现在的IoU大于阈值，也会被标记为FP。如果没有被访问过，则将当前预测框Det标记为正确检测框TP，并将该GT标记为访问过，以防止后面还有预测框与其对应。

在遍历完所有的预测框后，我们会得到每一个预测框的属性，即TP或FP。在遍历的过程中，我们可以通过当前TP的数量来计算模型的召回率（Recall，R），即当前一共检测出的标签框与所有标签框的比值，如式（1-2）所示。

$$R = \frac{TP}{len(GTs)} \quad (1-2)$$

除了召回率，还有一个重要指标是准确率（Precision，P），即当前遍历过的预测框中，属于正确预测边框的比值，如式（1-3）所示。

$$P = \frac{TP}{TP + FP} \quad (1-3)$$

遍历到每一个预测框时，都可以生成一个对应的P与R，这两个值可以组成一个点(R,P)，将所有的点绘制成曲线，即形成了P-R曲线，如图1.15所示。

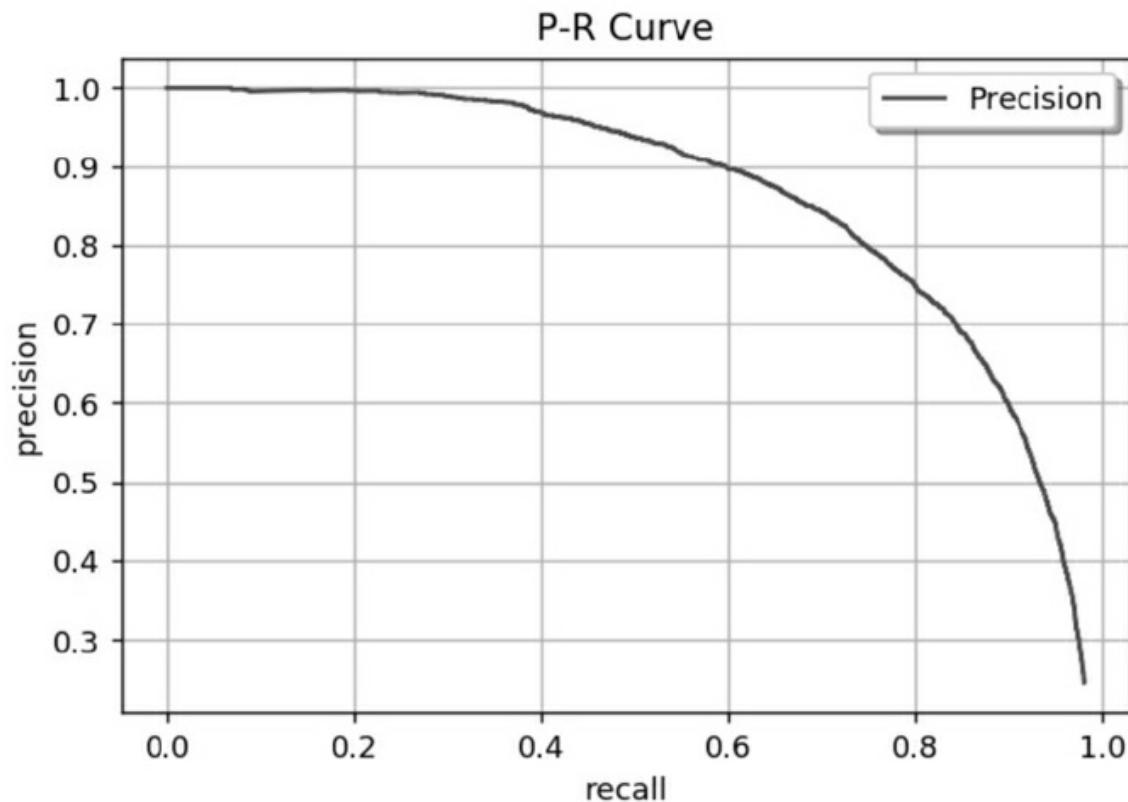


图1.15 物体检测的P-R曲线

然而，即使有了P-R曲线，评价模型仍然不直观，如果直接取曲线上的点，在哪里选取都不合适，因为召回率高的时候准确率会很低，准确率高的时候往往召回率很低。这时，AP就派上用场了，计算公式如式（1-4）所示。

$$AP = \int_0^1 P dR \quad (1-4)$$

从公式中可以看出，AP代表了曲线的面积，综合考量了不同召回率下的准确率，不会对P与R有任何偏好。每个类别的AP是相互独立的，将每个类别的AP进行平均，即可得到mAP。严格意义上讲，还需要对曲线进行一定的修正，再进行AP计算。除了求面积的方式，还可以使用11个不同召回率对应的准确率求平均的方式求AP。

下面从代码层面详细讲述AP求解过程。假设当前经过标签数据与预测数据的加载，我们得到了下面两个变量：

·`det_boxes`: 包含全部图像中所有类别的预测框，其中一个边框包含了[`left,top,right,bottom,score,NameofImage`]。

·`gt_boxes`: 包含了全部图像中所有类别的标签，其中一个标签的内容为[`left,top,right,bottom,0`]。需要注意的是，最后一位0代表该标签有没有被匹配过，如果匹配过则会置为1，其他预测框再去匹配则为误检框。

下面是所有类别的评测过程。

---

```
for c in classes:  
    # 通过类别作为关键字，得到每个类别的预测、标签及总标签数  
    dects = det_boxes[c]  
    gt_class = gt_boxes[c]  
    npos = num_pos[c]  
    # 利用得分作为关键字，对预测框按照得分从高到低排序
```

```

dects = sorted(dects, key=lambda conf: conf[5], reverse=True)
# 设置两个与预测边框长度相同的列表，标记是True Positive还是False Positive
TP = np.zeros(len(dects))
FP = np.zeros(len(dects))
# 对某一个类别的预测框进行遍历
for d in range(len(dects)):
    # 将IoU默认置为最低
    iouMax = sys.float_info.min
    # 遍历与预测框同一图像中的同一类别的标签，计算IoU
    if dects[d][-1] in gt_class:
        for j in range(len(gt_class[dects[d][-1]])):
            iou = Evaluator.iou(dects[d][:4], gt_class[dects[d][-1]][j][:4])
            if iou > iouMax:
                iouMax = iou
                jmax = j # 记录与预测有最大IoU的标签
    # 如果最大IoU大于阈值，并且没有被匹配过，则赋予TP
    if iouMax >= cfg['iouThreshold']:
        if gt_class[dects[d][-1]][jmax][4] == 0:
            TP[d] = 1
            gt_class[dects[d][-1]][jmax][4] = 1 # 标记为匹配过
    # 如果被匹配过，赋予FP
    else:
        FP[d] = 1
    # 如果最大IoU没有超过阈值，赋予FP
    else:
        FP[d] = 1
    # 如果对应图像中没有该类别的标签，赋予FP
    else:
        FP[d] = 1
# 利用NumPy的cumsum()函数，计算累计的FP与TP
acc_FP = np.cumsum(FP)
acc_TP = np.cumsum(TP)
rec = acc_TP / npos # 得到每个点的Recall
prec = np.divide(acc_TP, (acc_FP + acc_TP)) # 得到每个点的Precision
# 利用Recall与Precision进一步计算得到AP
[ap, mpre, mrec, ii] = Evaluator.CalculateAveragePrecision(rec, prec)

```

---

## 1.3 PyTorch简介

工欲善其事，必先利其器。对于物体检测的学习，选择一个优秀的框架是一件十分重要的事情。笔者综合考虑了框架性能、简洁性、长远发展等多个因素，最终选择了PyTorch作为本书的深度学习框架。

本节将会介绍PyTorch框架的诞生与发展历程，以及为什么选择PyTorch，最后将介绍多种安装PyTorch的方法。

### 1.3.1 诞生与特点

2017年的1月，来自于Facebook的研究人员在GitHub上推出了PyTorch，并凭借其简洁易用的特性，迅速在人工智能各大应用领域流行起来，一时间成为最炙手可热的深度学习框架。

PyTorch的前身是诞生于2002年的Torch框架。Torch使用了小众化的Lua作为编程接口，虽然对于深度神经网络的实现比较高效，但由于Lua使用人数不多，Torch并没有受到广泛的关注。而在计算科学领域，Python由于其简单易用与完整的生态，优势渐渐体现了出来。在此基础上，2017年Facebook的团队对Torch张量之上的所有模块进行了重构，并增加了自动求导这一先进的理念，实现了一个高效的动态图框架PyTorch。

在2018年NeurIPS大会上，Facebook推出了PyTorch 1.0版本，该版本吸收了Caffe 2和ONNX模块化及面向生产的特点，使得算法可以从研究原型快速无缝衔接到底部部署中。这意味着，PyTorch的生态将更加完备，在进行算法原型研究、移动部署时将更加便利。

### 1.3.2 各大深度学习框架对比

为了更好地支持深度学习算法的研究与落地，各大公司和研究机构陆续推出了多种深度学习框架，除了PyTorch之外，还有几种知名的框架，如TensorFlow、MXNet、Keras、Caffe和Theano等，广泛应用于计算机视觉、自然语言处理、语音识别等领域，如图1.16所示。



图1.16 各大主流深度学习计算框架

不同的深度学习框架通常有不同的设计理念，优势和劣势也各不相同，本节将简要介绍这几种主流的框架。

#### 1. TensorFlow简介

2015年，谷歌大脑（Google Brain）团队推出了深度学习开源框架TensorFlow，其前身是谷歌的DistBelief框架。凭借其优越的性能与谷歌在深度学习领域的巨大影响力，TensorFlow一经推出就引起了广泛的的关注，逐渐成为用户最多的深度学习框架。

TensorFlow使用数据流图进行网络计算，图中的节点代表具体的数学运算，边则代表了节点之间流动的多维张量Tensor。TensorFlow有Python与C++两种编程接口，并且随着发展，也渐渐支持Java、Go等语言，并且可以在ARM移动式平台上进行编译与优化，因此，TensorFlow拥有非常完备的生态与生产环境。

TensorFlow的优点很明显，功能完全，对于多GPU的支持更好，有强大且活跃的社区，并且拥有强大的可视化工具TensorBoard。当然，其缺点在于系统设计较为复杂，接口变动较快，兼容性较差，并且由于其构造的图是静态的，导致图必须先编译再执行，不利于算法的预研等。

## 2. MXNet简介

MXNet由DMLC（Distributed Machine Learning Community）组织创建，该组织成员以陈天奇、李沐为代表，大部分都为中国人。MXNet项目于2015年在GitHub上正式开源，于2016年被亚马逊AWS正式选择为其云计算的官方深度学习平台，并在2017年进入Apache软件基金会，正式成为了Apache的孵化器项目。

MXNet的特色是将命令式编程与声明式编程进行结合，在命令式编程上提供了张量计算，在声明式编程上支持符号表达式，用户可以自由地进行选择。此外，MXNet提供了多种语言接口，有超强的分布式支持，对于内存和显存做了大量优化，尤其适用于分布式环境中。

但是，MXNet的推广不够有力，文档的更新没有跟上框架的迭代速度，导致新手上手MXNet较难，因此MXNet也一直没有得到大规模的应用。目前国内的众多AI创业公司都在使用MXNet框架。

## 3. Keras简介

Keras是建立在TensorFlow、Theano及CNTK等多个框架之上的神经网络API，对深度学习的底层框架作了进一步封装，提供了更为简洁、易上手的API。Keras使用Python语言，并且可以在CPU与GPU间无缝切换。

Keras的首要设计原则就是用户的使用体验，把神经网络模块化，因此Keras使用相对简单，入门快。但是，Keras构建于第三方框架之上，导致其灵活性不足，调试不方便，用户在使用时也很难学习到神经网络真正的内容。从性能角度看，Keras也是较慢的一个框架。

#### 4. Caffe与Caffe 2简介

Caffe（Convolutional Architecture for Fast Feature Embedding）发布于2013年，作者是贾扬清博士。Caffe的核心语言是C++，支持CPU与GPU两种模式的计算。Caffe的优点是设计清晰、实现高效，尤其是对于C++的支持，使工程师可以方便地在各种工程应用中部署Caffe模型，曾经占据了神经网络框架的半壁江山。

Caffe的主要缺点是灵活性不足。在Caffe中，实现一个神经网络新层，需要利用C++来完成前向传播与反向传播的代码，并且需要编写CUDA代码实现在GPU端的计算，总体上更偏底层，显然与当前深度学习框架动态图、灵活性的发展趋势不符。

在2017年，Facebook推出了Caffe 2，主要开发者仍然是贾扬清博士。Caffe 2是一个轻量化与跨平台的深度学习框架，继承了Caffe大量的设计理念，同时为移动端部署做了很多优化，性能极佳。Caffe 2的核心库是C++，但也提供了Python的API，可以方便地在多个平台进行模型训练与部署。

作为Facebook推出的框架，Caffe 2的优点是高性能与跨平台部署。

PyTorch的优点是灵活性与原型的快速实现。为了进一步提升开发者的效率，Facebook于2018年宣布将Caffe 2代码全部并入PyTorch。

### 1.3.3 为什么选择PyTorch

前面介绍的深度学习框架多多少少都有一些缺陷，而PyTorch作为后起之秀，是一个难得的集简洁与高性能于一身的一个框架，笔者在本书中选择PyTorch来实现物体检测算法，主要原因有4点，如图1.17所示。

·简洁优雅：PyTorch是一个十分Pythonic的框架，代码风格与普通的Python代码很像，甚至可以看做是带有GPU优化的NumPy模块，这使得使用者很容易理解模型的框架与逻辑。此外，PyTorch是一个动态图框架，拥有自动求导机制，对神经网络有着尽量少的概念抽象，这使得使用者更容易调试、了解模型的每一步到底发生了什么，而不至于是一个黑箱。

·易上手：PyTorch与Python一样，追求用户的使用体验，所有的接口都十分易用，对于使用者极为友好。与Keras不同的是，在接口易用的同时，PyTorch很难得地保留了灵活性，使用者可以利用PyTorch自由地实现自己的算法。另外，PyTorch的文档简洁又精髓，建议初学者可以通读一遍。

·速度快：PyTorch在追求简洁易用的同时，在模型的速度表现上也极为出色，相比TensorFlow等框架，很多模型在PyTorch上的实现可能会更快。这一点也使得学术界有大量PyTorch的忠实用户，因为使用PyTorch既可以快速实现自己的想法，又能够保证优秀的速度性能。

·发展趋势：PyTorch在问世的两年多时间里，发展趋势十分迅猛，fast.ai、NVIDIA等知名公司也选择使用PyTorch作为深度学习框架。PyTorch背后是Facebook人工智能研究院，有这一顶级AI机构强有力的支持，生态完备，尤其是在Caffe 2并入PyTorch之后，PyTorch未来的发

展值得期待。

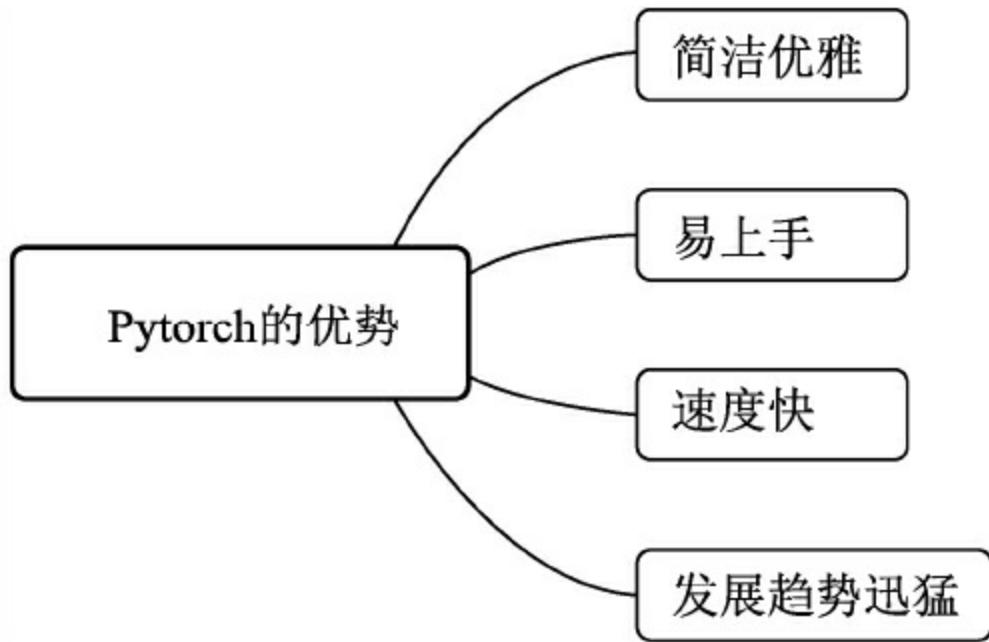


图1.17 PyTorch的优势

### 1.3.4 安装方法

PyTorch目前支持Ubuntu、Mac OS、Windows等多个系统，本书将主要围绕Ubuntu系统进行讲解。PyTorch官网中提供了pip、Conda、源码等多种安装方法，由于源码安装较为复杂，通常情况下使用不到，在此只介绍pip与Conda两种安装方法。

由于本书介绍的物体检测算法模型通常较大，因此我们需要使用带有GPU加速版本的PyTorch。在安装PyTorch之前，你需要有一台带有GPU的机器，并从NVIDIA官网下载安装对应的显卡驱动，在终端中输入以下命令可以证明驱动已装好，并显示GPU的内存、使用情况等信息：

```
nvidia-smi
```

在安装完显卡驱动后，我们还需要安装CUDA（Compute Unified Device Architecture）。CUDA是NVIDIA推出的通用并行计算架构，可以使类似于PyTorch之类的深度学习框架调用GPU来解决复杂的计算问题。

为了实现更高效的GPU并行计算，通常我们还需要安装cuDNN库。cuDNN库是由NVIDIA开发的专用于深度神经网络的GPU加速库。由于CUDA与cuDNN的安装较为简单，在此就不展开了，读者可以参考官网教程进行安装。在完成上述步骤后，即可使用pip或者Conda安装PyTorch。

#### 1. pip安装

pip是一个通用的Python包管理工具，可以便捷地实现Python包的查

找、下载、安装与卸载。使用pip安装PyTorch的方法可以参考官网教程，我们以Python 3.6、CUDA 9.0、PyTorch 0.4.0为例，使用下面的指令即可完成PyTorch安装：

```
pip3 install torch==0.4.0
```

安装后，在终端中输入python3，再输入以下指令：

```
>>> import torch  
>>> torch.cuda.is_available() #判断当前GPU是否可用  
True
```

如果没有报错，则表明PyTorch安装成功。

## 2. Conda安装

Conda是一个开源的软件管理包系统和环境管理系统，可以安装多个版本的软件包，并在其间自由切换。相比于pip，Conda功能更为强大，可以提供多个Python环境，并且解决包依赖的能力更强。我们可以通过安装Anaconda来使用Conda，安装方式可以参考官网教程，在此不再展开介绍。

安装好Conda后，可以使用如下指令安装PyTorch。

```
conda install pytorch=0.4.0 cuda90 -c pytorch
```

 **注意：**在使用pip或者Conda安装软件包时，如果下载速度较慢，可以考虑将pip或者Conda的源更换为国内的源，这样速度会更快。

## 1.4 基础知识准备

在后续的物体检测学习过程中，我们会涉及数据集的操作，以及模型的设计、优化、训练与评测等，这些内容都需要有一定的Linux基础。此外，PyTorch框架使用了Python作为基本编程语言，因此读者也需要具备一定的Python基础。

因此，本节将会简要地介绍本书所需的Linux与Python基础，并介绍几种笔者认为十分好用的工具，可以极大地方便读者的学习。

### 1.4.1 Linux基础

Linux诞生于1991年，是一个免费使用与自由传播的类UNIX操作系统，凭借其稳定的性能与开放的社区，在手机、电脑、超级计算机等各种计算机设备中都能看到Linux的身影。我们现今使用的更多是Linux发行版系统，即将Linux内核与应用软件做了打包，并进行依赖管理，如Ubuntu、CentOS和Fedora等。Linux内核与发行版系统的关系如图1.18所示。需要注意的是，Linux内核是独立的程序，每个发行版维护自己修改的内核。

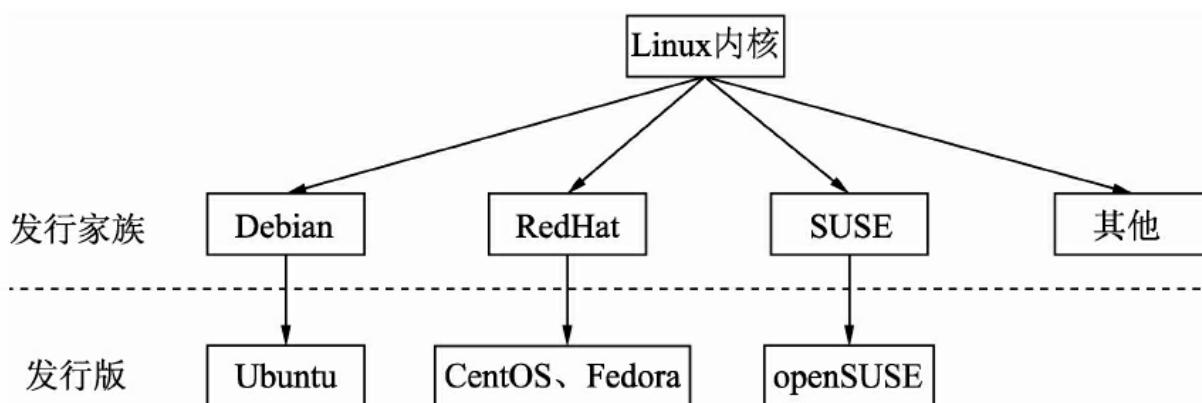


图1.18 Linux内核与多个发行版

由于Linux系统知识树很大，本节不可能对所有知识点都覆盖到，因此这里将简要介绍基本目录结构和环境变量这两个知识点。

#### 1. 基本目录结构

Linux的首要思想是“一切都是文件”，数据与程序都是以文件形式存在，甚至是主机与外围众多设备之间的交互也抽象成对文件的操作，因此对于Linux系统，我们首先需要了解其目录树结构。为了统一各大Linux发行版对目录文件的定义，FHS结构在1994年对Linux根目录做了

统一的规范，必须包含boot、lib、home、usr和opt等文件。下面对几个主要的目录进行说明：

·boot：Linux系统启动时用到的文件，建议单独分区，大小512MB即可。

·lib：系统使用到的函数库目录，协助系统中程序的执行，比较重要的有lib/modules目录，存放着内存文件。

·bin：可执行的文件目录，包含了如ls、mv和cat等常用的命令。

·home：默认的用户目录，包含了所有用户的目录与数据，建议设置较大的磁盘空间。

·usr：应用程序存放的目录，其中，usr/local目录下存放一些软件升级包，如Python、CUDA等，usr/lib目录下存放一些不能直接运行但却是其他程序不可或缺的库文件，usr/share目录下存放一些共享的数据。

·opt：额外安装的软件所在的目录，例如常用的ROS可执行文件，一般就存放在opt/ros目录下。

## 2. 环境变量

环境变量，通俗讲是指操作系统执行程序时默认设定的参数，如各种可执行文件、库的路径等。在本书中，我们将会用到Python、PyTorch和CUDA等多个库，具体使用时调用的是哪一个版本的库，需要设定环境变量。

环境变量有系统级与用户级之分。系统级的环境变量是每一个登录到系统的用户都要读取的变量，可通过以下两个文件进行设置：

·/etc/environment：用于为所有进程设置环境变量，是系统登录时读

取的第一个文件，与登录用户无关，一般要重启系统才会生效。

`./etc/profile`: 用于设置针对系统所有用户的环境变量，是系统登录时读取的第二个文件，与登录用户有关。

用户级环境变量是指针对当前登录用户设定的环境变量，也可以通过两个文件进行设置：

`~/.profile`: 对应当前用户的profile文件，用于设置当前用户的工作环境，默认执行一次。

`~/.bashrc`: 对应当前用户的bash初始化文件，每打开一个终端，就会被执行一次。

我们可以在终端中使用echo来查看当前的环境变量，例如：

---

```
echo $PYTHONPATH
```

---

如果想永久性地设置环境变量，可以在上述的多种文件中添加环境变量。例如我们想指定使用的Python路径，可以修改`~/.bashrc`文件，在最下方添加：

---

```
export PYTHONPATH=/home/yourpythonpath:$PYTHONPATH
```

---

在终端中执行`source ~/.bashrc`即可生效。

如果仅仅是临时设置环境变量，可以在终端中使用`set`或者`export`命令，这种方式只在当前终端中生效，对其他终端无效。

---

```
export PYTHONPATH=/home/yourpythonpath
```

---

## 1.4.2 Python基础

Python是一种面向对象的动态类型语言，得益于其优雅、简单与明确的设计哲学。自20世纪90年代诞生到现在，Python在人工智能、大数据、网页处理等领域取得了大规模的应用，也是当今最炙手可热的语言之一。Python也是PyTorch深度学习框架的第一语言。

相信阅读本书的读者对Python应该有一定的基础，在本节中我将用尽量简短的篇幅，介绍Python语言的一些重要特性与使用方法。

### 1. 变量与对象

掌握一门语言，首先要了解其对于变量与对象的定义。在Python中，变量与对象的概念分别如下：

·**对象**：内存中存储数据的实体，有明确的类型。Python中的一切都是对象，函数也属于对象。

·**变量**：指向对象的指针，对对象的引用。作为弱类型语言，Python中的变量是没有类型的。

---

```
>>> a = 1                      # 这里1为对象，a是指向对象1的变量
>>> a = 'hello'                 # 变量a可以指向任意的对象，没有类型限制
>>> a
'hello'
```

---

Python中的对象还可以进一步分为可变对象与不可变对象，这一点尤其要注意。

·**不可变对象**：对象对应内存中的值不会变，因此如果指向该对象

的变量被改变了， Python则会重新开辟一片内存， 变量再指向这个新的内存， 包括int、 float、 str、 tuple等。

·可变对象： 对象对应内存中的值可以改变， 因此变量改变后， 该对象也会改变， 即原地修改， 如list、 dict、 set等。

对于不可变对象， 所有指向该对象的变量在内存中共用一个地址：

---

```
>>> a = 1
>>> b = 1
>>> c = a + 0
>>> id(a) == id(b) and id(a) == id(c) # 使用id()函数来查看3个变量的内存地址
True
```

---

如果修改了不可变对象的变量的值，则原对象的其他变量不变；相比之下， 如果修改了可变对象的变量，则相当于可变对象被修改了， 其他变量也会发生变化。

---

```
>>> a = 1
>>> b = a
>>> b = 2
>>> a # 由于a与b指向的1都是不可变对象， 因此改变b的值与a没有关系
1
>>> c = [1]
>>> d = c
>>> d.append(2) # c与d指向的是相同的可变对象， d的操作是原地修改
>>> c
[1, 2]
```

---

 **注意：**当对象的引用计数为0时， 该对象对应的内存会被回收。

Python中的变量也存在深拷贝与浅拷贝的区别， 不可变对象无论深/浅拷贝， 其地址都是一样的， 而可变对象则存在3种情况， 下面以list为例。



---

在Python中，使用一个变量时并不严格要求必须预先声明这个变量，但是在真正使用这个变量之前，它必须被绑定到某个内存对象（被定义、赋值）中。这种变量名的绑定将在当前作用域中引入新的变量，同时屏蔽外层作用域中的同名变量。

---

```
a = 1
def local():
    a = 2      # 由于Python不需要预先声明，因此在局部作用域引入了新的变量，而没有修改全局
local()
print(a)      #这里a的值仍为1
```

---

上述例子中局部变量是无法修改全局变量的。想要实现局部修改全局变量，通常有两种办法，增加global等关键字，或者使用list和dict等可变对象的内置函数。

---

```
a = 1
b = [1]
def local():
    global a          # 使用global关键字，表明在局部使用的是全局的a变量
    a = 2
    b.append(2)       #对于可变对象，使用内置函数则会修改全局变量
local()
print(a)            #这里a的值已经被改变为2
print(b)            #这里输出的是[1, 2]
```

---

Python的作用域从内而外，可以分为Local（局部）、Enclosed（嵌套）、Global（全局）及Built-in（内置）4种，如图1.19所示。变量的搜索遵循LEGB原则，如果一直搜索不到则会报错。

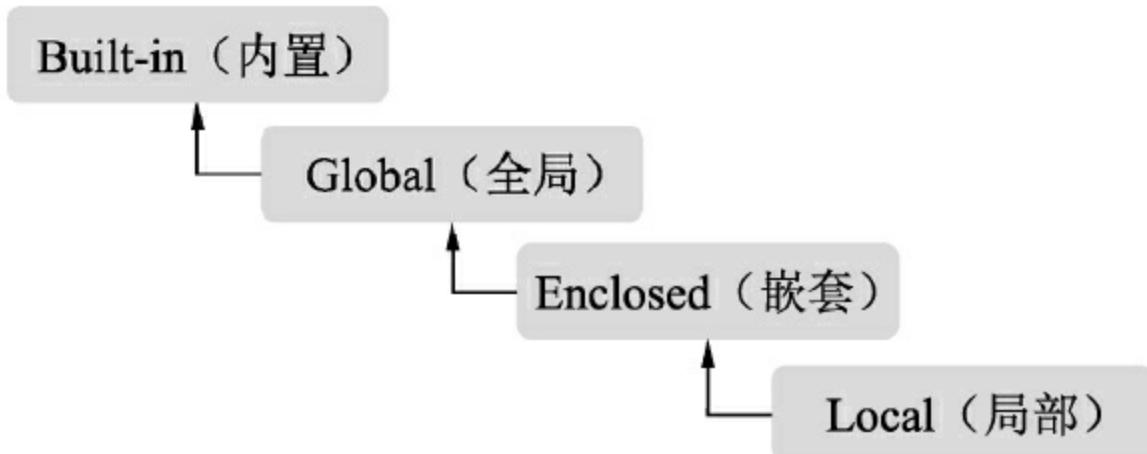


图1.19 Python的LEGB原则

这4种作用域的含义如下：

·局部：在函数与类中，每当调用函数时都会创建一个局部作用域，局部变量域像一个栈，仅仅是暂时存在，依赖于创建该局部作用域的函数是否处于活动的状态。

·嵌套：一般出现在函数中嵌套了一个函数时，在外围函数中的作用域称为嵌套作用域，主要目的是为了实现闭包。

·全局：模型文件顶层声明的变量具有全局作用域，从外部看来，模块的全局变量就是一个模块对象的属性，全局作用域仅限于单个模块的文件中。

·内置：系统内解释器定义的变量。这种变量的作用域是解释器在则在，解释器亡则亡。

### 3. 高阶函数

在编程语言中，高阶函数是指接受函数作为输入或者输出的函数。对于Python而言，函数是一等对象，即可以赋值给变量、添加到集合

中、传参到函数中，也可以作为函数的返回值。下面介绍map()、reduce()、filter()和sorted()这4种常见的高阶函数。

首先，Python中的变量可以指向函数：

```
>>> f = abs
>>> f(-1) # 这里的f(-1)是abs函数
1
```

map()函数可以将一个函数映射作用到可迭代的序列中，并返回函数输出的序列：

```
>>> def f(x): return x * x
>>> map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9]) # 将定义的函数f依次作用于列表的各个元素
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> map(str, range(4))
['0', '1', '2', '3']
```

reduce()函数与map()函数不同，其输入的函数需要传入两个参数。reduce()的过程是先使用输入函数对序列中的前两个元素进行操作，得到的结果再和第三个元素进行运算，直到最后一个元素。

```
# reduce的计算过程: reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
>>> from functools import reduce      # 需要从functools引入reduce函数
>>> def f(x, y): return x*10+y
>>> reduce(f, [1, 3, 5, 7, 9])
13579
```

filter()函数的作用主要是通过输入函数对可迭代序列进行过滤，并返回满足过滤条件的可迭代序列。

```
>>> def is_odd(n): return n % 2 == 0
>>> filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])    # 对奇偶过滤，保留偶数
[2, 4, 6, 10]
```

---

`sorted()`函数可以完成对可迭代序列的排序。与列表本身自带的`sort()`函数不同，这里的`sorted()`函数返回的是一个新的列表。`sorted()`函数可以传入关键字`key`来指定排序的标准，参数`reverse`代表是否反向。

---

```
sorted([3, 5, -87, 0, -21], key=abs, reverse=True) # 绝对值排序，并且为反序  
[-87, -21, 5, 3, 0]
```

---

对于一些简单逻辑函数，可以使用`lambda`匿名表达式来取代函数的定义，这样可以节省函数名称的定义，以及简化代码的可读性等。

---

```
>>> add = lambda x, y : x+y # 使用lambda方便地实现add函数  
>>> add(1,2)  
3  
>>> map(lambda x: x+1, [1, 2, 3, 4, 5, 6, 7, 8, 9]) # lambda实现元素加1的操作  
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

---

#### 4. 迭代器与生成器

迭代器（Iterator）与生成器（Generator）是Python最强大的功能之一，尤其是在处理大规模数据序列时，会带来诸多便利。在检测模型训练时，图像与标签数据的加载通常就是利用迭代生成器实现的。

迭代器不要求事先准备好整个迭代过程中所有的元素，可以使用`next()`来访问元素。Python中的容器，如`list`、`dict`和`set`等，都属于可迭代对象，对于这些容器，我们可以使用`iter()`函数封装成迭代器。

---

```
>>> x = [1, 2, 3]  
>>> y = iter(x)  
>>> z = iter(x)  
>>> next(y), next(y), next(z) # 迭代器之间相互独立  
(1, 2, 1)
```

---

实际上，任何实现了`__iter__()`和`__next__()`方法的对象都是迭代器，其中`__iter__()`方法返回迭代器本身，`__next__()`方法返回容器中的下一个值。`for`循环本质上也是一个迭代器的实现，作用于可迭代对象，在遍历时自动调用`next()`函数来获取下一个元素。

生成器是迭代器的一种，可以控制循环遍历的过程，实现一边循环一边计算，并使用`yield`来返回函数值，每次调用到`yield`会暂停。生成器迭代的序列可以不是完整的，从而可以节省出大量的内存空间。

有多种创建迭代器的方法，最简单的是使用生成器表达式，与`list`很相似，只不过使用`()`括号。

---

```
>>> a = (x for x in range(10))          # 利用()括号实现了一个简单的生成器
>>> next(a), next(a)
(0, 1)
```

---

最为常见的是使用`yield`关键字来创建一个生成器。例如下面代码中，第一次调用`f()`函数会返回1并保持住，第二次调用`f()`会继续执行，并返回2。

---

```
>>> def f():
...     yield 1
...     yield 2
...     yield 3
>>> f1 = f()
>>> print([next(f1) for i in range(2)])
[1, 2]
```

---

使用生成器还可以便捷地实现斐波那契数列的生成，如下：

---

```
def fibonacci():
    a = [1, 1]
    while True:
        a.append(sum(a))           # 往列表里添加下一个元素
```

```
        yield a.pop(0)          # 取出第0个元素，并停留在当前执行点
for x in fibonacci():
    print(x)
    if x > 50:
        break               # 仅打印小于50的数字
```

---

执行上述代码，则会生成小于50的斐波那契数列的数字。

### 1.4.3 高效开发工具

为了更好地探索深度学习，享受coding的乐趣，笔者在此推荐几种日常使用的工具，可以起到事半功倍的效果，如图1.20所示。

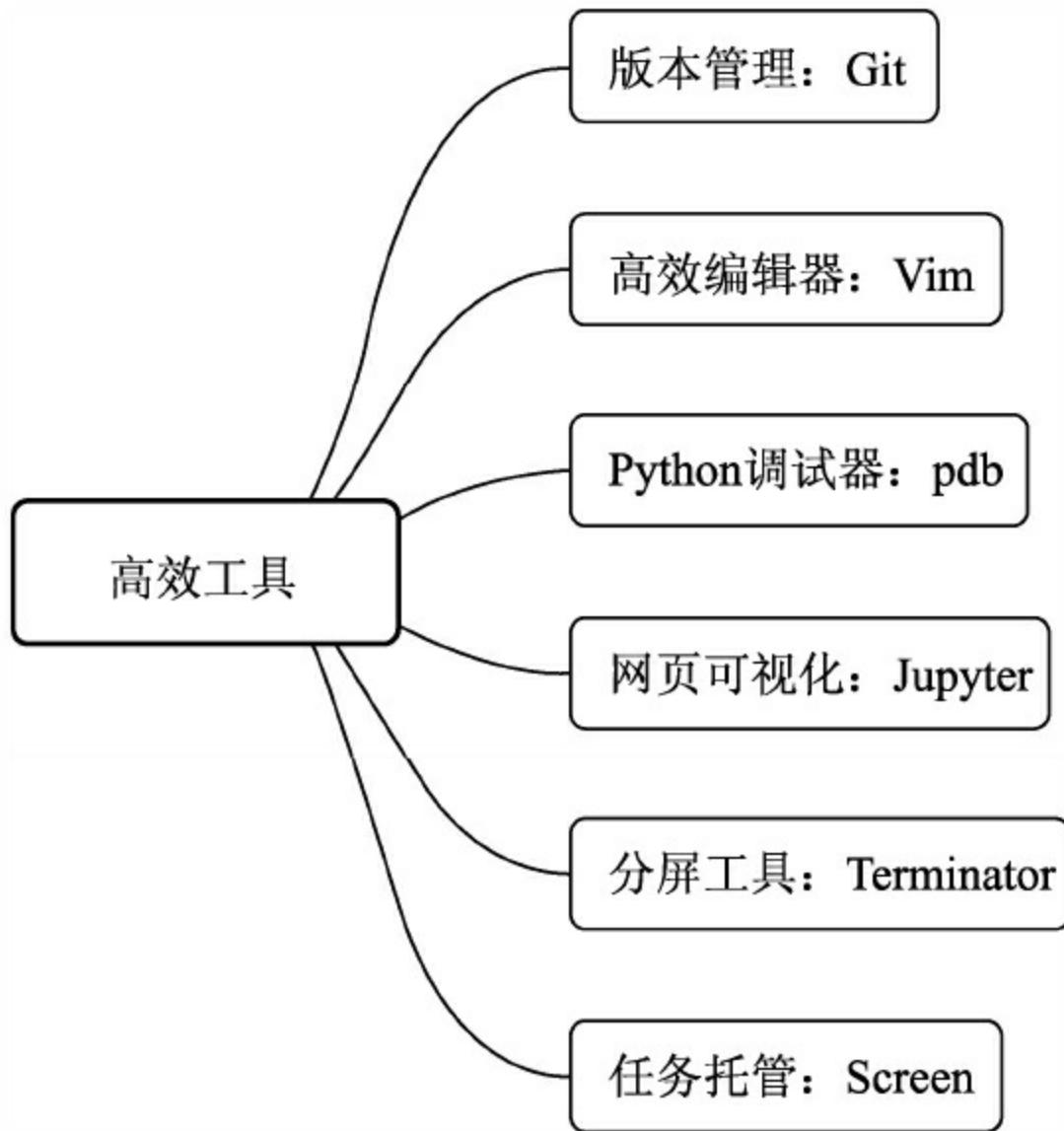


图1.20 高效开发工具

#### 1. 版本管理: Git

Git是一个开源的分布式版本控制系统，用于高效敏捷开发工程项目。有了Git，开发人员就不必将不同版本的文件复制成不同的副本，而是可以通过Git系统的版本控制来完成，尤其是在多人协同开发时，会提供诸多便利。在深度学习领域，大部分的模型与工程都是使用Git进行维护开发的，包括本书的代码，因此希望每一位读者都能打好Git的基础，受用终生。

Git目前支持Linux、Mac OS、Windows等系统，在此我们以Ubuntu系统为例。

利用如下指令即可进行Git的安装。

```
sudo apt install git
```

安装完后，为了使用便捷，可以利用git config指令来配置自己的邮箱与密码信息，具体如下：

```
git config --global user.name "your name"  
git config --global user.email "youremail@youremail.com"
```

使用git init指令可以初始化一个Git仓库，执行后会在当前目录中生成一个.git目录，其中保存了所有有关Git的数据，但切勿手动更改.git里的文件。

Git根据文件的存在位置，有工作区、暂存区与版本库区3个概念，如图1.21所示。

这3个概念的含义如下：

- 工作区：计算机里能看到的实际目录。工作区修改后执行git add命

令，暂存区的内容会被更新，修改或者新增的文件会被写入git对象区，将对象的ID记录在暂存区的索引中。

·暂存区：有时也叫做索引（index），一般存放在.git目录下的index文件中。当执行git commit操作后，暂存区的目录会被写入版本库中，master会做相应的更新。而当执行git checkout与对应的文件时，暂存区的文件会覆盖工作区的文件，清除工作区中还没有添加到暂存区的改动。

·版本库区：工作区下会存在一个.git目录，称之为版本库区。当执行git reset指令时，版本库区的目录树会替换暂存区的目录，但是工作区不受影响。当执行git checkout HEAD时，HEAD指向的master分支会覆盖工作区及暂存区的文件，这个指令较为危险，容易把没有提交的文件清除掉。

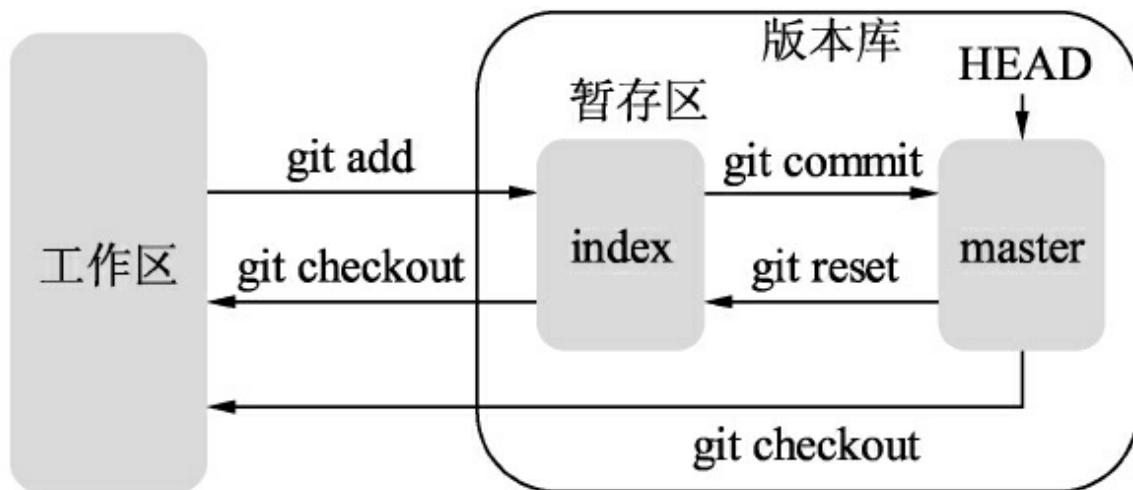


图1.21 Git的文件区间示意

当多人协同开发或者需要有多个不同的版本时，就需要用到Git的分支管理功能了，这也是Git极为强大与重要的功能之一。版本管理主要有以下4个指令：

```
git branch branchname          # 创建一个名为branchname的分支  
git checkout branchname        # 切换到branchname的分支, git branch也可以查看当前的Git分支  
git checkout -b branchname      # 创建一个名为branchname的分支, 并切换到该分支, 十分常用  
git merge                      # 合并分支
```

---

当然, Git还有git log、git status等重要的指令, 在此不一一展开。如果想要使用远程的Git仓库, 不依赖于本地计算机, 并进行仓库的社交, GitHub就派上用场了。

GitHub是一个基于Git的代码托管平台, 是目前最为流行的代码托管服务, 已拥有超过400万个项目。如果想要使用GitHub, 首先需要在其官网注册一个账号, 并使用创建仓库的命令创建名为repository的仓库。

为了能将自己本地的代码提交到GitHub上, 还需要添加GitHub账号可以识别的秘钥, 具体指令如下:

---

```
ssh-keygen -t rsa -C "youremail@youremail.com"
```

---

上述邮箱为注册GitHub时使用的邮箱。执行上述指令后, 一路按回车键, 会在~/.ssh文件夹下生成id\_rsa.pub文件, 复制里边的秘钥并粘贴到GitHub账号的SSH Keys里, 即可实现本地与GitHub仓库的配对。

完成配对后, 可以使用git clone命令将远程仓库拉取到本地。

---

```
git clone username@host:/path/to/repository
```

---

在本地进行代码的开发, 并以此执行git add、git commit命令后, 使用git push可以实现将本地版本仓库内的代码推到GitHub上。

---

```
git push origin master
```

---

总之，Git与GitHub的功能很强大，本书也仅仅做了简要的介绍，相信读者在日后的开发中会渐渐掌握更多的Git知识，领悟Git的威力。

## 2. 高效编辑器：Vim

Vim是一个功能强大且可以定制功能的文本编辑器，尤其适用于面向程序开发，提供了代码补全、编译及错误跳转等功能。Vim由Vi发展而来，Vi是Linux与UNIX系统中最基本的文本编辑器，Vim对Vi进行了诸多改良，在2000年赢得了Slashdot Beanie的最佳开放源代码编辑器大奖。

也许有很多读者喜欢使用如Visual Studio、Clion等IDE软件来进行程序开发，但是在深度学习模型开发中，很多时候我们会把模型放到远程服务器中，这时IDE就不适用了。此外，IDE较大，在很多平台中安装并不方便，并且调试开发程序不灵活，开发人员也无法掌握其编译、配置的精髓。因此，在本书的学习中，Vim编辑器是一个必备的工具。

在Linux终端中，使用`vim file`命令即可以打开一个Vim环境。Vim有下面3种基本模式。

- 命令模式：刚打开Vim时就进入了命令模式，此时敲入任何字母都代表了命令，而不是直接插入到光标处。命令模式下有一些常用的基本命令，可以有效提升开发者的效率。

- 输入模式：在命令模式中输入*i*字符就进入了输入模式，在该模式下可以进行代码的增、删等操作。

- 底线命令模式：在命令模式下输入“`:`”（英文冒号）即进入底线命令模式，这里有更为丰富的命令，如保存、退出和跳转等。

以上3种模式的切换方式如图1.22所示。

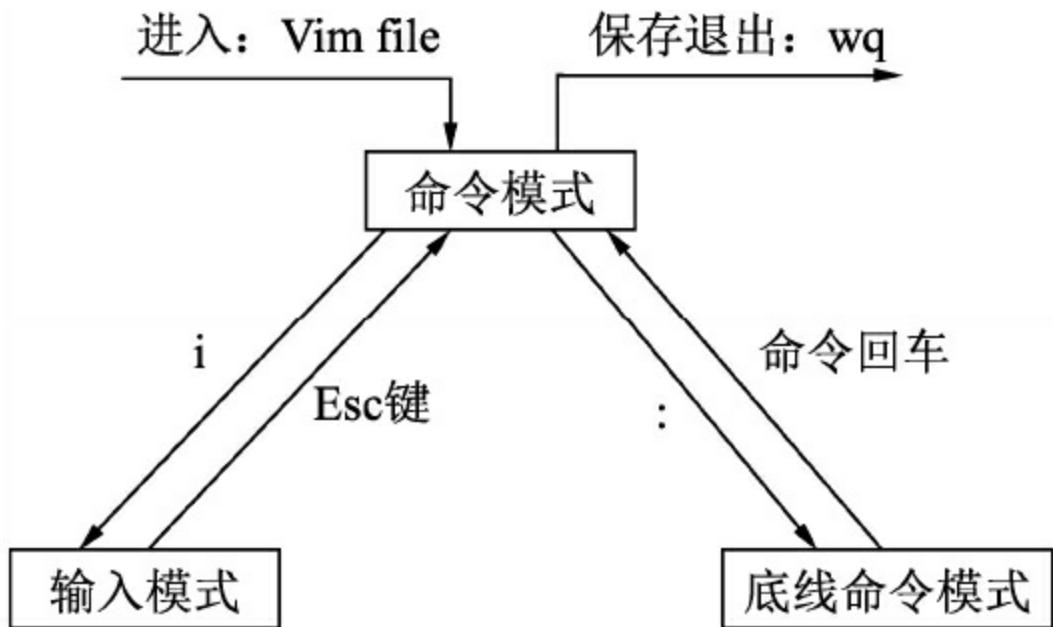


图1.22 Vim的3种模式切换方式

下面介绍一些常用的Vim命令。首先是从命令模式到输入模式的切换，为了快速锁定插入代码的位置，会有不同的需求，这里列举了常见的5种切换方式，如表1.1所示。

表1.1 Vim常用的插入方式

输入命令	含    义
i	在光标处插入
I	在光标所在的行首插入
A	在光标所在的行末插入
o	在光标所在行开辟下一行并插入
O	在光标所在行开辟上一行并插入

在Vim中，我们经常需要上下移动光标，浏览程序，这时就需要快速跳转的功能。这里列举几种常见的光标移动方式，如表1.2所示。

表1.2 Vim常用的光标移动方式

输入命令	含    义
\$	移动到行末
^	移动到行首
:n	移动到第n行，十分有用
Ctrl+y/e/d/u	下滚一行/上滚一行/下滚半页/上滚半页

当然，Vim对于复制、搜索、保存等方式也有方便的处理方法，这里统一放到了表1.3中。限于篇幅，Vim更丰富、有趣的功能，还有待读者自己去发现、探索。

表1.3 Vim其他常见的功能

输入命令	含    义
yy/nyy	复制光标一行/后续n行
p/P	粘贴到后一行/前一行
dd	删除整行
u/Ctrl+r	撤销/恢复
/word	搜索word这个词，继续输入n为搜索下一个，N为搜索上一个
w/w!	保存/强制保存
q/q!	退出/强制退出
:set nu	显示行号
:set unnu	不显示行号

### 3. Python调试器：pdb

对于开发人员来说，调试可以便于查看程序运行的过程，定位程序运行的错误，是一项十分重要的技能。此外，深度学习算法通常会使用远程服务器的形式进行开发，这时使用类似于Pycharm的IDE进行调试的方法也不适用了。所以，如果你还在继续使用print()函数来调试Python程序的话，你需要了解一个新的工具：pdb。

pdb是Python自带的一个库，可以提供交互式的调试功能，提供了

断点设置、单步调试、源码查看、堆栈查看等丰富的功能。pdb类似于C++中的gdb调试工具。通常有两种方式来使用pdb，第一种方式是在命令行中加入pdb模块来启动Python程序，如下：

```
python3 -m pdb test.py
```

这种方式会从程序第一行开始就进入交互环境，适用于较小的程序调试。在更大型的工程里，我们更习惯用插入断点的方式进行调试，如下：

```
import pdb  
pdb.set_trace()
```

将set\_trace()函数放到代码中的任何地方，执行程序时都会在此处产生一个断点，尤其是在使用PyTorch这种极度Python化的框架时，使用极为方便。pdb主要的调试命令如表1.4所示。

表1.4 pdb常用的调试命令

命 令	缩 写	含 义
next	n	执行下一行，但不会进入子程序中
step	s	执行下一行，会进入子程序中

(续)

命 令	缩 写	含 义
continue	c	继续执行到下一个断点
where	w/bt	打印堆栈的轨迹
return	r	执行到当前函数结束

#### 4. 网页可视化：Jupyter

Jupyter Notebook是一个基于Web应用的交互式笔记本，使用者可以

方便地在Web端与Python程序进行交互，以及进行数据的可视化与分析。Jupyter由IPython Notebook发展而来，现已支持40多种编程语言。在学习深度学习及物体检测算法中，使用Jupyter主要有两点好处。

·数据可视化：Jupyter的交互界面简洁优雅，并且对Python的多种可视化库提供了支持，如Matplotlib、Pillow、Pandas等，这对于训练数据的分析、模型的评测等都提供了极大的帮助。物体检测本身也是视觉任务，可视化是一个至关重要的部分。

·远程访问：为了使用更多的GPU资源，我们通常会将训练任务提交到服务器中，导致无法轻易地访问查看服务器数据。对于Jupyter而言，只需在服务器中开启Notebook服务，即可在本地的浏览器端进行访问，并且对于系统、环境都沒有限制。

使用pip工具可以轻松地安装Jupyter，代码如下：

---

```
# Python 2  
pip2 install Jupyter  
# Python 3  
pip3 install Jupyter
```

---

安装成功后，在需要操作的目录下输入jupyter notebook即可开启Jupyter服务。如果是在本地执行的服务，则可以在浏览器中输入<http://127.0.0.1:8000>，即可打开Jupyter界面，如图1.23所示。

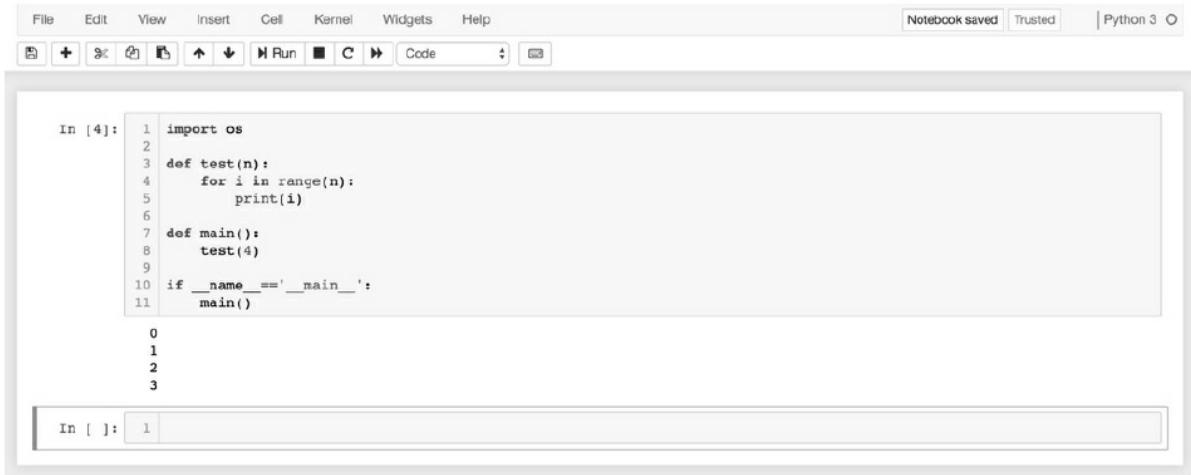


图1.23 Jupyter界面示例

Jupyter默认使用8000端口，如果是在远程服务器开启的服务，则浏览器中的IP也要相应地修改为服务器的IP。

Jupyter也可以设置访问密码，并且支持Markdown语法。Jupyter还有很多丰富有趣的功能，由于篇幅有限，其他功能留给读者自己去探索。

## 5. 分屏工具：Terminator

对于Ubuntu系统的使用者而言，在进行程序的开发、执行时经常需要打开多个终端界面，这时终端分屏的功能可以避免界面的来回切换。然而Ubuntu自带的终端Terminal并不支持分屏功能，因此推荐一个替代终端：Terminator，可以轻松地实现终端分屏和切换等操作，界面如图1.24所示。

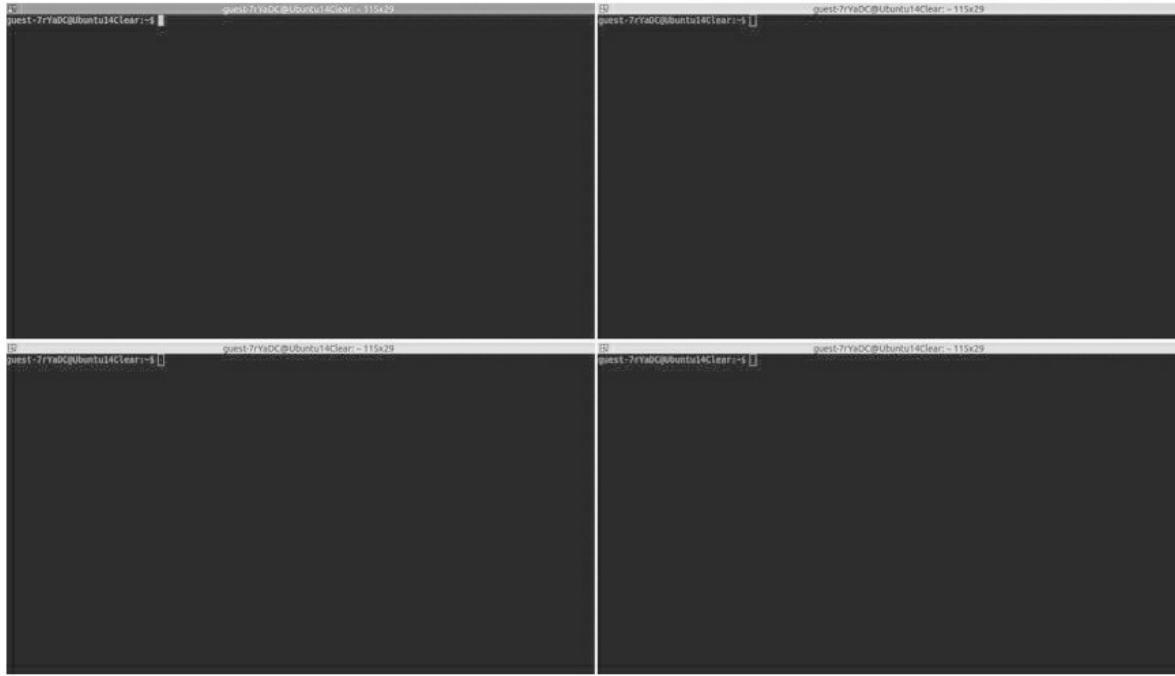


图1.24 Terminator界面示例

Terminator的安装极为方便，使用apt或者apt-get即可完成安装，命令如下：

```
sudo apt install terminator
```

安装完成后，按Ctrl+Shift+T键即可打开Terminator。Terminator常见的操作快捷键如表1.5所示。

表1.5 Terminator的常用操作

操作	含义
Ctrl+Shift+E	水平分割终端
Ctrl+Shift+O	竖直分割终端
Ctrl+Shift+W	关闭当前终端
Ctrl+Shift+Q	关闭所有终端

## 6. 任务托管：Screen

在远程服务器中，我们通常使用终端的脚本命令来执行模型的训练。训练通常会有几天时间，这时如果网络出现异常，或者有事需要关掉终端时，终端任务也会被杀掉，这显然不是我们想要看到的。因此，我们需要将任务托管到后台执行，这时Screen就派上用场了。

Screen是一款由GNU开发的软件，可用于多个命令行终端之间的自由切换与管理。即使网络断开，只要Screen本身没有停止，其内部执行的会话将一直保留。Screen软件可通过apt快速进行安装，如下：

---

```
sudo apt install screen
```

---

Screen软件的打开、关闭都十分简单，具体操作如表1.6所示。

表1.6 Screen的常用操作

操作	含义
screen -S name	新建一个名字为name的Screen窗口
Ctrl+a->Ctrl+d	关闭当前的Screen窗口
Ctrl+a->k->y	永久性删除当前的Screen窗口
screen -ls	列举当前所有的Screen窗口
screen -r name	回复名字为name的Screen窗口

## 1.5 总结

作为本书的开篇，本章从宏观的人工智能与物体检测发展讲起，介绍了人工智能领域各项技术的发展与作用。作为基础而又重要的技术分支，物体检测技术在人工智能中占有举足轻重的地位，笔者简要地讲述了其从无到有的发展过程与经典的解决算法。

为了更好地学习与研究物体检测，笔者引进了PyTorch这个优秀的动态图框架，并对比了当前其他优秀的框架，给出了选择PyTorch的原因。最后，为了使读者更好地学习物体检测，加快代码开发的效率，还介绍了如Jupyter、Screen等常用的开发工具，希望能对读者有所帮助。

在了解了本书所需的背景知识后，第2章将简明扼要地介绍PyTorch的基础知识，为后续卷积网络与物体检测的学习打下坚实的基础。

## 第2章 PyTorch基础

工欲善其事，必先利其器。掌握一个好的工具能够使学习的效率翻倍。而PyTorch作为一个优秀的动态图框架，简洁且入门快，适用于快速实现与各种算法的学习。

本章主要讲解PyTorch的基础知识，从基本数据、神经网络，再到常用的模型处理与数据处理方法，通过简单的示例将最常用的方法展示出来，力求让读者熟悉PyTorch处理神经网络的过程。由于篇幅有限，如果读者想了解更为细致的内容，可查阅PyTorch的官方文档。

## 2.1 基本数据：Tensor

Tensor，即张量，是PyTorch中的基本操作对象，可以看做是包含单一数据类型元素的多维矩阵。从使用角度来看，Tensor与NumPy的ndarrays非常类似，相互之间也可以自由转换，只不过Tensor还支持GPU的加速。

本节首先介绍Tensor的基本数据类型，然后讲解创建Tensor的多种方法，最后依次介绍Tensor在实际使用中的常见操作，如变形、组合、排序等。

## 2.1.1 Tensor数据类型

Tensor在使用时可以有不同的数据类型，如表2.1所示，官方给出了7种CPU Tensor类型与8种GPU Tensor类型，在使用时可以根据网络模型所需的精度与显存容量，合理地选取。16位半精度浮点是专为GPU上运行的模型设计的，以尽可能地节省GPU显存占用，但这种节省显存空间的方式也缩小了所能表达数据的大小。PyTorch中默认的数据类型是torch.FloatTensor，即torch.Tensor等同于torch.FloatTensor。

表2.1 Tensor数据类型

数 �据 类 型	CPU Tensor	GPU Tensor
32位浮点	torch.FloatTensor	torch.cuda.FloatTensor

(续)

数 据 类 型	CPU Tensor	GPU Tensor
64位浮点	torch.DoubleTensor	torch.cuda.DoubleTensor
16位半精度浮点	N/A	torch.cuda.HalfTensor
8位无符号整型	torch.ByteTensor	torch.cuda.ByteTensor
8位有符号整型	torch.CharTensor	torch.cuda.CharTensor
16位有符号整型	torch.ShortTensor	torch.cuda.ShortTensor
32位有符号整型	torch.IntTensor	torch.cuda.IntTensor
64位有符号整型	torch.LongTensor	torch.cuda.LongTensor

PyTorch可以通过set\_default\_tensor\_type函数设置默认使用的Tensor类型，在局部使用完后如果需要其他类型，则还需要重新设置回所需的类型。

```
torch.set_default_tensor_type('torch.DoubleTensor')
```

对于Tensor之间的类型转换，可以通过type(new\_type)、type\_as()、

int()等多种方式进行操作，尤其是type\_as()函数，在后续的模型学习中可以看到，我们想保持Tensor之间的类型一致，只需要使用type\_as()即可，并不需要明确具体是哪种类型。下面分别举例讲解这几种方法的使用方式。

---

```
# 创建新Tensor，默认类型为torch.FloatTensor
>>> a = torch.Tensor(2, 2)
>>> a
tensor(1.00000e-36 *
      [[-4.0315,  0.0000],
       [ 0.0700,  0.0000]])
# 使用int()、float()、double()等直接进行数据类型转换
>>> b = a.double()
>>> b
tensor(1.00000e-36 *
      [[-4.0315,  0.0000],
       [ 0.0700,  0.0000]], dtype=torch.float64)
# 使用type()函数
>>> c = a.type(torch.DoubleTensor)
>>> c
tensor(1.00000e-36 *
      [[-4.0315,  0.0000],
       [ 0.0700,  0.0000]], dtype=torch.float64)
# 使用type_as()函数
>>> d = a.type_as(b)
>>> d
tensor(1.00000e-36 *
      [[-4.0315,  0.0000],
       [ 0.0700,  0.0000]], dtype=torch.float64)
```

---

## 2.1.2 Tensor的创建与维度查看

Tensor有多种创建方法，如基础的构造函数Tensor()，还有多种与NumPy十分类似的方法，如ones()、eye()、zeros()和randn()等，图2.1列举了常见的Tensor创建方法。

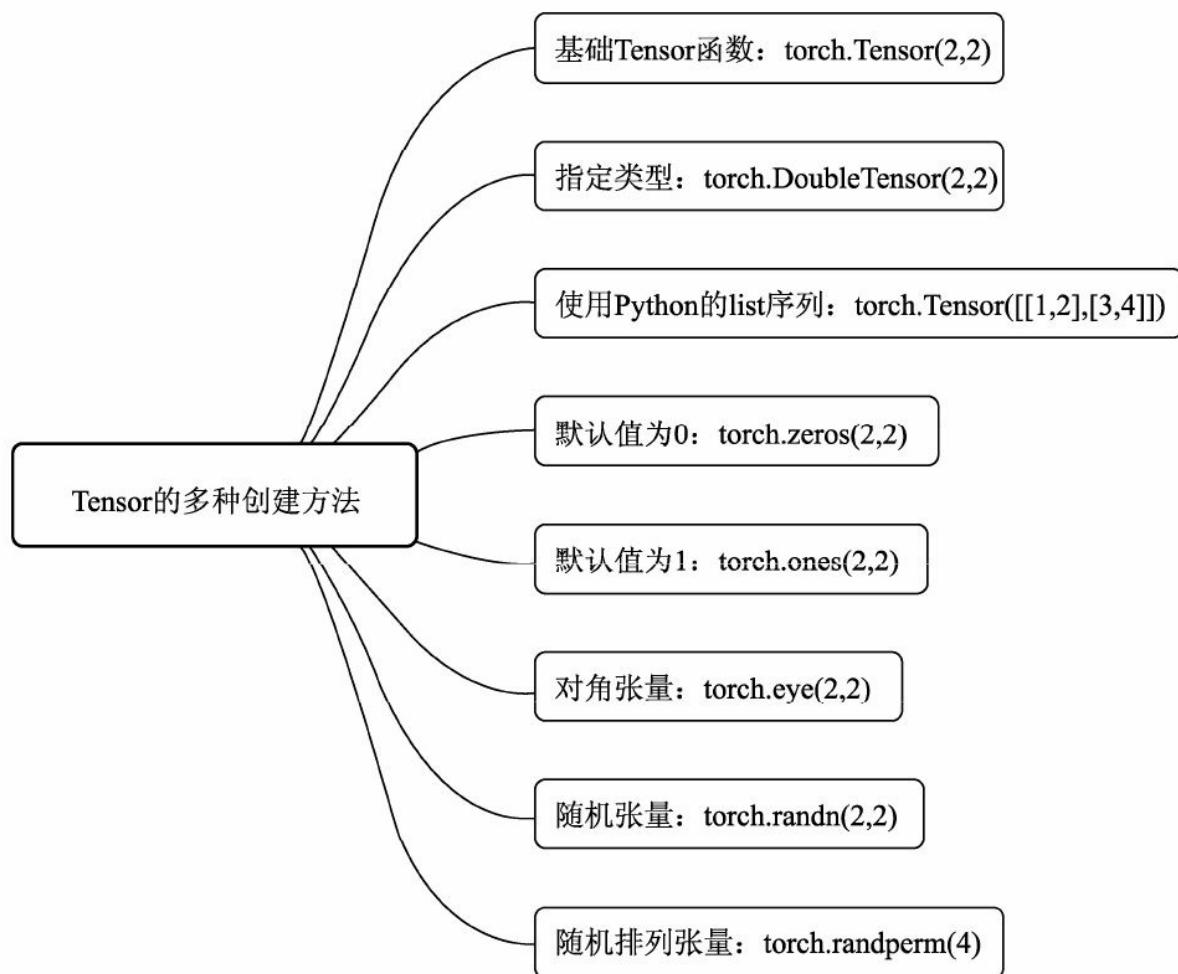


图2.1 Tensor的多种创建方法

下面从代码角度实现Tensor的多种方式创建。

```
# 最基础的Tensor()函数创建方法，参数为Tensor的每一维大小
>>> a=torch.Tensor(2, 2)
```

```
>>> a
tensor(1.00000e-18 *
      [[-8.2390,  0.0000],
       [ 0.0000,  0.0000]])
>>> b = torch.DoubleTensor(2,2)
>>> b
tensor(1.00000e-310 *
      [[ 0.0000,  0.0000],
       [ 6.9452,  0.0000]], dtype=torch.float64)
# 使用Python的list序列进行创建
>>> c = torch.Tensor([[1, 2], [3, 4]])
>>> c
tensor([[ 1.,  2.],
        [ 3.,  4.]])
# 使用zeros()函数，所有元素均为0
>>> d = torch.zeros(2, 2)
>>> d
tensor([[ 0.,  0.],
        [ 0.,  0.]])
# 使用ones()函数，所有元素均为1
>>> e = torch.ones(2, 2)
>>> e
tensor([[ 1.,  1.],
        [ 1.,  1.]])
# 使用eye()函数，对角线元素为1，不要求行列数相同，生成二维矩阵
>>> f = torch.eye(2, 2)
>>> f
tensor([[ 1.,  0.],
        [ 0.,  1.]])
# 使用randn()函数，生成随机数矩阵
>>> g = torch.randn(2, 2)
>>> g
tensor([-0.3979,  0.2728],
      [ 1.4558, -0.4451]])
# 使用arange(start, end, step)函数，表示从start到end，间距为step，一维向量
>>> h = torch.arange(1, 6, 2)
>>> h
tensor([ 1.,  3.,  5.])
# 使用linspace(start, end, steps)函数，表示从start到end，一共steps份，一维向量
>>> i = torch.linspace(1, 6, 2)
>>> i
tensor([ 1.,  6.])
# 使用randperm(num)函数，生成长度为num的随机排列向量
>>> j = torch.randperm(4)
>>> j
tensor([ 1,  2,  0,  3])
# PyTorch 0.4中增加了torch.tensor()方法，参数可以为Python的list、NumPy的ndarray等
>>> k = torch.tensor([1, 2, 3])
```

```
tensor([ 1,  2,  3])
```

---

对于Tensor的维度，可使用Tensor.shape或者size()函数查看每一维的大小，两者等价。

---

```
>>> a=torch.randn(2,2)
>>> a.shape                                     # 使用shape查看Tensor维度
torch.Size([2, 2])
>>> a.size()                                    # 使用size()函数查看Tensor维度
torch.Size([2, 2])
```

---

查看Tensor中的元素总个数，可使用Tensor.numel()或者Tensor.nelement()函数，两者等价。

---

```
# 查看Tensor中总的元素个数
>>> a.numel()
4
>>> a.nelement()
4
```

---

### 2.1.3 Tensor的组合与分块

组合与分块是将Tensor相互叠加或者分开，是十分常用的两个功能，PyTorch提供了多种操作函数，如图2.2所示。

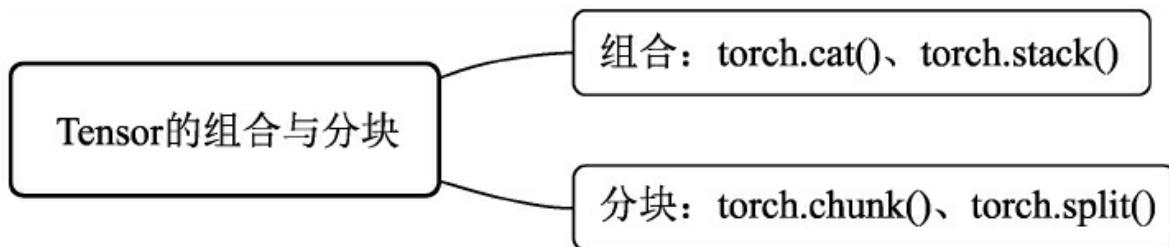


图2.2 Tensor的组合与分块操作

组合操作是指将不同的Tensor叠加起来，主要有`torch.cat()`和`torch.stack()`两个函数。`cat`即concatenate的意思，是指沿着已有的数据的某一维度进行拼接，操作后数据的总维数不变，在进行拼接时，除了拼接的维度之外，其他维度必须相同。而`torch.stack()`函数指新增维度，并按照指定的维度进行叠加，具体示例如下：

---

```
# 创建两个2×2的Tensor
>>> a=torch.Tensor([[1,2],[3,4]])
>>> a
tensor([[ 1.,  2.],
        [ 3.,  4.]])
>>> b = torch.Tensor([[5,6], [7,8]])
>>> b
tensor([[ 5.,  6.],
        [ 7.,  8.]])
# 以第一维进行拼接，则变成4×2的矩阵
>>> torch.cat([a,b], 0)
tensor([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.],
        [ 7.,  8.]])
# 以第二维进行拼接，则变成24的矩阵
```

```

>>> torch.cat([a,b], 1)
tensor([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
# 以第0维进行stack，叠加的基本单位为序列本身，即a与b，因此输出[a, b]，输出维度为2×2×2
>>> torch.stack([a,b], 0)
tensor([[[ 1.,  2.],
         [ 3.,  4.]],
        [[ 5.,  6.],
         [ 7.,  8.]]])
# 以第1维进行stack，叠加的基本单位为每一行，输出维度为2×2×2
>>> torch.stack([a,b], 1)
tensor([[[ 1.,  2.],
         [ 5.,  6.]],
        [[ 3.,  4.],
         [ 7.,  8.]]])
# 以第2维进行stack，叠加的基本单位为每一行的每一个元素，输出维度为2×2×2
>>> torch.stack([a,b], 2)
tensor([[[ 1.,  5.],
         [ 2.,  6.]],
        [[ 3.,  7.],
         [ 4.,  8.]]])

```

---

分块则是与组合相反的操作，指将Tensor分割成不同的子Tensor，主要有`torch.chunk()`与`torch.split()`两个函数，前者需要指定分块的数量，而后者则需要指定每一块的大小，以整型或者list来表示。具体示例如下：

---

```

>>> a=torch.Tensor([[1,2,3],[4,5,6]])
>>> a
tensor([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
# 使用chunk，沿着第0维进行分块，一共分两块，因此分割成两个1×3的Tensor
>>> torch.chunk(a, 2, 0)
(tensor([[ 1.,  2.,  3.]]), tensor([[ 4.,  5.,  6.]]))
# 沿着第1维进行分块，因此分割成两个Tensor，当不能整除时，最后一个的维数会小于前面的
# 因此第一个Tensor为2×2，第二个为2×1
>>> torch.chunk(a, 2, 1)
(tensor([[ 1.,  2.],
         [ 4.,  5.]]), tensor([[ 3.],
         [ 6.]]))
# 使用split，沿着第0维分块，每一块维度为2，由于第一维维度总共为2，因此相当于没有分割
>>> torch.split(a, 2, 0)
(tensor([[ 1.,  2.,  3.]]),

```

```
[ 4.,  5.,  6.]])),)  
# 沿着第1维分块，每一块维度为2，因此第一个Tensor为2×2，第二个为2×1  
>>> torch.split(a, 2, 1)  
(tensor([[ 1.,  2.],  
        [ 4.,  5.]]), tensor([[ 3.],  
        [ 6.]]))  
# split也可以根据输入的list进行自动分块，list中的元素代表了每一个块占的维度  
>>> torch.split(a, [1,2], 1)  
(tensor([[ 1.],  
        [ 4.]]), tensor([[ 2.,  3.],  
        [ 5.,  6.]]))
```

---

## 2.1.4 Tensor的索引与变形

索引操作与NumPy非常类似，主要包含下标索引、表达式索引、使用`torch.where()`与`Tensor.clamp()`的选择性索引。

```
>>> a = torch.Tensor([[0,1], [2, 3]])
>>> a
tensor([[ 0.,  1.],
        [ 2.,  3.]])
# 根据下标进行索引
>>> a[1]
tensor([ 2.,  3.])
>>> a[0,1]
tensor(1.)
# 选择a中大于0的元素，返回和a相同大小的Tensor，符合条件的置1，否则置0
>>> a>0
tensor([[ 0,  1],
        [ 1,  1]], dtype=torch.uint8)
# 选择符合条件的元素并返回，等价于torch.masked_select(a, a>0)
>>> a[a>0]
tensor([ 1.,  2.,  3.])
# 选择非0元素的坐标，并返回
>>> torch.nonzero(a)
tensor([[ 0,  1],
        [ 1,  0],
        [ 1,  1]])
# torch.where(condition, x, y)，满足condition的位置输出x，否则输出y
>>> torch.where(a>1, torch.full_like(a, 1), a)
tensor([[ 0.,  1.],
        [ 1,  1.]])
# 对Tensor元素进行限制可以使用clamp()函数，示例如下，限制最小值为1，最大值为2
>>> a.clamp(1,2)
tensor([[ 1.,  1.],
        [ 2.,  2.]])
```

变形操作则是指改变Tensor的维度，以适应在深度学习的计算中，数据维度经常变换的需求，是一种十分重要的操作。在PyTorch中主要有4类不同的变形方法，如表2.2所示。

表2.2 PyTorch常用的变形操作

变 形 操 作	功 能
view()、 resize()、 reshape()	调整Tensor的形状，元素总数相同
transpose()、 permute()	各维度之间的变换
squeeze()、 unsqueeze()	处理size为1的维度
expand()、 expand_as()	复制元素来扩展维度

### 1. view()、 resize()和reshape()函数

view()、 resize()和reshape()函数可以在不改变Tensor数据的前提下任意改变Tensor的形状，必须保证调整前后的元素总数相同，并且调整前后共享内存，三者的作用基本相同。

```
>>> a=torch.arange(1,5)
>>> a
tensor([ 1.,  2.,  3.,  4.])
# 分别使用view()、 resize()及reshape()函数进行维度变换
>>> b=a.view(2,2)
>>> b
tensor([[ 1.,  2.],
        [ 3.,  4.]])
>>> c=a.resize(4,1)
>>> c
tensor([[ 1.],
        [ 0.],
        [ 0.],
        [ 4.]])
>>> d=a.reshape(4,1)
>>> d
tensor([[ 1.],
        [ 0.],
        [ 0.],
        [ 4.]])
# 改变了b、c、d的一个元素，a也跟着改变了，说明两者共享内存
>>> b[0,0]=0
>>> c[1,0]=0
>>> d[2,0]=0
>>> a
tensor([ 0.,  0.,  0.,  4.])
```

如果想要直接改变Tensor的尺寸，可以使用`resize_(0)`的原地操作函数。在`resize_(0)`函数中，如果超过了原Tensor的大小则重新分配内存，多出部分置0，如果小于原Tensor大小则剩余的部分仍然会隐藏保留。

---

```
>>> c=a.resize_(2,3)
>>> c
tensor([[ 0.0000,  2.0000,  3.0000],
        [ 4.0000,  0.0000,  0.0000]])
# 发现操作后a也跟着改变了
>>> a
tensor([[ 0.0000,  2.0000,  3.0000],
        [ 4.0000,  0.0000,  0.0000]])
```

---

## 2. transpose()和permute()函数

`transpose()`函数可以将指定的两个维度的元素进行转置，而`permute()`函数则可以按照给定的维度进行维度变换。

---

```
>>> a=torch.randn(2,2,2)
>>> a
tensor([[[[-0.9268,  0.6006],
           [ 1.0213,  0.5328]],
          [[-0.7024,  0.7978],
           [ 1.0553, -0.6524]]]])
# 将第0维和第1维的元素进行转置
>>> a.transpose(0,1)
tensor([[[[-0.9268,  0.6006],
           [-0.7024,  0.7978]],
          [[ 1.0213,  0.5328],
           [ 1.0553, -0.6524]]]])
# 按照第2、1、0的维度顺序重新进行元素排列
>>> a.permute(2,1,0)
tensor([[[[-0.9268, -0.7024],
           [ 1.0213,  1.0553]],
          [[ 0.6006,  0.7978],
           [ 0.5328, -0.6524]]]])
```

---

## 3. squeeze()和unsqueeze()函数

在实际的应用中，经常需要增加或减少Tensor的维度，尤其是维度为1的情况，这时候可以使用squeeze()与unsqueeze()函数，前者用于去除size为1的维度，而后者则是将指定的维度的size变为1。

---

```
>>> a=torch.arange(1,4)
>>> a.shape
torch.Size([3])
# 将第0维变为1，因此总的维度为1、3
>>> a.unsqueeze(0).shape
torch.Size([1, 3])
# 第0维如果是1，则去掉该维度，如果不是1则不起任何作用
>>> a.unsqueeze(0).squeeze(0).shape
torch.Size([3])
```

---

#### 4. expand()和expand\_as()函数

有时需要采用复制元素的形式来扩展Tensor的维度，这时expand就派上用场了。expand()函数将size为1的维度复制扩展为指定大小，也可以使用expand\_as()函数指定为示例Tensor的维度。

---

```
>>> a=torch.randn(2,2,1)
>>> a
tensor([[[ 0.5379],
         [-0.6294]],
        [[ 0.7006],
         [ 1.2900]]])
# 将第2维的维度由1变为3，则复制该维的元素，并扩展为3
>>> a.expand(2,2,3)
tensor([[[ 0.5379,  0.5379,  0.5379],
         [-0.6294, -0.6294, -0.6294]],
        [[ 0.7006,  0.7006,  0.7006],
         [ 1.2900,  1.2900,  1.2900]]])
```

---

 **注意：**在进行Tensor操作时，有些操作如transpose()、permute()等可能会把Tensor在内存中变得不连续，而有些操作如view()等是需要Tensor内存连续的，这种情况下需要使用contiguous()操作先将内存变为

连续的。在PyTorch v0.4版本中增加了`reshape()`操作，可以看做是`Tensor.contiguous().view()`。

## 2.1.5 Tensor的排序与取极值

比较重要的是排序函数sort()，选择沿着指定维度进行排序，返回排序后的Tensor及对应的索引位置。max()与min()函数则是沿着指定维度选择最大与最小元素，返回该元素及对应的索引位置。

```
>>> a=torch.randn(3,3)
>>> a
tensor([[ 1.0179, -1.4278, -0.0456],
        [-1.1668,  0.4531, -1.5196],
        [-0.1498, -0.2556, -1.4915]])
# 按照第0维即按行排序，每一列进行比较，True代表降序，False代表升序
>>> a.sort(0, True)[0]
tensor([[ 1.0179,  0.4531, -0.0456],
        [-0.1498, -0.2556, -1.4915],
        [-1.1668, -1.4278, -1.5196]])
>>> a.sort(0, True)[1]
tensor([[ 0,  1,  0],
        [ 2,  2,  2],
        [ 1,  0,  1]])
# 按照第0维即按行选取最大值，即将每一列的最大值选取出来
>>> a.max(0)
(tensor([ 1.0179,  0.4531, -0.0456]), tensor([ 0,  1,  0]))
```

对于Tensor的单元素数学运算，如abs()、sqrt()、log()、pow()和三角函数等，都是逐元素操作（element-wise），输出的Tensor形状与原始Tensor形状一致。

对于类似求和、求均值、求方差、求距离等需要多个元素完成的操作，往往需要沿着某个维度进行计算，在Tensor中属于归并操作，输出形状小于输入形状。由于比较简单且与NumPy极为相似，在此就不详细展开。

## 2.1.6 Tensor的自动广播机制与向量化

PyTorch在0.2版本以后，推出了自动广播语义，即不同形状的Tensor进行计算时，可自动扩展到较大的相同形状，再进行计算。广播机制的前提是任一个Tensor至少有一个维度，且从尾部遍历Tensor维度时，两者维度必须相等，其中一个要么是1要么不存在。

---

```
>>> a=torch.ones(3,1,2)
>>> b=torch.ones(2,1)
# 从尾部遍历维度，1对应2，2对应1，3对应不存在，因此满足广播条件，最后求和后的维度为[3,2,2]
>>> (a+b).size()
torch.Size([3, 2, 2])
>>> c=torch.ones(2,3)
# a与c最后一维的维度为2对应3，不满足广播条件，因此报错
>>> (a+c).size()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
RuntimeError: The size of tensor a (2) must match the size of tensor b (3)
at non-singleton dimension 2
```

---

向量化操作是指可以在同一时间进行批量地并行计算，例如矩阵运算，以达到更好的计算效率的一种方式。在实际使用时，应尽量使用向量化直接对Tensor操作，避免低效率的for循环对元素逐个操作，尤其是在训练网络模型时，如果有大量的for循环，会极大地影响训练的速度。

## 2.1.7 Tensor的内存共享

为了实现高效计算，PyTorch提供了一些原地操作运算，即in-place operation，不经过复制，直接在原来的内存上进行计算。对于内存的共享，主要有如下3种情况，如图2.3所示。

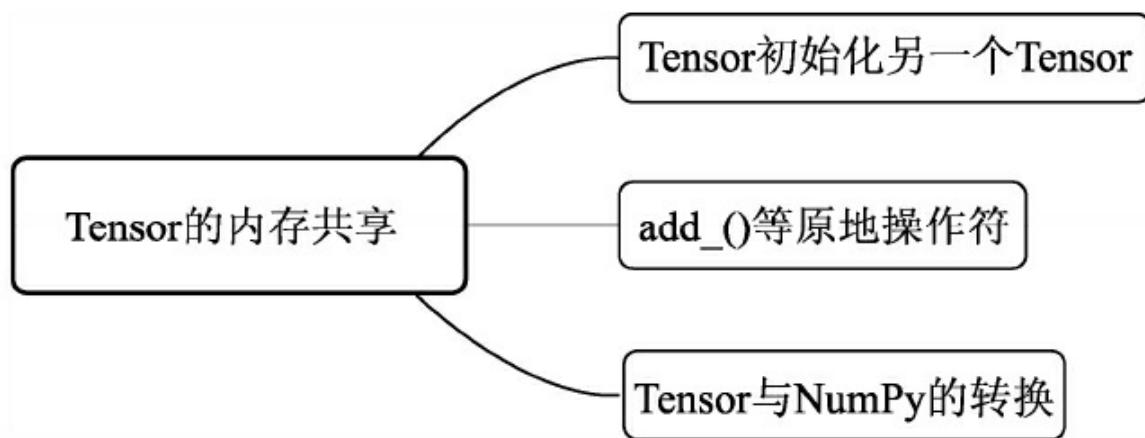


图2.3 Tensor的内存共享

### 1. 通过Tensor初始化Tensor

直接通过Tensor来初始化另一个Tensor，或者通过Tensor的组合、分块、索引、变形操作来初始化另一个Tensor，则这两个Tensor共享内存。

---

```
>>> a=torch.randn(2,2)
>>> a
tensor([[ 0.0666, -0.3389],
       [ 0.8224,  0.6678]])
# 用a初始化b，或者用a的变形操作初始化c，这三者共享内存，一个变，其余的也改变了
>>> b=a
>>> c=a.view(4)
>>> b[0,0]=0
>>> c[3]=4
>>> a
```

```
tensor([[ 0.0000, -0.3389],
       [ 0.8224,  4.0000]])
```

---

## 2. 原地操作符

PyTorch对于一些操作通过加后缀“\_”实现了原地操作，如add\_()和resize\_()等，这种操作只要被执行，本身的Tensor则会被改变。

---

```
>>> a=torch.Tensor([[1,2],[3,4]])
# add_()函数使得a也改变了
>>> b=a.add_(a)
>>> a
tensor([[ 2.,  4.],
       [ 6.,  8.]])
# resize_()函数使得a也发生了改变
>>> c=a.resize_(4)
>>> a
tensor([ 2.,  4.,  6.,  8.])
```

---

## 3. Tensor与NumPy转换

Tensor与NumPy可以高效地进行转换，并且转换前后的变量共享内存。在进行PyTorch不支持的操作时，甚至可以曲线救国，将Tensor转换为NumPy类型，操作后再转为Tensor。

---

```
>>> a=torch.randn(1,2)
>>> a
tensor([-0.3228,  1.2726])
# Tensor转为NumPy
>>> b=a.numpy()
>>> b
array([-0.32281783,  1.2725701 ], dtype=float32)
# NumPy转为Tensor
>>> c=torch.from_numpy(b)
>>> c
tensor([-0.3228,  1.2726])
#Tensor转为list
>>> d=a.tolist()
>>> d
```

$[-0.3228178322315216, 1.2725701332092285]$

---

## 2.2 Autograd与计算图

基本数据Tensor可以保证完成前向传播，想要完成神经网络的训练，接下来还需要进行反向传播与梯度更新，而PyTorch提供了自动求导机制autograd，将前向传播的计算记录成计算图，自动完成求导。

在PyTorch 0.4版本之前，Tensor仅仅是对多维数组的抽象，使用自动求导机制需要将Tensor封装成`torch.autograd.Variable`类型，才能构建计算图。PyTorch 0.4版本则将Tensor与Variable进行了整合，以前Variable的使用情景都可以直接使用Tensor，变得更简单实用。

本节首先介绍Tensor的自动求导属性，然后对计算图进行简要的讲解。

## 2.2.1 Tensor的自动求导：Autograd

自动求导机制记录了Tensor的操作，以便自动求导与反向传播。可以通过`requires_grad`参数来创建支持自动求导机制的Tensor。

```
>>> import torch  
>>> a = torch.randn(2, 2, requires_grad=True)
```

`require_grad`参数表示是否需要对该Tensor进行求导，默认为`False`；设置为`True`则需要求导，并且依赖于该Tensor的之后的所有节点都需要求导。值得注意的是，在PyTorch 0.4对于Tensor的自动求导中，`volatile`参数已经被其他`torch.no_grad()`等函数取代了。

Tensor有两个重要的属性，分别记录了该Tensor的梯度与经历的操作。

·`grad`: 该Tensor对应的梯度，类型为Tensor，并与Tensor同维度。

·`grad_fn`: 指向function对象，即该Tensor经过了什么样的操作，用作反向传播的梯度计算，如果该Tensor由用户自己创建，则该`grad_fn`为`None`。

具体的参数使用示例如下：

```
>>> import torch  
>>> a=torch.randn(2, 2, requires_grad=True)  
>>> b=torch.randn(2, 2)  
# 可以看到默认的Tensor是不需要求导的，设置requires_grad为True后则需要求导  
>>> a.requires_grad, b.requires_grad  
True, False  
# 也可以通过内置函数requires_grad_()将Tensor变为需要求导  
>>> b.requires_grad_()
```

```
tensor([[ 0.0260, -0.1183],
       [-1.0907,  0.8107]])
>>> b.requires_grad
True
# 通过计算生成的Tensor, 由于依赖的Tensor需要求导, 因此c也需要求导
>>> c = a + b
>>> c.requires_grad
True
# a与b是自己创建的, grad_fn为None, 而c的grad_fn则是一个Add函数操作
>>> a.grad_fn, b.grad_fn, c.grad_fn
(None, None, <AddBackward1 object at 0x7fa7a53e04a8>)
>>> d = c.detach()
>>> d.requires_grad
False
```

---

⚠ 注意：早些版本使用.data属性来获取数据，PyTorch 0.4中建议使用Tensor.detach()函数，因为.data属性在某些情况下不安全，原因在于对.data生成的数据进行修改不会被autograd追踪。Tensor.detach()函数生成的数据默认requires\_grad为False。

## 2.2.2 计算图

计算图是PyTorch对于神经网络的具体实现形式，包括每一个数据Tensor及Tensor之间的函数function。在此我们以 $z = wx + b$ 为例，通常在神经网络中， $x$ 为输入， $w$ 与 $b$ 为网络需要学习的参数， $z$ 为输出，在这一层，计算图构建方法如图2.4所示。

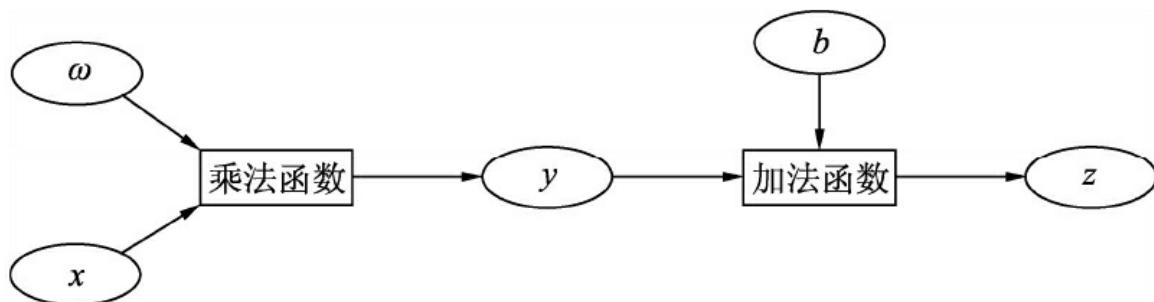


图2.4 计算图过程

在图2.4中， $x$ 、 $\omega$ 和 $b$ 都是用户自己创建的，因此都为叶节点， $\omega x$ 首先经过乘法算子产生中间节点 $y$ ，然后与 $b$ 经过加法算法产生最终输出 $z$ ，并作为根节点。

Autograd的基本原理是随着每一步Tensor的计算操作，逐渐生成计算图，并将操作的function记录在Tensor的grad\_fn中。在前向计算完后，只需对根节点进行backward函数操作，即可从当前根节点自动进行反向传播与梯度计算，从而得到每一个叶子节点的梯度，梯度计算遵循链式求导法则。

---

```
>>> import torch
# 生成3个Tensor变量，并作为叶节点
>>> x = torch.randn(1)
>>> w = torch.ones(1, requires_grad=True)
>>> b = torch.ones(1, requires_grad=True)
```

```
# 自己生成的，因此都为叶节点
>>> x.is_leaf, w.is_leaf, b.is_leaf
(True, True, True)
# 默认是不需求求导，关键字赋值为True后则需求求导
>>> x.requires_grad, w.requires_grad, b.requires_grad
(False, True, True)
# 进行前向计算，由计算生成的变量都不是叶节点
>>> y=w*x
>>> z=y+b
>>> y.is_leaf, z.is_leaf
(False, False)
# 由于依赖的变量有需求求导的，因此y与z都需求求导
>>> y.requires_grad, z.requires_grad
(True, True)
# grad_fn记录生成该变量经过了什么操作，如y是Mul，z是Add
>>> y.grad_fn
<MulBackward1 object at 0x7f4d4b49e208>
>>> z.grad_fn
<AddBackward1 object at 0x7f4d4b49e0f0>
# 对根节点调用backward()函数，进行梯度反传
>>> z.backward(retain_graph=True)
>>> w.grad
tensor([-2.2474])
>>> b.grad
tensor([ 1.])
```

---

### 2.2.3 Autograd注意事项

PyTorch的Autograd机制使得其可以灵活地进行前向传播与梯度计算，在实际使用时，需要注意以下3点，如图2.5所示。

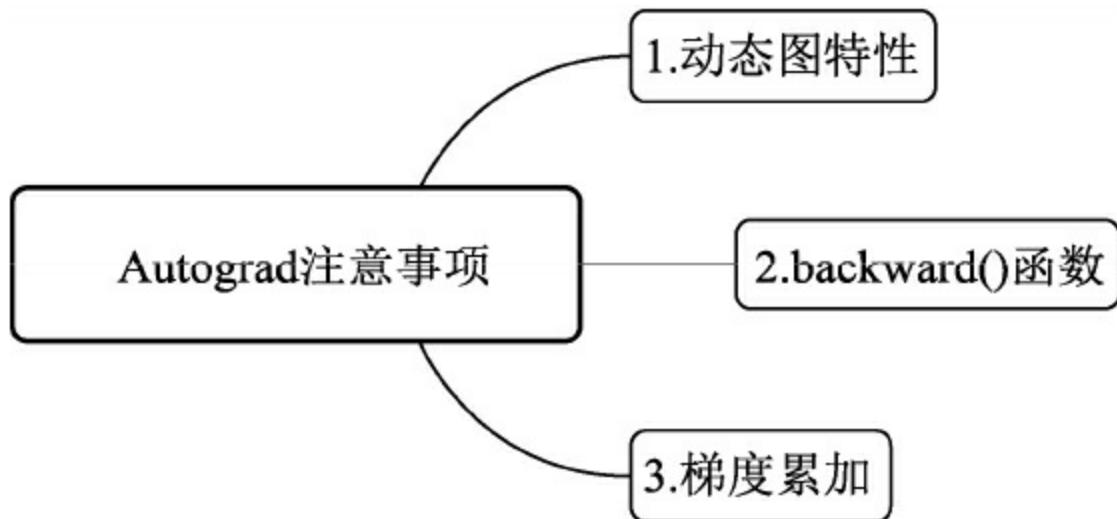


图2.5 Autograd使用注意事项

**动态图特性：** PyTorch建立的计算图是动态的，这也是PyTorch的一大特点。动态图是指程序运行时，每次前向传播时从头开始构建计算图，这样不同的前向传播就可以有不同的计算图，也可以在前向时插入各种Python的控制语句，不需要事先把所有的图都构建出来，并且可以很方便地查看中间过程变量。

`backward()`函数还有一个需要传入的参数`grad_variables`，其代表了根节点的导数，也可以看做根节点各部分的权重系数。因为PyTorch不允许Tensor对Tensor求导，求导时都是标量对于Tensor进行求导，因此，如果根节点是向量，则应配以对应大小的权重，并求和得到标量，再反传。如果根节点的值是标量，则该参数可以省略，默认为1。

当有多个输出需要同时进行梯度反传时，需要将retain\_graph设置为True，从而保证在计算多个输出的梯度时互不影响。

## 2.3 神经网络工具箱torch.nn

torch.autograd库虽然实现了自动求导与梯度反向传播，但如果我们要完成一个模型的训练，仍需要手写参数的自动更新、训练过程的控制等，还是不够便利。为此，PyTorch进一步提供了集成度更高的模块化接口torch.nn，该接口构建于Autograd之上，提供了网络模组、优化器和初始化策略等一系列功能。

### 2.3.1 nn.Module类

nn.Module是PyTorch提供的神经网络类，并在类中实现了网络各层的定义及前向计算与反向传播机制。在实际使用时，如果想要实现某个神经网络，只需继承nn.Module，在初始化中定义模型结构与参数，在函数forward()中编写网络前向过程即可。

下面具体以一个由两个全连接层组成的感知机为例，介绍如何使用nn.Module构造模块化的神经网络。新建一个perception.py文件，内容如下：

---

```
import torch
from torch import nn
# 首先建立一个全连接的子module，继承nn.Module
class Linear(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(Linear, self).__init__()      # 调用nn.Module的构造函数
        # 使用nn.Parameter来构造需要学习的参数
        self.w = nn.Parameter(torch.randn(in_dim, out_dim))
        self.b = nn.Parameter(torch.randn(out_dim))
    # 在forward中实现前向传播过程
    def forward(self, x):
        x = x.matmul(self.w)            # 使用Tensor.matmul实现矩阵相乘
        y = x + self.b.expand_as(x)     # 使用Tensor.expand_as()来保证矩阵
                                         # 形状一致
        return y
# 构建感知机类，继承nn.Module，并调用了Linear的子module
class Perception(nn.Module):
    def __init__(self, in_dim, hid_dim, out_dim):
        super(Perception, self).__init__()
        self.layer1 = Linear(in_dim, hid_dim)
        self.layer2 = Linear(hid_dim, out_dim)
    def forward(self, x):
        x = self.layer1(x)
        y = torch.sigmoid(x)           # 使用torch中的sigmoid作为激活函数
        y = self.layer2(y)
        y = torch.sigmoid(y)
        return y
```

---

编写完网络模型模块后，在终端中可以使用如下方法调用该模块。

```
>>> import torch
>>> from perception import Perception # 调用上述模块
# 实例化一个网络，并赋值全连接中的维数，最终输出二维代表了二分类
>>> perception = Perception(2,3,2)
# 可以看到perception中包含上述定义的layer1与layer2
>>> perception
Perception(
    (layer1): Linear()
    (layer2): Linear()
)
# named_parameters()可以返回学习参数的迭代器，分别为参数名与参数值
>>> for name, parameter in perception.named_parameters():
...     print(name, parameter)
...
layer1.w Parameter containing:
tensor([[ 0.1265, -0.6858,  0.0637],
       [ 0.5424, -0.2596, -2.1362]])
layer1.b Parameter containing:
tensor([-0.1427,  1.4034,  0.1175])
layer2.w Parameter containing:
tensor([[ 0.2575, -3.6569],
       [ 0.3657, -1.2370],
       [ 0.7178, -0.9569]])
layer2.b Parameter containing:
tensor([ 0.2041, -0.2558])
# 随机生成数据，注意这里的4代表了样本数为4，每个样本有两维
>>> data = torch.randn(4,2)
>>> data
tensor([[ 0.1399, -0.6214],
       [ 0.1325, -1.6260],
       [ 0.0035, -1.0567],
       [-0.6020, -0.9674]])
# 将输入数据传入perception，perception()相当于调用perception中的forward()函数
>>> output = perception(data)
>>> output
tensor([[ 0.7654,  0.0308],
       [ 0.7829,  0.0386],
       [ 0.7779,  0.0331],
       [ 0.7781,  0.0326]])
```

可以看到，利用nn.Module搭建神经网络简单易实现，同时较为规范。在实际使用时，应注意如下5点。

## 1. nn.Parameter函数

在类的`__init__()`中需要定义网络学习的参数，在此使用`nn.Parameter()`函数定义了全连接中的 $\omega$ 和 $b$ ，这是一种特殊的Tensor的构造方法，默认需要求导，即`requires_grad`为True。

## 2. forward()函数与反向传播

`forward()`函数用来进行网络的前向传播，并需要传入相应的Tensor，例如上例的`perception(data)`即是直接调用了`forward()`。在具体底层实现中，`perception.__call__(data)`将类的实例`perception`变成了可调用对象`perception(data)`，而在`perception.__call__(data)`中主要调用了`forward()`函数，具体可参考官方代码。

`nn.Module`可以自动利用Autograd机制实现反向传播，不需要自己手动实现。

## 3. 多个Module的嵌套

在Module的搭建时，可以嵌套包含子Module，上例的`Perception`中调用了`Linear`这个类，这样的代码分布可以使网络更加模块化，提升代码的复用性。在实际的应用中，PyTorch也提供了绝大多数的网络层，如全连接、卷积网络中的卷积、池化等，并自动实现前向与反向传播。在后面的章节中会对比较重要的层进行讲解。

## 4. nn.Module与nn.functional库

在PyTorch中，还有一个库为`nn.functional`，同样也提供了很多网络层与函数功能，但与`nn.Module`不同的是，利用`nn.functional`定义的网络层不可自动学习参数，还需要使用`nn.Parameter`封装。`nn.functional`的设计

计初衷是对于一些不需要学习参数的层，如激活层、BN（Batch Normalization）层，可以使用nn.functional，这样这些层就不需要在nn.Module中定义了。

总体来看，对于需要学习参数的层，最好使用nn.Module，对于无参数学习的层，可以使用nn.functional，当然这两者间并没有严格的好坏之分。

## 5. nn.Sequential()模块

当模型中只是简单的前馈网络时，即上一层的输出直接作为下一层的输入，这时可以采用nn.Sequential()模块来快速搭建模型，而不必手动在forward()函数中一层一层地前向传播。因此，如果想快速搭建模型而不考虑中间过程的话，推荐使用nn.Sequential()模块。

在上面的例子中，Perception类中的layer1与layer2是直接传递的，因此该Perception类可以使用nn.Sequential()快速搭建。在此新建一个perception\_sequential.py文件，内容如下：

---

```
from torch import nn
class Perception(nn.Module):
    def __init__(self, in_dim, hid_dim, out_dim):
        super(Perception, self).__init__()
        # 利用nn.Sequential()快速搭建网络模块
        self.layer = nn.Sequential(
            nn.Linear(in_dim, hid_dim),
            nn.Sigmoid(),
            nn.Linear(hid_dim, out_dim),
            nn.Sigmoid()
        )
    def forward(self, x):
        y = self.layer(x)
        return y
```

---

在终端中进入上述perception\_sequential.py文件的同级目录下，输入

python3进入交互环境，使用如下指令即可调用该网络结构。

---

```
>>> import torch # 引入torch模块
# 从上述文件中引入Perception类
>>> from perception_sequential import Perception
>>> model = Perception(100, 1000, 10).cuda() # 构建类的实例，并表明在CUDA上
# 打印model结构，会显示Sequential中每一层的具体参数配置
>>> model
Perception(
  (layer): Sequential(
    (0): Linear(in_features=100, out_features=1000, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=1000, out_features=10, bias=True)
    (3): Sigmoid()
  )
)
>>> input = torch.randn(100).cuda()
>>> output = model(input) # 将输入传入实例化的模型
>>> output.shape
torch.Size([10])
```

---

## 2.3.2 损失函数

在深度学习中，损失反映模型最后预测结果与实际真值之间的差距，可以用来分析训练过程的好坏、模型是否收敛等，例如均方损失、交叉熵损失等。在PyTorch中，损失函数可以看做是网络的某一层而放到模型定义中，但在实际使用时更偏向于作为功能函数而放到前向传播过程中。

PyTorch在torch.nn及torch.nn.functional中都提供了各种损失函数，通常来讲，由于损失函数不含有可学习的参数，因此这两者在功能上基本没有区别。

---

```
# 接着2.3.1节中的终端环境继续运行，来进一步求损失
>>> from torch import nn
>>> import torch.nn.functional as F
# 设置标签，由于是二分类，一共有4个样本，因此标签维度为14，每个数为0或1两个类别
>>> label = torch.Tensor([0, 1, 1, 0]).long()
# 实例化nn中的交叉熵损失类
>>> criterion = nn.CrossEntropyLoss()
# 调用交叉熵损失
>>> loss_nn = criterion(output, label)
>>> loss_nn
tensor(0.7616)
# 由于F.cross_entropy是一个函数，因此可以直接调用，不需要实例化，两者求得的损失值相同
>>> loss_functional = F.cross_entropy(output, label)
>>> loss_loss_functional
tensor(0.7616)
```

---

### 2.3.3 优化器nn.optim

在上述介绍中，`nn.Module`模块提供了网络骨架，`nn.functional`提供了各式各样的损失函数，而`Autograd`又自动实现了求导与反向传播机制，这时还缺少一个如何进行模型优化、加速收敛的模块，`nn.optim`应运而生。

`nn.optim`中包含了各种常见的优化算法，包括随机梯度下降算法SGD（Stochastic Gradient Descent，随机梯度下降）、Adam（Adaptive Moment Estimation）、Adagrad、RMSProp，这里仅对常用的SGD与Adam两种算法进行详细介绍。

#### 1. SGD方法

梯度下降（Gradient Descent）是迭代法中的一种，是指沿着梯度下降的方向求解极小值，一般可用于求解最小二乘问题。在深度学习中，当前更常用的是SGD算法，以一个小批次（Mini Batch）的数据为单位，计算一个批次的梯度，然后反向传播优化，并更新参数。SGD的表达式如式（2-1）与式（2-2）所示。

$$g_t = \nabla_{\theta_{t-1}} f(\theta_{t-1}) \quad (2-1)$$

$$\Delta \theta_t = -\eta \times g_t \quad (2-2)$$

公式中， $g_t$ 代表了参数的梯度， $\eta$ 代表了学习率（Learning Rate），即梯度影响参数更新的程度，是训练中非常重要的一个超参数。SGD优化算法的好处主要有两点：

·分担训练压力：当前数据集通常数量较多，尺度较大，使用较大的数据同时训练显然不现实，SGD则提供了小批量训练并优化网络的方法，有效分担了GPU等计算硬件的压力。

·加快收敛：由于SGD一次只采用少量的数据，这意味着会有更多次的梯度更新，在某些数据集中，其收敛速度会更快。

当然，SGD也有其自身的缺点：

·初始学习率难以确定：SGD算法依赖于一个较好的初始学习率，但设置初始学习率并不直观，并且对于不同的任务，其初始值也不固定。

·容易陷入局部最优：SGD虽然采用了小步快走的思想，但是容易陷入局部的最优解，难以跳出。

有效解决局部最优的通常做法是增加动量（momentum），其概念来自于物理学，在此是指更新的时候一定程度上保留之前更新的方向，同时利用当前批次的梯度进行微调，得到最终的梯度，可以增加优化的稳定性，降低陷入局部最优难以跳出的风险。其函数如式（2-3）与式（2-4）所示。

$$m_t = \mu \times m_{t-1} + g_t \quad (2-3)$$

$$\Delta\theta_t = -\eta \times m_t \quad (2-4)$$

公式中的 $\mu$ 为动量因子，当此次梯度下降方向与上次相同时，梯度会变大，也就会加速收敛。当梯度方向不同时，梯度会变小，从而抑制梯度更新的震荡，增加稳定性。在训练的中后期，梯度会在局部极小值周围震荡，此时 $g_t$ 接近于0，但动量的存在使得梯度更新并不是0，从而有可能跳出局部最优解。

虽然SGD算法并不完美，但在当今的深度学习算法中仍然取得了大量的应用，使用SGD有时能够获得性能更佳的模型。

## 2. Adam方法

在SGD之外，Adam是另一个较为常见的优化算法。Adam利用了梯度的一阶矩与二阶矩动态地估计调整每一个参数的学习率，是一种学习率自适应算法。

Adam的优点在于，经过调整后，每一次迭代的学习率都在一个确定范围内，使得参数更新更加平稳。此外，Adam算法可以使模型更快收敛，尤其适用于一些深层网络，或者神经网络较为复杂的场景。

下面利用PyTorch来搭建常用的优化器，传入参数包括网络中需要学习优化的Tensor对象、学习率和权值衰减等。

---

```
from torch import optim
optimizer = optim.SGD(model.parameters(), lr = 0.001, momentum = 0.9)
optimizer = optim.Adam([var1, var2], lr = 0.0001)
```

---

下面通过一个三层感知机的例子来介绍基本的优化过程。新建一个mlp.py文件，内容如下：

---

```
from torch import nn
class MLP(nn.Module):
    def __init__(self, in_dim, hid_dim1, hid_dim2, out_dim):
        super(MLP, self).__init__()
        # 通过Sequential快速搭建三层的感知机
        self.layer = nn.Sequential(
            nn.Linear(in_dim, hid_dim1),
            nn.ReLU(),
            nn.Linear(hid_dim1, hid_dim2),
            nn.ReLU(),
            nn.Linear(hid_dim2, out_dim),
            nn.ReLU()
```

---

```
    )
def forward(self, x):
    x = self.layer(x)
    return x
```

---

在终端环境中执行如下代码：

---

```
>>> import torch
>>> from mlp import MLP
>>> from torch import optim
>>> from torch import nn
# 实例化模型，并赋予每一层的维度
>>> model = MLP(28*28, 300, 200, 10)
>>> model # 打印model的结构，由3个全连接层组成
MLP(
  (layer): Sequential(
    (0): Linear(in_features=784, out_features=300, bias=True)
    (1): ReLU()
    (2): Linear(in_features=300, out_features=200, bias=True)
    (3): ReLU()
    (4): Linear(in_features=200, out_features=10, bias=True)
    (5): ReLU()
  )
)
# 采用SGD优化器，学习率为0.01
>>> optimizer = optim.SGD(params = model.parameters(), lr=0.01)
>>> data = torch.randn(10, 28*28)
>>> output = model(data)
# 由于是10分类，因此label元素从0到9，一共10个样本
>>> label = torch.Tensor([1,0,4,7,9,3,4,5,3,2]).long()
>>> label
tensor([ 1,  0,  4,  7,  9,  3,  4,  5,  3,  2])
# 求损失
>>> criterion = nn.CrossEntropyLoss()
>>> loss = criterion(output, label)
>>> loss
tensor(2.2762)
# 清空梯度，在每次优化前都需要进行此操作
>>> optimizer.zero_grad()
# 损失的反向传播
>>> loss.backward()
# 利用优化器进行梯度更新
>>> optimizer.step()
```

---

对于训练过程中的学习率调整，需要注意以下两点：

·不同参数层分配不同的学习率：优化器也可以很方便地实现将不同的网络层分配成不同的学习率，即对于特殊的层单独赋予学习率，其余的保持默认的整体学习率，具体示例如下：

---

```
# 对于model中需要单独赋予学习率的层，如special层，则使用'lr'关键字单独赋予
optimizer = optim.SGD(
    [{'params': model.special.parameters(), 'lr': 0.001},
     {'params': model.base.parameters()}, lr=0.0001})
```

---

·学习率动态调整：对于训练过程中动态的调整学习率，可以在迭代次数超过一定值后，重新赋予optim优化器新的学习率。

## 2.4 模型处理

模型是神经网络训练优化后得到的成果，包含了神经网络骨架及学习得到的参数。PyTorch对于模型的处理提供了丰富的工具，本节将从模型的生成、预训练模型的加载和模型保存3个方面进行介绍。

## 2.4.1 网络模型库：torchvision.models

对于深度学习，`torchvision.models`库提供了众多经典的网络结构与预训练模型，例如VGG、ResNet和Inception等，利用这些模型可以快速搭建物体检测网络，不需要逐层手动实现。`torchvision`包与PyTorch相独立，需要通过pip指令进行安装，如下：

```
pip install torchvision # 适用于Python 2  
pip3 install torchvision # 适用于Python 3
```

以VGG模型为例，在`torchvision.models`中，VGG模型的特征层与分类层分别用`vgg.features`与`vgg.classifier`来表示，每个部分是一个`nn.Sequential`结构，可以方便地使用与修改。下面讲解如何使用`torchvision.model`模块。

```
>>> from torch import nn  
>>> from torchvision import models  
# 通过torchvision.model直接调用VGG16的网络结构  
>>> vgg = models.vgg16()  
# VGG16的特征层包括13个卷积、13个激活函数ReLU、5个池化，一共31层  
>>> len(vgg.features)  
31  
# VGG16的分类层包括3个全连接、2个ReLU、2个Dropout，一共7层  
>>> len(vgg.classifier)  
7  
# 可以通过出现的顺序直接索引每一层  
>>> vgg.classifier[-1]  
Linear(in_features=4096, out_features=1000, bias=True)  
# 也可以选取某一部分，如下代表了特征网络的最后一个卷积模组  
>>> vgg.features[24:]  
Sequential(  
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (25): ReLU(inplace)  
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
    (27): ReLU(inplace)  
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
(29): ReLU(inplace)
(30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,ceil_
mode=False)
)
```

---

## 2.4.2 加载预训练模型

对于计算机视觉的任务，包括物体检测，我们通常很难拿到很大的数据集，在这种情况下重新训练一个新的模型是比较复杂的，并且不容易调整，因此，Fine-tune（微调）是一个常用的选择。所谓Fine-tune是指利用别人在一些数据集上训练好的预训练模型，在自己的数据集上训练自己的模型。

在具体使用时，通常有两种情况，第一种是直接利用torchvision.models中自带的预训练模型，只需要在使用时赋予pretrained参数为True即可。

```
>>> from torch import nn
>>> from torchvision import models
# 通过torchvision.model直接调用VGG16的网络结构
>>> vgg = models.vgg16(pretrained=True)
```

第二种是如果想要使用自己的本地预训练模型，或者之前训练过的模型，则可以通过model.load\_state\_dict()函数操作，具体如下：

```
>>> import torch
>>> from torch import nn
>>> from torchvision import models
# 通过torchvision.model直接调用VGG16的网络结构
>>> vgg = models.vgg16()
>>> state_dict = torch.load("your model path")
# 利用load_state_dict，遍历预训练模型的关键字，如果出现在了VGG中，则加载预训练参数
>>> vgg.load_state_dict({k:v for k, v in state_dict.items() if k in
vgg.state_dict()})
```

通常来讲，对于不同的检测任务，卷积网络的前两三层的作用是非常类似的，都是提取图像的边缘信息等，因此为了保证模型训练中能够

更加稳定，一般会固定预训练网络的前两三个卷积层而不进行参数的学习。例如VGG模型，可以设置前三个卷积模组不进行参数学习，设置方式如下：

---

```
for layer in range(10):
    for p in vgg[layer].parameters():
        p.requires_grad = False
```

---

### 2.4.3 模型保存

在PyTorch中，参数的保存通过torch.save()函数实现，可保存对象包括网络模型、优化器等，而这些对象的当前状态数据可以通过自身的state\_dict()函数获取。

```
torch.save({
    'model': model.state_dict(),
    'optimizer': optimizer.state_dict(),
    'model_path.pth'})
```

## 2.5 数据处理

数据对于深度学习而言是至关重要的，丰富、完整、规范的数据集往往能训练出效果更佳的网络模型。本节将首先介绍当前较为主流的公开数据集，然后从数据的加载、数据的GPU加速、数据的可视化3个方面介绍PyTorch的使用方法。

## 2.5.1 主流公开数据集

深度学习能够取得快速发展的一个原因是建立在大量数据的基础上，“数据为王”毫不夸张。世界上一些先进的研究机构与公司开源了一些公开数据集，这些数据集规模较大，质量较高，一方面方便研究者利用这些数据训练自己的模型，同时也为先进的论文研究提供了标准的评测平台。

数据集随着深度学习算法的提升，其规模也不断变大，任务也渐渐地由简单到复杂。下面简要介绍在物体检测领域较为重要的3个公开数据集。

### 1. ImageNet数据集

ImageNet数据集首次在2009年计算机视觉与模式识别（CVPR）会议上发布，其目的是促进计算机图像识别的技术发展。ImageNet数据集从问世以来，一直被作为计算机视觉领域最重要的数据集之一，任何基于ImageNet的技术上的进步都会对计算机视觉领域产生重要的影响。ImageNet数据集一共有1400多万张图片，共计2万多个类别，超过百万的图片有明确的类别标注，以及图像中物体的位置信息，如图2.6所示。

与ImageNet数据集对应的有一个享誉全球的ImageNet国际计算机视觉挑战赛（ILSVRC），该竞赛包含了图像分类、物体检测与定位、视频物体识别等多个领域。在2012年，Hinton带领的团队利用深度学习算法在分类竞赛中完胜其他对手，在计算机领域引起了轰动，进而掀起了深度学习的浪潮。



图2.6 ImageNet数据集图片示例

随后的几年中，众多公司与知名研究机构展开了激烈的角逐，陆续诞生了众多先进的算法，而在2017年的比赛则是最后一届，以后会往更高的理解层发展。值得一提的是，通常我们在训练自己的模型时也经常使用从ImageNet上预训练得到的模型。

## 2. PASCAL VOC数据集

PASCAL VOC为图像分类与物体检测提供了一整套标准的数据集，并从2005年到2012年每年都举行一场图像检测竞赛。PASCAL全称为Pattern Analysis, Statistical Modelling and Computational Learning，其中常用的数据集主要有VOC 2007与VOC 2012两个版本，VOC 2007中包含了9963张标注过的图片及24640个物体标签。在VOC 2007之上，VOC 2012进一步升级了数据集，一共有11530张图片，包含人、狗、椅子和桌子等20个物体类别，图片大小约500×375像素。VOC整体图像质量较

好，标注比较完整，非常适合进行模型的性能测试，如图2.7所示。



图2.7 PASCAL VOC数据集图片示例

PASCAL VOC的另一个贡献在于提供了一套标准的数据集格式，尤其是对于物体检测领域，大部分的开源物体检测算法都提供了PASCAL VOC的数据接口。对于物体检测，有3个重要的文件夹，具体如下：

- JPEGImages：包含所有训练与测试的图片。
- Annotations：存放XML格式的标签数据，每一个XML文件都对应于JPEGImages文件夹下的一张图片。
- ImageSets：对于物体检测只需要Main子文件夹，并在Main文件夹中建立Trainval.txt、train.txt、val.txt及test.txt，在各文件中记录相应的图片名即可。

### 3. COCO（Common Objects in Context）数据集

COCO是由微软赞助的一个大型数据集，针对物体检测、分割、图

像语义理解和人体关节点等，拥有超过30万张图片，200万多个实例及80个物体类别。在ImageNet竞赛停办后，COCO挑战赛成为了当前物体检测领域最权威的一个标杆。相比PASCAL VOC，COCO数据集难度更大，其拥有的小物体更多，物体大小的跨度也更大，如图2.8所示。



图2.8 COCO数据集图片示例

与PASCAL VOC一样，众多的开源算方法也通常会提供以COCO格式为标准的数据加载方式。为了更好地使用数据集，COCO也提供了基于Lua、Python及MATLAB的API，具体使用方法可以参考开源代码。

当然，随着自动驾驶领域的快速发展，也出现了众多自动驾驶领域的数据集，如KITTI、Cityscape和Udacity等，具体使用方法可以查看相关数据集官网。

## 2.5.2 数据加载

PyTorch将数据集的处理过程标准化，提供了Dataset基本的数据类，并在torchvision中提供了众多数据变换函数，数据加载的具体过程主要分为3步，如图2.9所示。

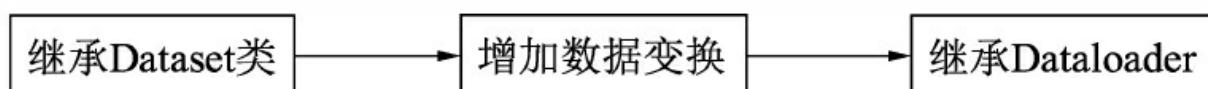


图2.9 PyTorch的数据加载过程

这3步的具体功能与实现如下：

### 1. 继承Dataset类

对于数据集的处理，PyTorch提供了torch.utils.data.Dataset这个抽象类，在使用时只需要继承该类，并重写\_\_len\_\_()和\_\_getitem\_\_()函数，即可以方便地进行数据集的迭代。

---

```
from torch.utils.data import Dataset
class my_data(Dataset):
    def __init__(self, image_path, annotation_path, transform=None):
        # 初始化，读取数据集
    def __len__(self):
        # 获取数据集的总大小
    def __getitem__(self, id):
        # 对于指定的id，读取该数据并返回
```

---

对上述类进行实例化，即可进行迭代如下：

---

```
dataset = my_data("your image path", "your annotation path") # 实例化该类
for data in dataset:
    print(data)
```

---

## 2. 数据变换与增强: torchvision.transforms

第一步虽然将数据集加载到了实例中，但在实际应用时，数据集中的图片有可能存在大小不一的情况，并且原始图片像素RGB值较大（0~255），这些都不利于神经网络的训练收敛，因此还需要进行一些图像变换工作。PyTorch为此提供了torchvision.transforms工具包，可以方便地进行图像缩放、裁剪、随机翻转、填充及张量的归一化等操作，操作对象是PIL的Image或者Tensor。

如果需要进行多个变换功能，可以利用transforms.Compose将多个变换整合起来，并且在实际使用时，通常会将变换操作集成到Dataset的继承类中。具体示例如下：

```
from torchvision import transforms
# 将transforms集成到Dataset类中，使用Compose将多个变换整合到一起
dataset = my_data("your image path", "your annotation path", transforms=
                  transforms.Compose([
                      transforms.Resize(256) # 将图像最短边缩小至256，宽高比例不变
                      # 以0.5的概率随机翻转指定的PIL图像
                      transforms.RandomHorizontalFlip()
                      # 将PIL图像转为Tensor，元素区间从[0, 255]归一到[0, 1]
                      transforms.ToTensor()
                      # 进行mean与std为0.5的标准化
                      transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
                  ]))
```

## 3. 继承dataloader

经过前两步已经可以获取每一个变换后的样本，但是仍然无法进行批量处理、随机选取等操作，因此还需要torch.utils.data.DataLoader类进一步进行封装，使用方法如下例所示，该类需要4个参数，第1个参数是之前继承了Dataset的实例，第2个参数是批量batch的大小，第3个参数是否打乱数据参数，第4个参数是使用几个线程来加载数据。

```
from torch.utils.data import DataLoader  
# 使用DataLoader进一步封装Dataset  
dataloader = DataLoader(dataset, batch_size=4,  
                        shuffle=True, num_workers=4)
```

---

dataloader是一个可迭代对象，对该实例进行迭代即可用于训练过程。

---

```
data_iter = iter(dataloader)  
for step in range(iters_per_epoch):  
    data = next(data_iter)  
    # 将data用于训练网络即可
```

---

### 2.5.3 GPU加速

PyTorch为数据在GPU上运行提供了非常便利的操作。首先可以使用`torch.cuda.is_available()`来判断当前环境下GPU是否可用，其次是对于Tensor和模型，可以直接调用`cuda()`方法将数据转移到GPU上运行，并且可以输入数字来指定具体转移到哪块GPU上运行。

```
>>> import torch
>>> from torchvision import models
>>> a = torch.randn(3,3)
>>> b = models.vgg16()
# 判断当前GPU是否可用
>>> if torch.cuda.is_available():
    a = a.cuda()
    # 指定将b转移到编号为1的GPU上
    b = b.cuda(1)
# 使用torch.device()来指定使用哪一个GPU
>>> device = torch.device("cuda: 1")
>>> c = torch.randn(3, 3, device = device, requires_grad = True)
```

对于在全局指定使用哪一块GPU，官方给出了两种方法，首先是在终端执行脚本时直接指定GPU的方式，如下：

```
CUDA_VISIBLE_DEVICES=2 python3 train.py
```

其次是在脚本中利用函数指定，如下：

```
import torch
torch.cuda.set_device(1)
```

官方建议使用第一种方法，即`CUDA_VISIBLE_DEVICE`的方式。

在工程应用中，通常使用`torch.nn.DataParallel(module, device_ids)`函数来处理多GPU并行计算的问题。示例如下：

---

```
model_gpu = nn.DataParallel(model, device_ids=[0,1])
output = model_gpu(input)
```

---

多GPU处理的实现方式是，首先将模型加载到主GPU上，然后复制模型到各个指定的GPU上，将输入数据按batch的维度进行划分，分配到每个GPU上独立进行前向计算，再将得到的损失求和并反向传播更新单个GPU上的参数，最后将更新后的参数复制到各个GPU上。

## 2.5.4 数据可视化

在训练神经网络时，如果想了解训练的具体情况，可以在终端中打印出各种训练信息，但这种方法不够直观，难以从整体角度判断模型的收敛情况，因此产生了各种数据可视化工具，可以在网络训练时更好地查看训练过程中的各个损失变化情况，监督训练过程，并为进一步的参数优化与训练优化提供方向。

在PyTorch中，常用的可视化工具有TensorBoardX和Visdom。

### 1. TensorBoardX简介

对于TensorFlow开发者，TensorBoard是较为熟悉的一套可视化工具，提供了数据曲线、计算图等数据的可视化功能。TensorBoard作为独立于TensorFlow的一个工具，可以将按固定格式保存的数据可视化。

在PyTorch中，也可以使用Tensorboard\_logger进行可视化，但其功能较少。TensorBoardX是专为PyTorch开发的一套数据可视化工具，功能与TensorBoard相当，支持曲线、图片、文本和计算图等不同形式的可视化，而且使用简单。下面以实现损失曲线的可视化为例，介绍TensorBoardX的使用方法。

通过如下命令即可完成TensorBoardX的安装。

---

```
pip3 install tensorboardX
```

---

在训练脚本中，添加如下几句指令，即可创建记录对象与数据的添加。

```
from tensorboardX import SummaryWriter  
# 创建writer对象  
writer = SummaryWriter('logs/tmp')  
# 添加曲线，并且可以使用'/'进行多级标题的指定  
writer.add_scalar('loss/total_loss', loss.data[0], total_iter)  
writer.add_scalar('loss/rpn_loss', rpn_loss.data[0], total_iter)
```

---

添加TensorBoardX指令后，则将在logs/tmp文件夹下生成events开头的记录文件，然后使用TensorBoard在终端中开启Web服务。

---

```
tensorboard --logdir=logs/tmp/  
TensorBoard 1.9.0 at http://pytorch:6006 (Press CTRL+C to quit)
```

---

在浏览器端输入上述指令下方的网址<http://pytorch:6006>，即可看到数据的可视化效果。

## 2. Visdom简介

Visdom由Facebook团队开发，是一个非常灵活的可视化工具，可用于多种数据的创建、组织和共享，支持NumPy、Torch与PyTorch数据，目的是促进远程数据的可视化，支持科学实验。

Visdom可以通过pip指令完成安装，如下：

---

```
pip3 install visdom
```

---

使用如下指令可以开启visdom服务，该服务基于Web，并默认使用8097端口。

---

```
python3 -m visdom.server
```

---

下面实现一个文本、曲线及图片的可视化示例，更多丰富的功能，

可以查看Visdom的GitHub主页。

```
import torch
import visdom
# 创建visdom客户端，使用默认端口8097，环境为first，环境的作用是对可视化的空间进行分区
vis = visdom.Visdom(env='first')
# vis对象有text()、line()和image()等函数，其中的win参数代表了显示的窗格（pane）
# 的名字
vis.text('first visdom', win='text1')
# 在此使用append为真来进行增加text，否则会覆盖之前的text
vis.text('hello PyTorch', win='text1', append=True)
# 绘制 $y=-x^2+20x+1$ 的曲线，opts可以进行标题、坐标轴标签等的配置
for i in range(20):
    vis.line(X=torch.FloatTensor([i]), Y=torch.FloatTensor([-i**2+20*i+1]),
              opts={'title': 'y=-x^2+20x+1'}, win='loss', update='append')
# 可视化一张随机图片
vis.image(torch.randn(3, 256, 256), win='random_image')
```

打开浏览器，输入<http://localhost:8097>，即可看到可视化的结果，如图2.10所示。可以看到在first的环境下，有3个窗格，分别为代码中添加的可视化数据。

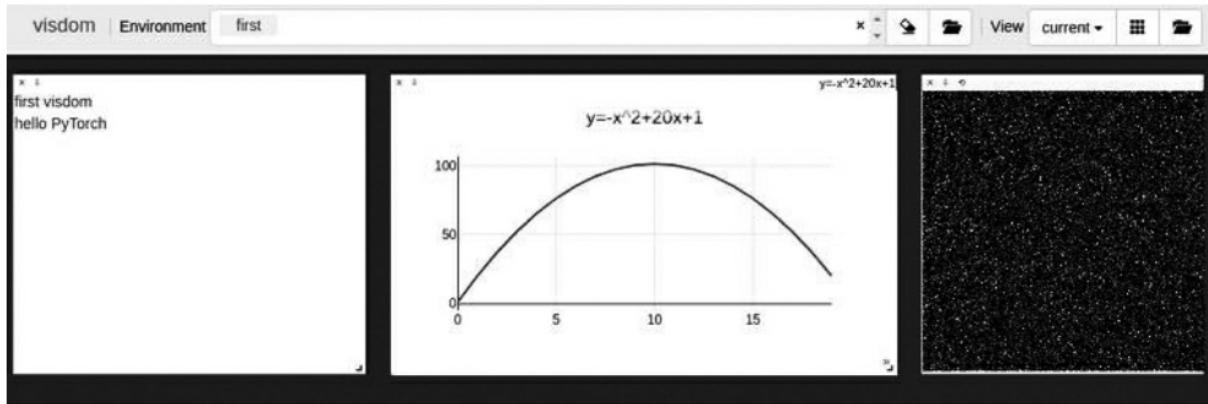


图2.10 Visdom数据可视化示例

## 2.6 总结

作为一个追求简洁高效的动态图框架，PyTorch拥有着最为贴近用户使用的接口。在本章中，笔者先后介绍了PyTorch的基础数据Tensor、计算图和神经网络工具箱torch.nn，以及模型初始化、加载与保存的基本方式。本章最后介绍了一些常见的物体检测数据集，以及与模型相关的数据处理过程。至此，PyTorch对于物体检测模型训练与前向的基本实现方法，在本章中都已简要介绍完毕。

掌握了工具之后，下一章我们就可以尝试利用PyTorch来搭建最基本的神经网络了，这也是学习物体检测算法的基础。

## 第3章 网络骨架：Backbone

当前的物体检测算法虽然各不相同，但第一步通常是利用卷积神经网络处理输入图像，生成深层的特征图，然后再利用各种算法完成区域生成与损失计算，这部分卷积神经网络是整个检测算法的“骨架”，也被称为Backbone。

Backbone是物体检测技术的基础，其中也涌现出了多种经典的结构，如VGGNet、ResNet和DenseNet等，如图3.1所示。

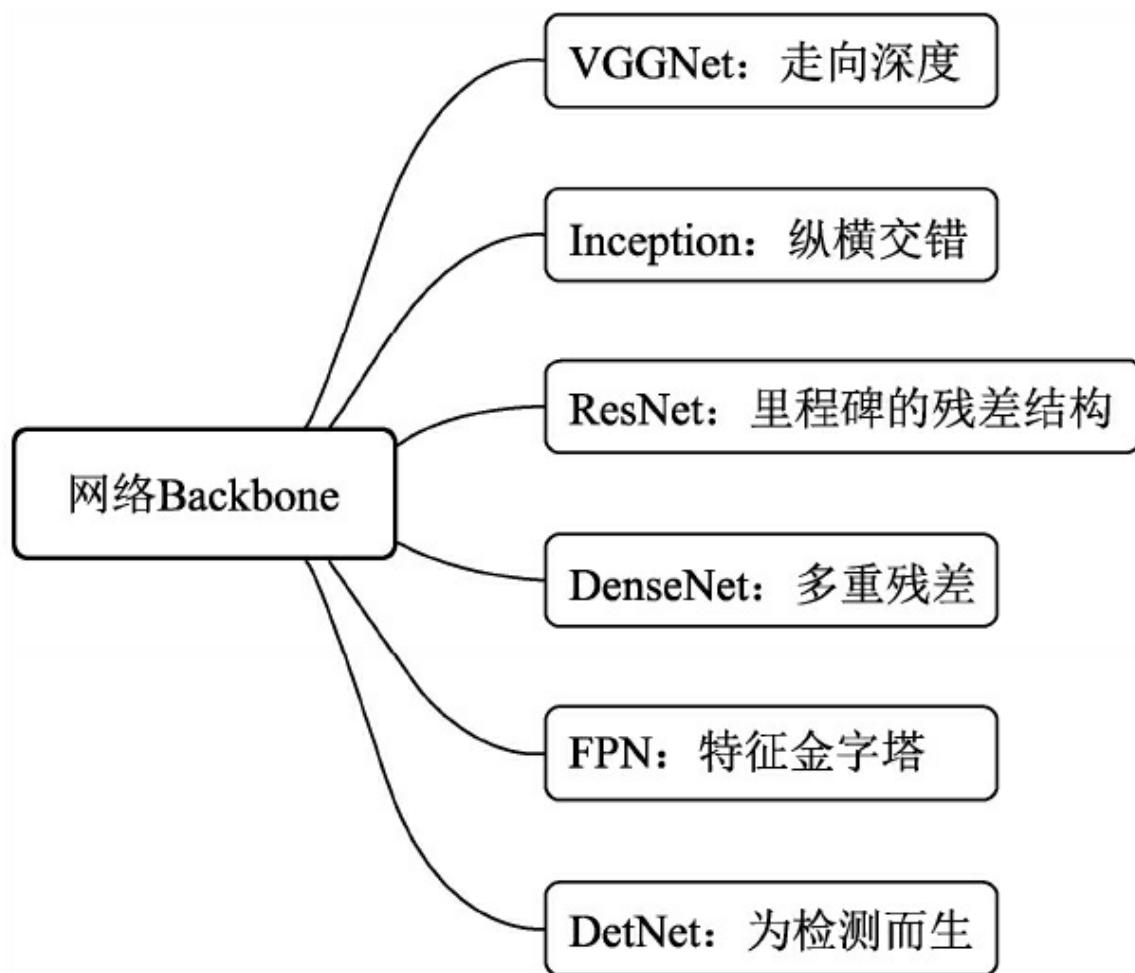


图3.1 卷积网络的经典Backbone

本章首先介绍网络骨架的基本组成单元，然后依次讲解各种经典的网络Backbone，为后续的物体检测算法奠定基础。

### 3.1 神经网络基本组成

物体检测算法使用的通常是包含卷积计算且具有深度结构的前馈神经网络，如卷积层、池化层、全连接层等不同的基本层，这些层有着不同的作用，如图3.2所示。本节将针对物体检测算法中常用的网络层进行一一介绍。

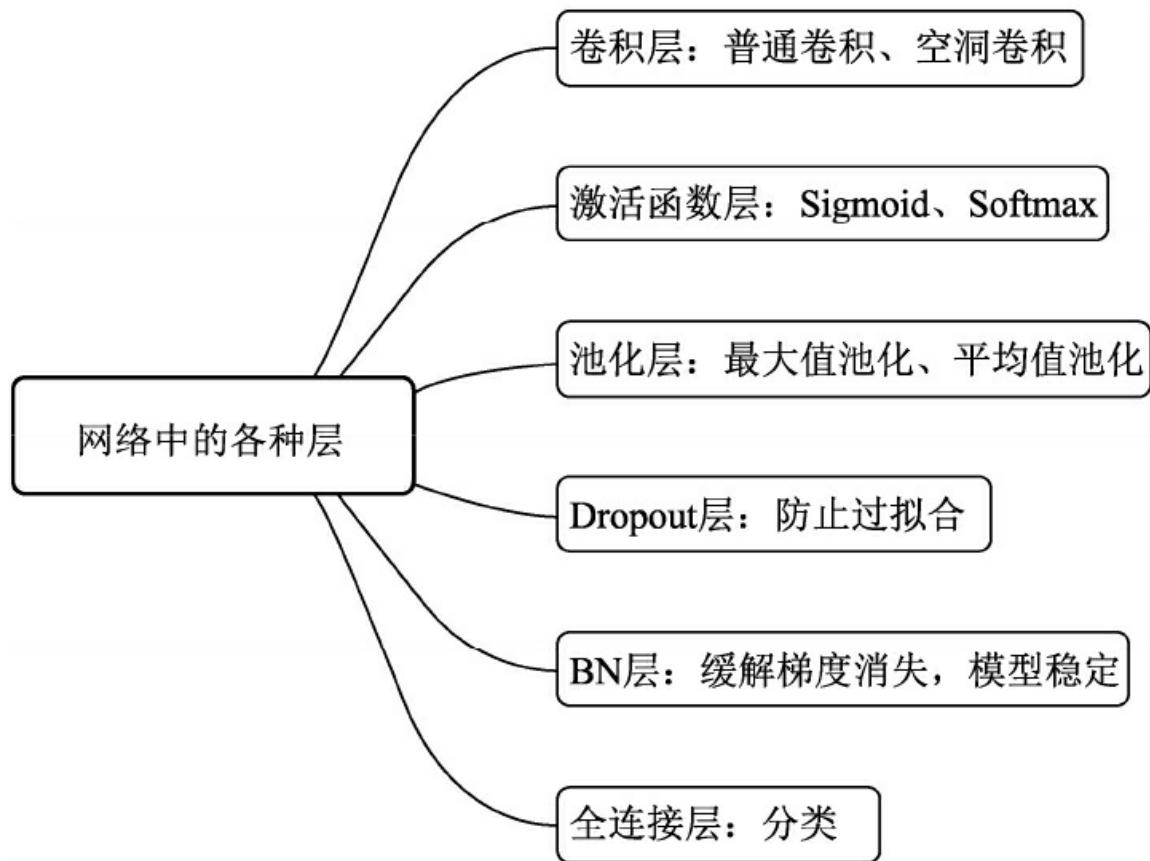


图3.2 卷积网络中的各种层

### 3.1.1 卷积层

卷积本是分析数学中的一种运算，在深度学习中使用的卷积运算通常是离散的。作为卷积神经网络中最基础的组成部分，卷积的本质是用卷积核的参数来提取数据的特征，通过矩阵点乘运算与求和运算来得到结果。

如图3.3所示为一个基本二维卷积的运算过程，公式为 $y=\omega x+b$ 。这里的特征图（ $x$ ）大小为 $1\times 5\times 5$ ，即输入通道数为1，卷积核（ $\omega$ ）的大小为 $3\times 3$ ，偏置（ $b$ ）为1，为保证输出维度和输入特征维度一致，还需要有填充（padding），这里使用zero-padding，即用0来填充。

1   0   1							
0   1   0							
1   0   1							

偏置： 1

卷积核

×

0   0   0   0   0   0   0						
0   1   8   4   2   5   0						
0   6   7   5   7   6   0						
0   5   0   8   3   5   0						
0   7   9   1   0   7   0						
0   2   3   9   8   3   0						
0   0   0   0   0   0   0						

特征图

=

9   20   19   14   13					
15   26   19   30   12					
22   20   32   23   13					
11   34   16   26   19					
12   12   19   17   4					

卷积结果

图3.3 卷积计算的基本过程

卷积核参数与对应位置像素逐位相乘后累加作为一次计算结果。以图3.3左上角为例，其计算过程为

$1\times 0+0\times 0+1\times 0+0\times 0+1\times 1+0\times 8+1\times 0+0\times 6+1\times 7+1=9$ ，然后在特征图上进行滑动，即可得到所有的计算结果。

在PyTorch中使用卷积非常简单，接下来从代码角度介绍如何完成

卷积操作。

---

```
>>> from torch import nn
# 使用torch.nn中的Conv2d()搭建卷积层
>>> conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, stride=1,
    padding=1, dilation=1, groups=1, bias=True)
# 查看卷积核的基本信息，本质上是一个Module
>>> conv
Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
# 通过.weight与.bias查看卷积核的权重与偏置
>>> conv.weight.shape
torch.Size([1, 1, 3, 3])
>>> conv.bias.shape
torch.Size([1])
# 输入特征图，需要注意特征必须是四维，第一维作为batch数，即使是1也要保留
>>> input = torch.ones(1, 1, 5, 5)
>>> output=conv(input)
# 当前配置的卷积核可以使输入和输出的大小一致
>>> input.shape
torch.Size([1, 1, 5, 5])
>>> output.shape
torch.Size([1, 1, 5, 5])
```

---

对于torch.nn.Conv2d()来说，传入的参数含义如下：

·`in_channels`: 输入特征图的通道数，如果是RGB图像，则通道数为3。卷积中的特征图通道数一般是2的整数次幂。

·`out_channels`: 输出特征图的通道数。

·`kernel_size`: 卷积核的尺寸，常见的有1、3、5、7。

·`stride`: 步长，即卷积核在特征图上滑动的步长，一般为1。如果大于1，则输出特征图的尺寸会小于输入特征图的尺寸。

·`padding`: 填充，常见的有零填充、边缘填充等，PyTorch默认为零填充。

·**dilation**: 空洞卷积，当大于1时可以增大感受野的同时保持特征图的尺寸（后面会细讲），默认为1。

·**groups**: 可实现组卷积，即在卷积操作时不是逐点卷积，而是将输入通道分为多个组，稀疏连接达到降低计算量的目的（后续会细讲），默认为1。

·**bias**: 是否需要偏置，默认为True。

在实际使用中，特征图的维度通常都不是1，假设输入特征图维度为 $m \times w_{in} \times h_{in}$ ，输出特征图维度为 $n \times w_{out} \times h_{out}$ ，则卷积核的维度为 $n \times m \times k \times k$ ，在此产生的乘法操作次数为 $n \times w_{out} \times h_{out} \times m \times k \times k$ 。

### 3.1.2 激活函数层

神经网络如果仅仅是由线性的卷积运算堆叠组成，则其无法形成复杂的表达空间，也就很难提取出高语义的信息，因此还需要加入非线性的映射，又称为激活函数，可以逼近任意的非线性函数，以提升整个神经网络的表达能力。在物体检测任务中，常用的激活函数有Sigmoid、ReLU及Softmax函数。

#### 1. Sigmoid函数

Sigmoid型函数又称为Logistic函数，模拟了生物的神经元特性，即当神经元获得的输入信号累计超过一定的阈值后，神经元被激活而处于兴奋状态，否则处于抑制状态。其函数表达如式（3-1）所示。

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \quad (3-1)$$

Sigmoid函数曲线与梯度曲线如图3.4所示。可以看到，Sigmoid函数将特征压缩到了(0,1)区间，0端对应抑制状态，而1对应激活状态，中间部分梯度较大。

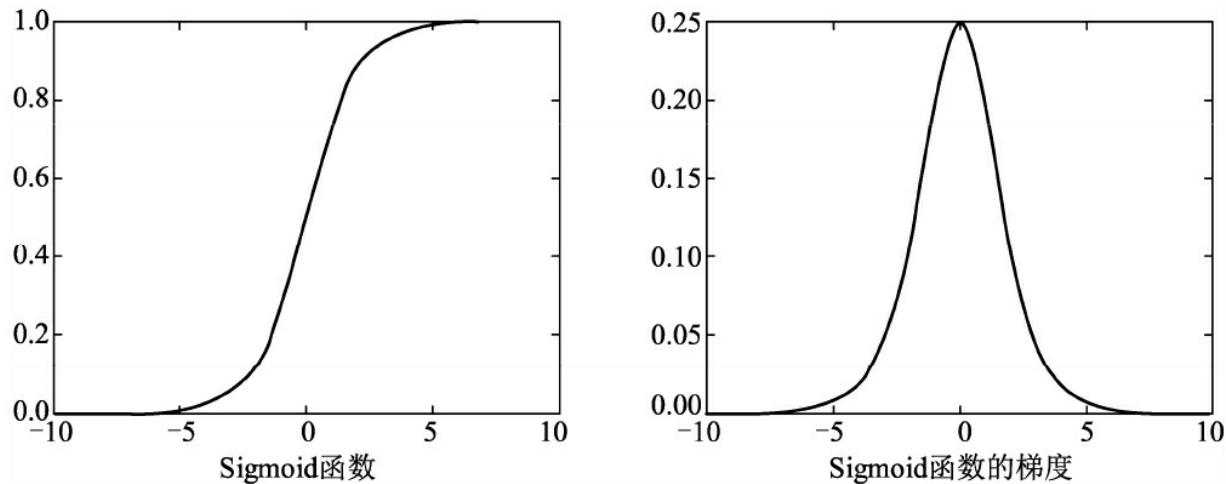


图3.4 Sigmoid函数及其梯度曲线

PyTorch实现Sigmoid函数很简单，示例如下：

```
# 引入torch.nn模块
>>> import torch
>>> from torch import nn
>>> input = torch.ones(1,1,2,2)
>>> input
tensor([[[[ 1.,  1.],
          [ 1.,  1.]]]])
>>> sigmoid = nn.Sigmoid() # 使用nn.Sigmoid()实例化sigmoid
>>> sigmoid(input)
tensor([[[[ 0.7311,  0.7311],
          [ 0.7311,  0.7311]]]])
```

Sigmoid函数可以用来做二分类，但其计算量较大，并且容易出现梯度消失现象。从曲线图（图3.4）中可以看出，在Sigmoid函数两侧的特征导数接近于0，这将导致在梯度反传时损失的误差难以传递到前面的网络层（因为根据链式求导，梯度接近于0）。

## 2. ReLU函数

为了缓解梯度消失现象，修正线性单元（Rectified Linear Unit, ReLU）被引入到神经网络中。由于其优越的性能与简单优雅的实现，

ReLU已经成为目前卷积神经网络中最为常用的激活函数之一。ReLU函数的表达式如式（3-2）所示。

$$ReLU(x) = \max(0, x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (3-2)$$

ReLU函数及其梯度曲线如图3.5所示。可以看出，在小于0的部分，值与梯度皆为0，而在大于0的部分中导数保持为1，避免了Sigmoid函数中梯度接近于0导致的梯度消失问题。

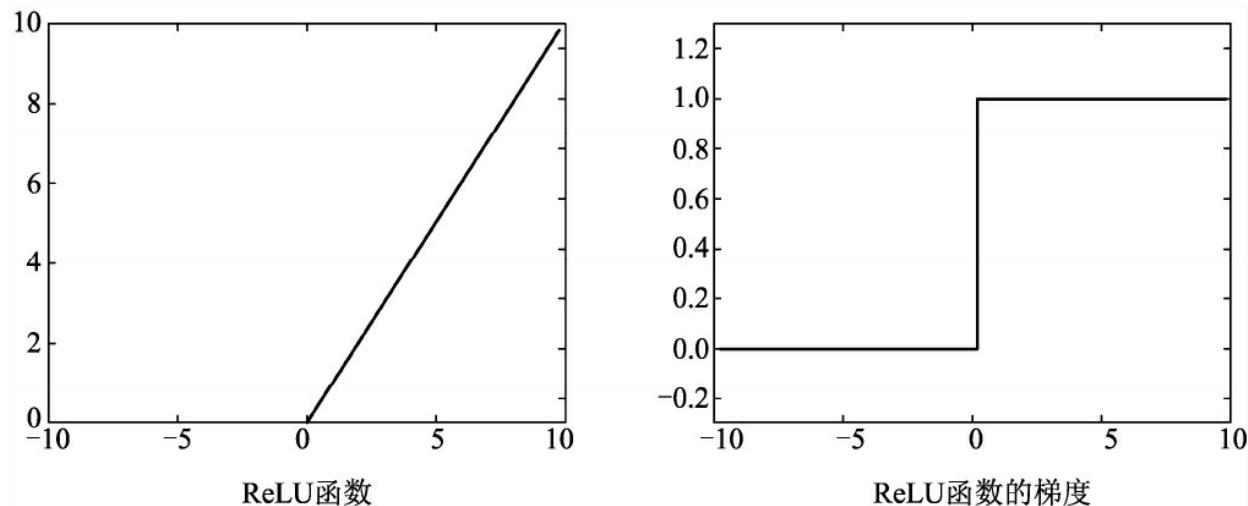


图3.5 ReLU函数及其梯度曲线

下面是PyTorch实现ReLU激活函数示例。

```
>>> import torch
>>> from torch import nn
>>> input = torch.randn(1,1,2,2)
>>> input
tensor([[[[ 1.8021,  0.5564],
          [-1.2117,  1.2384]]]])
# nn.ReLU()可以实现inplace操作，即可以直接将运算结果覆盖到输入中，以节省内存
>>> relu = nn.ReLU(inplace=True)
>>> relu(input)                                # 可以看出大于0的值保持不变，小于0的值被置为0
tensor([[[[ 1.8021,  0.5564],
```

```
[ 0.0000, 1.2384]]])
```

---

ReLU函数计算简单，收敛快，并在众多卷积网络中验证了其有效性。

### 3. Leaky ReLU函数

ReLU激活函数虽然高效，但是其将负区间所有的输入都强行置为0，Leaky ReLU函数优化了这一点，在负区间内避免了直接置0，而是赋予很小的权重，其函数表达式如式（3-3）所示。

$$\text{LeakyReLU}(x) = \max\left(\frac{1}{a_i}x, x\right) = \begin{cases} \frac{1}{a_i}x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (3-3)$$

以上公式中的 $a_i$ 代表权重，即小于0的值被缩小的比例。Leaky ReLU的函数曲线如图3.6所示。

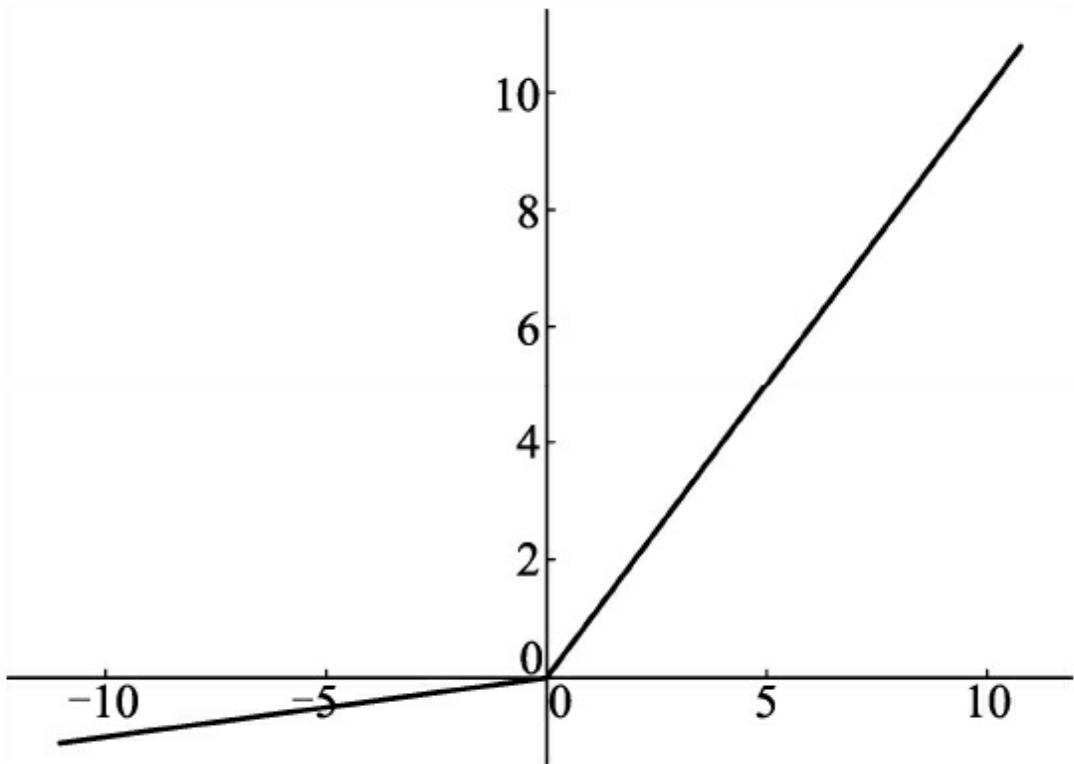


图3.6 Leaky ReLU函数曲线

下面使用PyTorch来实现简单的Leaky ReLU激活函数过程。

```
>>> import torch
>>> from torch import nn
>>> input = torch.randn(1,1,2,2)
>>> input
tensor([[[[-1.7528,  0.1343],
          [-0.9622,  0.0120]]]])
# 利用nn.LeakyReLU()构建激活函数，并且其为0.04，即ai为25，True代表in-place操作
>>> leakyrelu = nn.LeakyReLU(0.04, True)
>>> leakyrelu(input) # 从结果看大于0的值保持不变，小于0的值被以0.04的比例缩小
tensor([[[[-0.0701,  0.1343],
          [-0.0385,  0.0120]]]])
```

虽然从理论上讲，Leaky ReLU函数的使用效果应该要比ReLU函数好，但是从大量实验结果来看并没有看出其效果比ReLU好。此外，对于ReLU函数的变种，除了Leaky ReLU函数之外，还有PReLU和RReLU

函数等，这里不做详细介绍。

#### 4. Softmax函数

在物体检测中，通常需要面对多物体分类问题，虽然可以使用Sigmoid函数来构造多个二分类器，但比较麻烦，多物体类别较为常用的分类器是Softmax函数。

在具体的分类任务中，Softmax函数的输入往往是多个类别的得分，输出则是每一个类别对应的概率，所有类别的概率取值都在0~1之间，且和为1。Softmax函数的表达如式（3-4）所示，其中， $V_i$ 表示第*i*个类别的得分， $C$ 代表分类的类别总数，输出 $S_i$ 为第*i*个类别的概率。

$$S_i = \frac{e^{V_i}}{\sum_j^C e^{V_j}} \quad (3-4)$$

在PyTorch中，Softmax函数在torch.nn.functional库中，使用方法如下：

---

```
>>> import torch.nn.functional as F
>>> score = torch.randn(1,4)
>>> score
tensor([[ 1.3858, -0.4449, -1.7636,  0.9768]])
# 利用torch.nn.functional.softmax()函数，第二个参数表示按照第几个维度进行
# Softmax计算
>>> F.softmax(score, 1)
tensor([[ 0.5355,  0.0858,  0.0230,  0.3557]])
```

---

### 3.1.3 池化层

在卷积网络中，通常会在卷积层之间增加池化（Pooling）层，以降低特征图的参数量，提升计算速度，增加感受野，是一种降采样操作。池化是一种较强的先验，可以使模型更关注全局特征而非局部出现的位置，这种降维的过程可以保留一些重要的特征信息，提升容错能力，并且还能在一定程度上起到防止过拟合的作用。

在物体检测中，常用的池化有最大值池化（Max Pooling）与平均值池化（Average Pooling）。池化层有两个主要的输入参数，即核尺寸 kernel\_size 与步长 stride。如图3.7所示为一个核尺寸与步长都为2的最大值池化过程，以左上角为例，9、20、15与26进行最大值池化，保留26。

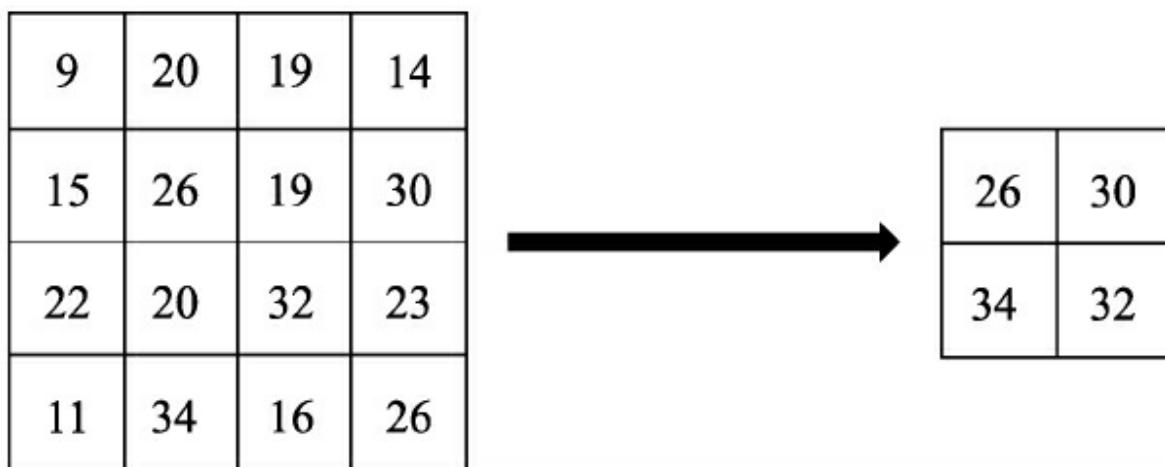


图3.7 池化过程示例

下面是PyTorch对于池化层的实现。

---

```
>>> import torch  
>>> from torch import nn
```

```
# 池化主要需要两个参数，第一个参数代表池化区域大小，第二个参数表示步长
>>> max_pooling = nn.MaxPool2d(2, stride=2)
>>> aver_pooling = nn.AvgPool2d(2, stride=2)
>>> input = torch.randn(1,1,4,4)
>>> input
tensor([[[[ 1.4873, -0.2228, -0.3972, -0.1336],
          [ 0.6129, 0.4522, -0.3175, -1.2225],
          [-1.0811, 2.3458, -0.4562, -1.9391],
          [-0.3609, -2.0500, -1.2374, -0.2012]]]])
# 调用最大值池化与平均值池化，可以看到size从[1, 1, 4, 4]变为了[1, 1, 2, 2]
>>> max_pooling(input)
tensor([[[[ 1.4873, -0.1336],
          [ 2.3458, -0.2012]]]])
>>> aver_pooling(input)
tensor([[[[ 0.5824, -0.5177],
          [-0.2866, -0.9585]]]])
```

---

### 3.1.4 Dropout层

在深度学习中，当参数过多而训练样本又比较少时，模型容易产生过拟合现象。过拟合是很多深度学习乃至机器学习算法的通病，具体表现为在训练集上预测准确率高，而在测试集上准确率大幅下降。2012年，Hinton等人提出了Dropout算法，可以比较有效地缓解过拟合现象的发生，起到一定正则化的效果。

Dropout的基本思想如图3.8所示，在训练时，每个神经元以概率 $p$ 保留，即以 $1-p$ 的概率停止工作，每次前向传播保留下来的神经元都不同，这样可以使得模型不太依赖于某些局部特征，泛化性能更强。在测试时，为了保证相同的输出期望值，每个参数还要乘以 $p$ 。当然还有另外一种计算方式称为Inverted Dropout，即在训练时将保留下的神经元乘以 $1/p$ ，这样测试时就不再需要再改变权重。

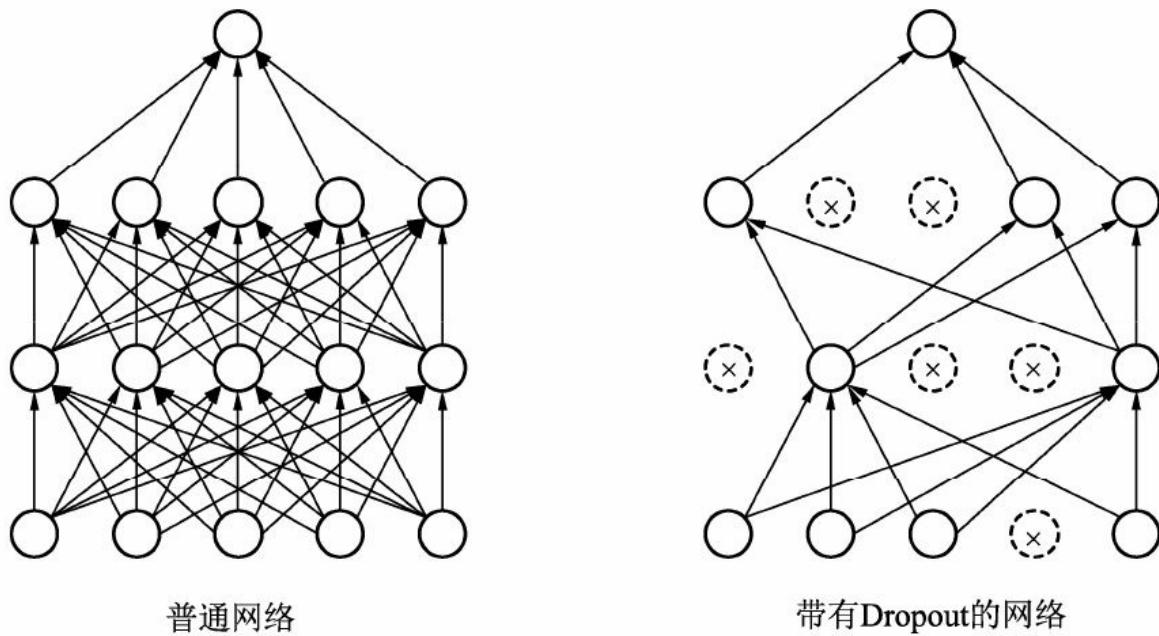


图3.8 Dropout与普通网络的对比

至于Dropout为什么可以防止过拟合，可以从以下3个方面解释。

·**多模型的平均**: 不同的固定神经网络会有不同的过拟合，多个取平均则有可能让一些相反的拟合抵消掉，而Dropout每次都是不同的神经元失活，可以看做是多个模型的平均，类似于多数投票取胜的策略。

·**减少神经元间的依赖**: 由于两个神经元不一定同时有效，因此减少了特征之间的依赖，迫使网络学习有更为鲁棒的特征，因为神经网络不应该对特定的特征敏感，而应该从众多特征中学习更为共同的规律，这也起到了正则化的效果。

·**生物进化**: Dropout类似于性别在生物进化中的角色，物种为了适应环境变化，在繁衍时取雄性和雌性的各一半基因进行组合，这样可以适应更复杂的新环境，避免了单一基因的过拟合，当环境发生变化时也不至于灭绝。

在PyTorch中使用Dropout非常简单，示例如下：

---

```
>>> import torch
>>> from torch import nn
# PyTorch将元素置0来实现Dropout层，第一个参数为置0概率，第二个为是否原地操作
>>> dropout = nn.Dropout(0.5, inplace=False)
>>> input = torch.randn(2, 64, 7, 7)
>>> output = dropout(input)
```

---

Dropout被广泛应用到全连接层中，一般保留概率设置为0.5，而在较为稀疏的卷积网络中则一般使用下一节将要介绍的BN层来正则化模型，使得训练更稳定。

### 3.1.5 BN层

为了追求更高的性能，卷积网络被设计得越来越深，然而网络却变得难以训练收敛与调参。原因在于，浅层参数的微弱变化经过多层线性变换与激活函数后会被放大，改变了每一层的输入分布，造成深层的网络需要不断调整以适应这些分布变化，最终导致模型难以训练收敛。

由于网络中参数变化导致的内部节点数据分布发生变化的现象被称做ICS（Internal Covariate Shift）。ICS现象容易使训练过程陷入饱和区，减慢网络的收敛。前面提到的ReLU从激活函数的角度出发，在一定程度上解决了梯度饱和的现象，而2015年提出的BN层，则从改变数据分布的角度避免了参数陷入饱和区。由于BN层优越的性能，其已经是当前卷积网络中的“标配”。

BN层首先对每一个batch的输入特征进行白化操作，即去均值方差过程。假设一个batch的输入数据为 $x$ :  $B = \{x_1, \dots, x_m\}$ ，首先求该batch数据的均值与方差，如式（3-5）和式（3-6）所示。

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad (3-5)$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad (3-6)$$

以上公式中， $m$ 代表batch的大小， $\mu_B$ 为批处理数据的均值， $\sigma_B^2$ 为批处理数据的方差。在求得均值方差后，利用式（3-7）进行去均值方差操作：

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad (3-7)$$

白化操作可以使输入的特征分布具有相同的均值与方差，固定了每一层的输入分布，从而加速网络的收敛。然而，白化操作虽然从一定程度上避免了梯度饱和，但也限制了网络中数据的表达能力，浅层学到的参数信息会被白化操作屏蔽掉，因此，BN层在白化操作后又增加了一个线性变换操作，让数据尽可能地恢复本身的表达能力，如公式（3-7）和公式（3-8）所示。

$$y_i \leftarrow \gamma \hat{x}_i + \beta \quad (3-8)$$

公式（3-8）中， $\gamma$ 与 $\beta$ 为新引进的可学习参数，最终的输出为 $y_i$ 。

BN层可以看做是增加了线性变换的白化操作，在实际工程中被证明了能够缓解神经网络难以训练的问题。BN层的优点主要有以下3点：

- 缓解梯度消失，加速网络收敛。BN层可以让激活函数的输入数据落在非饱和区，缓解了梯度消失问题。此外，由于每一层数据的均值与方差都在一定范围内，深层网络不必去不断适应浅层网络输入的变化，实现了层间解耦，允许每一层独立学习，也加快了网络的收敛。

- 简化调参，网络更稳定。在调参时，学习率调得过大容易出现震荡与不收敛，BN层则抑制了参数微小变化随网络加深而被放大的问题，因此对于参数变化的适应能力更强，更容易调参。

- 防止过拟合。BN层将每一个batch的均值与方差引入到网络中，由于每个batch的这两个值都不相同，可看做为训练过程增加了随机噪音，

可以起到一定的正则效果，防止过拟合。

在测试时，由于是对单个样本进行测试，没有batch的均值与方差，通常做法是在训练时将每一个batch的均值与方差都保留下来，在测试时使用所有训练样本均值与方差的平均值。

PyTorch中使用BN层很简单，示例如下：

---

```
>>> from torch import nn
# 使用BN层需要传入一个参数为num_features，即特征的通道数
>>> bn = nn.BatchNorm2d(64)
# eps为公式中的 $\epsilon$ , momentum为均值方差的动量, affine为添加可学习参数
>>> bn
BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
>>> input = torch.randn(4, 64, 224, 224)
>>> output = bn(input)
# BN层不改变输入、输出的特征大小
>>> output.shape
torch.Size([4, 64, 224, 224])
```

---

尽管BN层取得了巨大的成功，但仍有一定的弊端，主要体现在以下两点：

- 由于是在batch的维度进行归一化，BN层要求较大的batch才能有效地工作，而物体检测等任务由于占用内存较高，限制了batch的大小，这会限制BN层有效地发挥归一化功能。

- 数据的batch大小在训练与测试时往往不一样。在训练时一般采用滑动来计算平均值与方差，在测试时直接拿训练集的平均值与方差来使用。这种方式会导致测试集依赖于训练集，然而有时训练集与测试集的数据分布并不一致。

因此，我们能不能避开batch来进行归一化呢？答案是可以的，最新的工作GN（Group Normalization）从通道方向计算均值与方差，使用更

为灵活有效，避开了batch大小对归一化的影响。

具体来讲，GN先将特征图的通道分为很多个组，对每一个组内的参数做归一化，而不是batch。GN之所以能够工作的原因，笔者认为是在特征图中，不同的通道代表了不同的意义，例如形状、边缘和纹理等，这些不同的通道并不是完全独立地分布，而是可以放到一起进行归一化分析。

### 3.1.6 全连接层

全连接层（Fully Connected Layers）一般连接到卷积网络输出的特征图后边，特点是每一个节点都与上下层的所有节点相连，输入与输出都被延展成一维向量，因此从参数量来看全连接层的参数量是最多的，如图3.9所示。

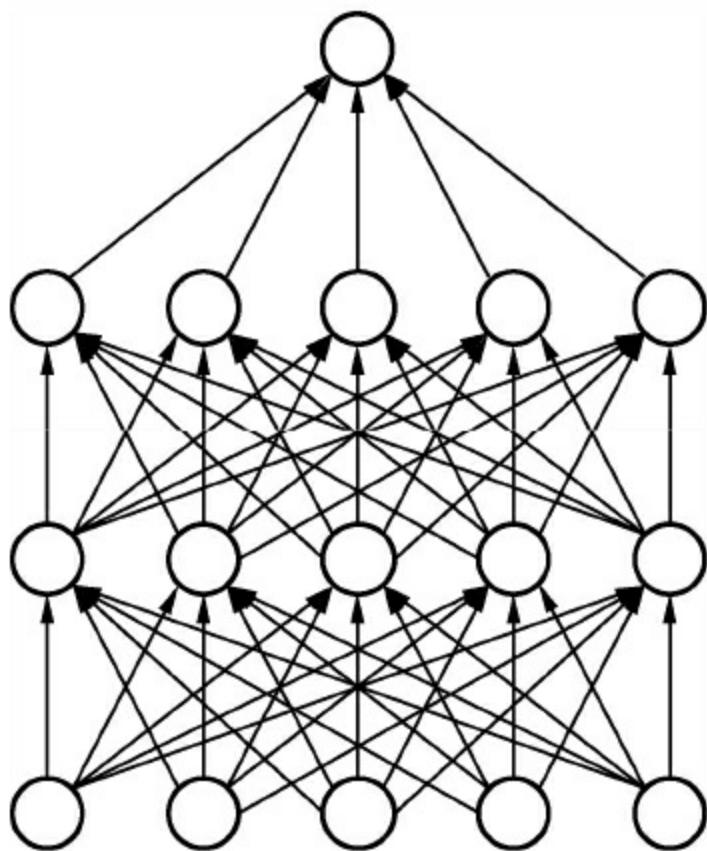


图3.9 全连接网络的计算过程

在物体检测算法中，卷积网络的主要作用是从局部到整体地提取图像的特征，而全连接层则用来将卷积抽象出的特征图进一步映射到特定维度的标签空间，以求取损失或者输出预测结果。

在第2章中的感知机例子即是使用了三层的全连接网络进行分类，PyTorch使用全连接层需要指定输入的与输出的维度。示例如下：

```
>>> import torch
>>> from torch import nn
# 第一维表示一共有4个样本
>>> input = torch.randn(4, 1024)
>>> linear = nn.Linear(1024, 4096)
>>> output = linear(input)
>>> input.shape
torch.Size([4, 1024])
>>> output.shape
torch.Size([4, 4096])
```

然而，随着深度学习算法的发展，全连接层的缺点也逐渐暴露了出来，最致命的问题在于其参数量的庞大。在此以VGGNet为例说明，其第一个全连接层的输入特征为 $7 \times 7 \times 512 = 25088$ 个节点，输出特征是大小为4096的一维向量，由于输出层的每一个点都来自于上一层所有点的权重相加，因此这一层的参数量为 $25088 \times 4096 \approx 10^8$ 。相比之下，VGGNet最后一个卷积层的卷积核大小为 $3 \times 3 \times 512 \times 512 \approx 2.4 \times 10^6$ ，全连接层的参数量是这一个卷积层的40多倍。

大量的参数会导致网络模型应用部署困难，并且其中存在着大量的参数冗余，也容易发生过拟合的现象。在很多场景中，我们可以使用全局平均池化层（Global Average Pooling，GAP）来取代全连接层，这种思想最早见于NIN（Network in Network）网络中，总体上，使用GAP有如下3点好处：

- 利用池化实现了降维，极大地减少了网络的参数量。
- 将特征提取与分类合二为一，一定程度上可以防止过拟合。
- 由于去除了全连接层，可以实现任意图像尺度的输入。

### 3.1.7 深入理解感受野

感受野 (Receptive Field) 是指特征图上的某个点能看到的输入图像的区域，即特征图上的点是由输入图像中感受野大小区域的计算得到的。举个简单的例子，如图3.10所示为一个三层卷积网络，每一层的卷积核为 $3 \times 3$ ，步长为1，可以看到第一层对应的感受野是 $3 \times 3$ ，第二层是 $5 \times 5$ ，第三层则是 $7 \times 7$ 。

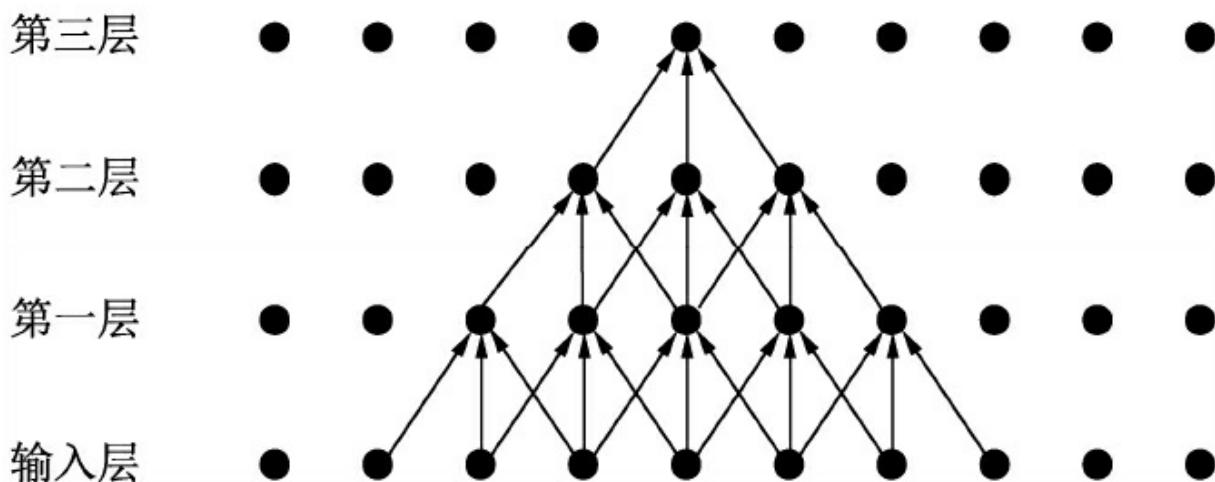


图3.10 感受野示意图

卷积层和池化层都会影响感受野，而激活函数层通常对于感受野没有影响。对于一般的卷积神经网络，感受野可由式（3-9）和式（3-10）计算得出。

$$RF_{l+1} = RF_l + (k - 1) \times S_l \quad (3-9)$$

$$S_l = \prod_{i=1}^l Stride_i \quad (3-10)$$

其中， $RF_{l+1}$ 与 $RF_l$ 分别代表第 $l+1$ 层与第 $l$ 层的感受野， $k$ 代表第 $l+1$ 层卷积核的大小， $S_l$ 代表前 $l$ 层的步长之积。注意，当前层的步长并不影响当前层的感受野。

通过上述公式求取出的感受野通常很大，而实际的有效感受野（Effective Receptive Field）往往小于理论感受野。从图3.10也可以看出，虽然第三层的感受野是 $7 \times 7$ ，但是输入层中边缘点的使用次数明显比中间点要少，因此做出的贡献不同。经过多层的卷积堆叠之后，输入层对于特征图点做出的贡献分布呈高斯分布形状。

理解感受野是理解卷积神经网络工作的基础，尤其是对于使用Anchor作为强先验区域的物体检测算法，如Faster RCNN和SSD，如何设置Anchor的大小，Anchor应该对应在特征图的哪一层，都应当考虑感受野。通常来讲，Anchor的大小应该与感受野相匹配，尤其是有效的感受野，过大或过小都不好。

在卷积网络中，有时还需要计算特征图的大小，一般可以按照式(3-11)进行计算。

$$n_{\text{out}} = \frac{n_{\text{in}} + 2p - k}{s} + 1 \quad (3-11)$$

其中， $n_{\text{in}}$ 与 $n_{\text{out}}$ 分别为输入特征图与输出特征图的尺寸， $p$ 代表这一层的padding大小， $k$ 代表这一层的卷积核大小， $s$ 为步长。

### 3.1.8 详解空洞卷积（Dilated Convolution）

空洞卷积最初是为解决图像分割的问题而提出的。常见的图像分割算法通常使用池化层来增大感受野，同时也缩小了特征图尺寸，然后再利用上采样还原图像尺寸。特征图缩小再放大的过程造成了精度上的损失，因此需要有一种操作可以在增加感受野的同时保持特征图的尺寸不变，从而替代池化与上采样操作，在这种需求下，空洞卷积就诞生了。

在近几年的物体检测发展中，空洞卷积也发挥了重要的作用。因为虽然物体检测不要求逐像素地检测，但是保持特征图的尺寸较大，对于小物体的检测及物体的定位来说也是至关重要的。

空洞卷积，顾名思义就是卷积核中间带有一些洞，跳过一些元素进行卷积。在此以 $3 \times 3$ 卷积为例，其中，图3.11a是普通的卷积过程，在卷积核紧密排列在特征图上滑动计算，而图3.11b代表了空洞数为2的空洞卷积，可以看到，在特征图上每2行或者2列选取元素与卷积核卷积。类似地，图3.11c代表了空洞数为3的空洞卷积。

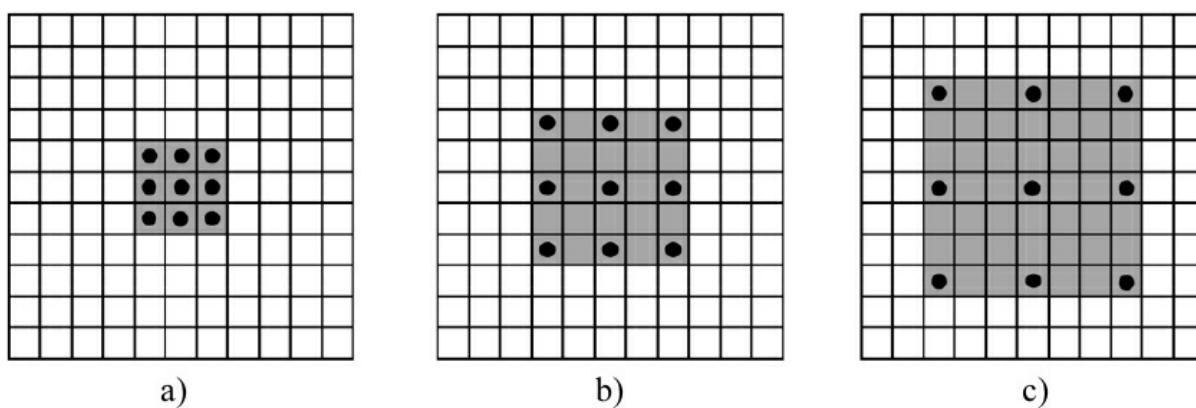


图3.11 普通卷积与空洞卷积的对比

在代码实现时，空洞卷积有一个额外的超参数dilation\_rate，表示空

洞数, 普通卷积dilation rate默认为1, 图3.11中的b与c的dilation rate分别为2与3。

在图3.11中, 同样的一个 $3 \times 3$ 卷积, 却可以起到 $5 \times 5$ 、 $7 \times 7$ 等卷积的效果。可以看出, 空洞卷积在不增加参数量的前提下, 增大了感受野。假设空洞卷积的卷积核大小为 $k$ , 空洞数为 $d$ , 则其等效卷积核大小 $k'$ 计算如式(3-12)所示。

$$k' = k + (k - 1) \times (d - 1) \quad (3-12)$$

在计算感受野时, 只需要将原来的卷积核大小 $k$ 更换为 $k'$ 即可。

空洞卷积的优点显而易见, 在不引入额外参数的前提下可以任意扩大感受野, 同时保持特征图的分辨率不变。这一点在分割与检测任务中十分有用, 感受野的扩大可以检测大物体, 而特征图分辨率不变使得物体定位更加精准。

PyTorch对于空洞卷积也提供了方便的实现接口, 在卷积时传入dilation参数即可。具体如下:

---

```
>>> from torch import nn
# 定义普通卷积, 默认dilation为1
>>> conv1 = nn.Conv2d(3, 256, 3, stride=1, padding=1, dilation=1)
>>> conv1
Conv2d(3, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
# 定义dilation为2的卷积, 打印卷积后会有dilation的参数
>>> conv2 = nn.Conv2d(3, 256, 3, stride=1, padding=1, dilation=2)
>>> conv2
Conv2d(3, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), dilation=(2, 2))
```

---

当然, 空洞卷积也有自己的一些缺陷, 主要表现在以下3个方面:

·网格效应 (Gridding Effect) : 由于空洞卷积是一种稀疏的采样方

式，当多个空洞卷积叠加时，有些像素根本没有被利用到，会损失信息的连续性与相关性，进而影响分割、检测等要求较高的任务。

·远距离的信息没有相关性：空洞卷积采取了稀疏的采样方式，导致远距离卷积得到的结果之间缺乏相关性，进而影响分类的结果。

·不同尺度物体的关系：大的dilation rate对于大物体分割与检测有利，但是对于小物体则有弊无利，**如何处理好多尺度问题的检测，是空洞卷积设计的重点。**

对于上述问题，有多篇文章提出了不同的解决方法，典型的有图森未来提出的HDC (Hybrid Dilated Convolution) 结构。该结构的设计准则是堆叠卷积的dilation rate不能有大于1的公约数，同时将dilation rate设置为类似于[1,2,5,1,2,5]这样的锯齿类结构。此外各dilation rate之间还需要满足一个数学公式，这样可以尽可能地覆盖所有空洞，以解决网格效应与远距离信息的相关性问题，具体细节可参考相关资料。

## 3.2 走向深度：VGGNet

随着AlexNet在2012年ImageNet大赛上大放异彩后，卷积网络进入了飞速的发展阶段，而2014年的ImageNet亚军结构VGGNet（Visual Geometry Group Network）则将卷积网络进行了改良，探索了网络深度与性能的关系，用更小的卷积核与更深的网络结构，取得了较好的效果，成为卷积结构发展史上较为重要的一个网络。

VGGNet网络结构组成如图3.12所示，一共有6个不同的版本，最常用的是VGG16。从图3.12中可以看出，VGGNet采用了五组卷积与三个全连接层，最后使用Softmax做分类。VGGNet有一个显著的特点：每次经过池化层（maxpool）后特征图的尺寸减小一倍，而通道数则增加一倍（最后一个池化层除外）。

AlexNet中有使用到 $5 \times 5$ 的卷积核，而在VGGNet中，使用的卷积核基本都是 $3 \times 3$ ，而且很多地方出现了多个 $3 \times 3$ 堆叠的现象，这种结构的优点在于，首先从感受野来看，两个 $3 \times 3$ 的卷积核与一个 $5 \times 5$ 的卷积核是一样的；其次，同等感受野时， $3 \times 3$ 卷积核的参数量更少。更为重要的是，两个 $3 \times 3$ 卷积核的非线性能力要比 $5 \times 5$ 卷积核强，因为其拥有两个激活函数，可大大提高卷积网络的学习能力。

下面使用PyTorch来搭建VGG16经典网络结构，新建一个vgg.py文件，并输入以下内容：

---

```
from torch import nn
class VGG(nn.Module):
    def __init__(self, num_classes=1000):
        super(VGG, self).__init__()
        layers = []
        in_dim = 3
        out_dim = 64
```

```
# 循环构造卷积层，一共有13个卷积层
for i in range(13):
    layers += [nn.Conv2d(in_dim, out_dim, 3, 1, 1), nn.ReLU(inplace=True)]
    in_dim = out_dim
    # 在第2、4、7、10、13个卷积层后增加池化层
    if i==1 or i==3 or i==6 or i==9 or i==12:
        layers += [nn.MaxPool2d(2, 2)]
    # 第10个卷积后保持和前边的通道数一致，都为512，其余加倍
    if i!=9:
        out_dim*=2
self.features = nn.Sequential(*layers)
# VGGNet的3个全连接层，中间有ReLU与Dropout层
self.classifier = nn.Sequential(
    nn.Linear(512 * 7 * 7, 4096),
    nn.ReLU(True),
    nn.Dropout(),
    nn.Linear(4096, 4096),
    nn.ReLU(True),
    nn.Dropout(),
    nn.Linear(4096, num_classes),
)
def forward(self, x):
    x = self.features(x)
    # 这里是将特征图的维度从[1, 512, 7, 7]变到[1, 512*7*7]
    x = x.view(x.size(0), -1)
    x = self.classifier(x)
    return x
```

---

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weights layers	11 weights layers	13 weights layers	16 weights layers	16 weights layers	19 weights layers
Input(224×224 RGB image)					
Conv3-64	Conv3-64 LRN	Conv3-64 <b>Conv3-64</b>	Conv3-64 Conv3-64	Conv3-64 Conv3-64	Conv3-64 Conv3-64
maxpool					
Conv3-128	Conv3-128	Conv3-128 <b>Conv3-128</b>	Conv3-128 Conv3-128	Conv3-128 Conv3-128	Conv3-128 Conv3-128
maxpool					
Conv3-256 Conv3-256	Conv3-256 Conv3-256	Conv3-256 Conv3-256	Conv3-256 Conv3-256 <b>Conv1-256</b>	Conv3-256 Conv3-256 <b>Conv3-256</b>	Conv3-256 Conv3-256 Conv3-256 <b>Conv3-256</b>
maxpool					
Conv3-512 Conv3-512	Conv3-512 Conv3-512	Conv3-512 Conv3-512	Conv3-512 Conv3-512 <b>Conv3-512</b>	Conv3-512 Conv3-512 <b>Conv3-512</b>	Conv3-512 Conv3-512 Conv3-512 <b>Conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
Soft-max					

图3.12 VGG网络结构图

在终端中进入上述vgg.py文件的同级目录，输入python3进入交互式环境，利用下面代码调用该模块。

---

```
>>> import torch
>>> from vgg import VGG
# 实例化VGG类，在此设置输出分类数为21，并转移到GPU上
>>> vgg = VGG(21).cuda()
>>> input = torch.randn(1, 3, 224, 224).cuda()
>>> input.shape
torch.Size([1, 3, 224, 224])
# 调用VGG，输出21类的得分
>>> scores = vgg(input)
```

```

>>> scores.shape
torch.Size([1, 21])
# 也可以单独调用卷积模块，输出最后一层的特征图
>>> features = vgg.features(input)
>>> features.shape
torch.Size([1, 512, 7, 7])
# 打印出VGGNet的卷积层，5个卷积组一共30层
>>> vgg.features
Sequential(
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU(inplace)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU(inplace)
(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(6): ReLU(inplace)
(7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): ReLU(inplace)
(9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU(inplace)
(12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(13): ReLU(inplace)
(14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): ReLU(inplace)
(16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(18): ReLU(inplace)
(19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(20): ReLU(inplace)
(21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(22): ReLU(inplace)
(23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(25): ReLU(inplace)
(26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(27): ReLU(inplace)
(28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29): ReLU(inplace)
(30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
# 打印出VGGNet的3个全连接层
>>> vgg.classifier
Sequential(
(0): Linear(in_features=25088, out_features=4096, bias=True)
(1): ReLU(inplace)
(2): Dropout(p=0.5)
(3): Linear(in_features=4096, out_features=4096, bias=True)

```

```
(4): ReLU(inplace)
(5): Dropout(p=0.5)
(6): Linear(in_features=4096, out_features=21, bias=True)
)
```

---

VGGNet简单灵活，拓展性很强，并且迁移到其他数据集上的泛化能力也很好，因此时至今日有很多检测与分割算法仍采用VGGNet的网络骨架。

### 3.3 纵横交错：Inception

一般来说，增加网络的深度与宽度可以提升网络的性能，但是这样做也会带来参数量的大幅度增加，同时较深的网络需要较多的数据，否则容易产生过拟合现象。除此之外，增加神经网络的深度容易带来梯度消失的现象。在2014年的ImageNet大赛上，获得冠军的Inception v1（又名GoogLeNet）网络较好地解决了这个问题。

Inception v1网络是一个精心设计的22层卷积网络，并提出了具有良好局部特征结构的Inception模块，即对特征并行地执行多个大小不同的卷积运算与池化，最后再拼接到一起。由于 $1\times 1$ 、 $3\times 3$ 和 $5\times 5$ 的卷积运算对应不同的特征图区域，因此这样做的好处是可以得到更好的图像表征信息。

Inception模块如图3.13所示，使用了三个不同大小的卷积核进行卷积运算，同时还有一个最大值池化，然后将这4部分级联起来（通道拼接），送入下一层。

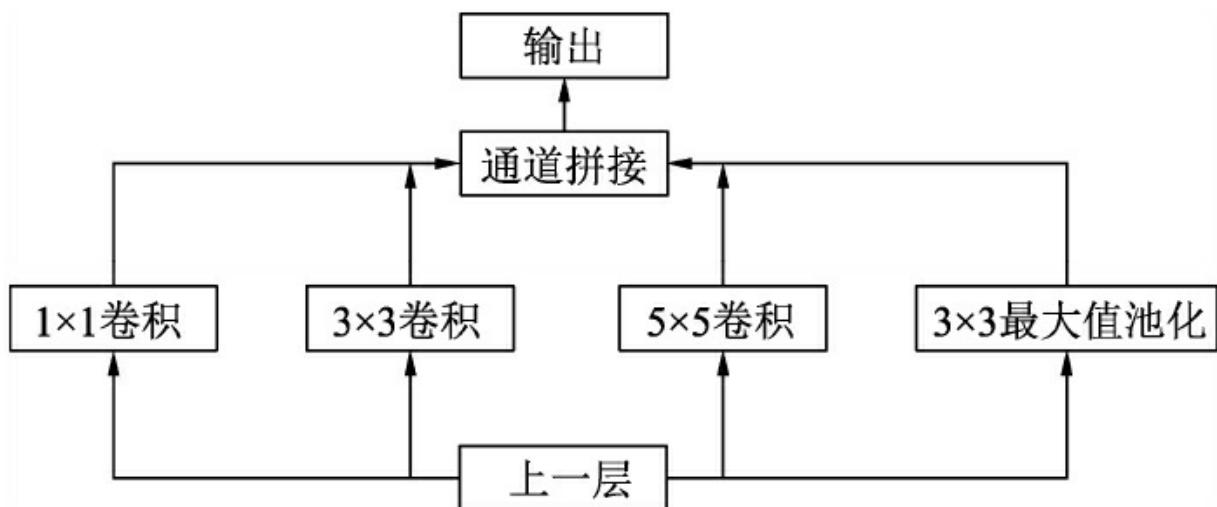


图3.13 Inception基础模块结构

在上述模块的基础上，为进一步降低网络参数量，Inception又增加了多个 $1\times 1$ 的卷积模块。如图3.14所示，这种 $1\times 1$ 的模块可以先将特征图降维，再送给 $3\times 3$ 和 $5\times 5$ 大小的卷积核，由于通道数的降低，参数量也有了较大的减少。值得一提的是，用 $1\times 1$ 卷积核实现降维的思想，在后面的多个轻量化网络中都会使用到。

Inception v1网络一共有9个上述堆叠的模块，共有22层，在最后的Inception模块处使用了全局平均池化。为了避免深层网络训练时带来的梯度消失问题，作者还引入了两个辅助的分类器，在第3个与第6个Inception模块输出后执行Softmax并计算损失，在训练时和最后的损失一并回传。

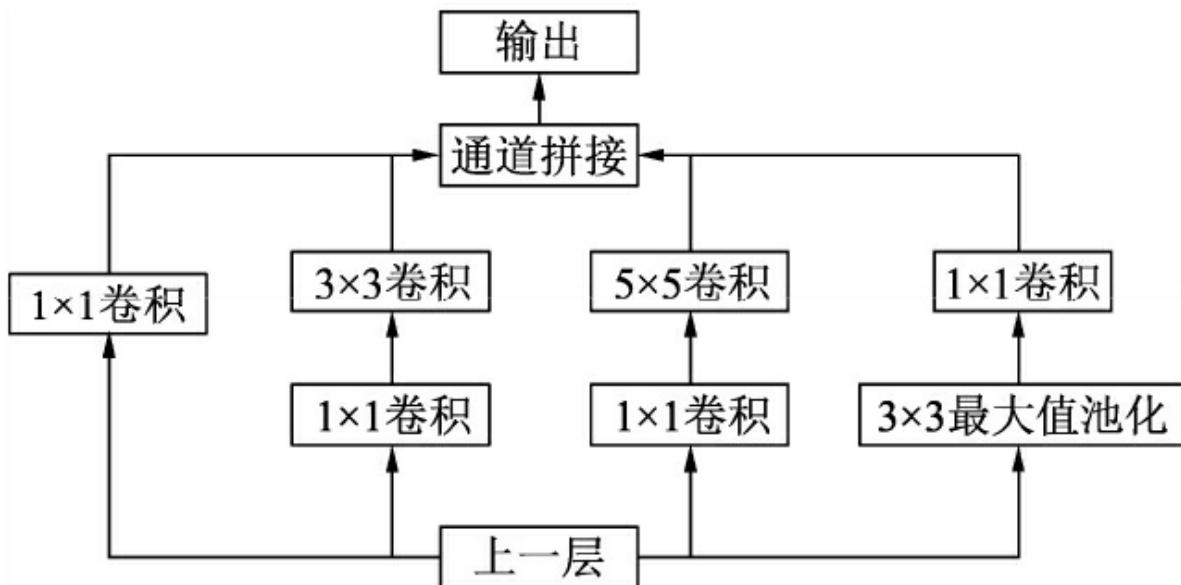


图3.14 改进的Inception模块结构

Inception v1的参数量是AlexNet的 $\frac{1}{12}$ ，VGGNet的 $\frac{1}{3}$ ，适合处理大规模数据，尤其是对于计算资源有限的平台。下面使用PyTorch来搭建一个单独的Inception模块，新建一个inceptionv1.py文件，代码如下：

---

```
import torch
```

```
from torch import nn
import torch.nn.functional as F
# 首先定义一个包含conv与ReLU的基础卷积类
class BasicConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0):
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding)
    def forward(self, x):
        x = self.conv(x)
        return F.relu(x, inplace=True)
# Inceptionv1的类，初始化时需要提供各个子模块的通道数大小
class Inceptionv1(nn.Module):
    def __init__(self, in_dim, hid_1_1, hid_2_1, hid_2_3, hid_3_1, out_3_5, out_4_1):
        super(Inceptionv1, self).__init__()
        # 下面分别是4个子模块各自的网络定义
        self.branch1x1 = BasicConv2d(in_dim, hid_1_1, 1)
        self.branch3x3 = nn.Sequential(
            BasicConv2d(in_dim, hid_2_1, 1),
            BasicConv2d(hid_2_1, hid_2_3, 3, padding=1)
        )
        self.branch5x5 = nn.Sequential(
            BasicConv2d(in_dim, hid_3_1, 1),
            BasicConv2d(hid_3_1, out_3_5, 5, padding=2)
        )
        self.branch_pool = nn.Sequential(
            nn.MaxPool2d(3, stride=1, padding=1),
            BasicConv2d(in_dim, out_4_1, 1)
        )
    def forward(self, x):
        b1 = self.branch1x1(x)
        b2 = self.branch3x3(x)
        b3 = self.branch5x5(x)
        b4 = self.branch_pool(x)
        # 将这四个子模块沿着通道方向进行拼接
        output = torch.cat((b1, b2, b3, b4), dim=1)
        return output
```

---

在终端中进入上述Inceptionv1.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用该模块。

---

```
>>> import torch
>>> from inceptionv1 import Inceptionv1
# 网络实例化，输入模块通道数，并转移到GPU上
>>> net_inceptionv1 = Inceptionv1(3, 64, 32, 64, 64, 96, 32).cuda()
```

```
>>> net_inceptionv1
Inceptionv1(
    # 第一个分支，使用 $1 \times 1$ 卷积，输出通道数为64
    (branch1x1): BasicConv2d(
        (conv): Conv2d(3, 64, kernel_size=(1, 1), stride=(1, 1))
    )
    # 第二个分支，使用 $1 \times 1$ 卷积与 $3 \times 3$ 卷积，输出通道数为64
    (branch3x3): Sequential(
        (0): BasicConv2d(
            (conv): Conv2d(3, 32, kernel_size=(1, 1), stride=(1, 1))
        )
        (1): BasicConv2d(
            (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        )
    )
    # 第三个分支，使用 $1 \times 1$ 卷积与 $5 \times 5$ 卷积，输出通道数为96
    (branch5x5): Sequential(
        (0): BasicConv2d(
            (conv): Conv2d(3, 64, kernel_size=(1, 1), stride=(1, 1))
        )
        (1): BasicConv2d(
            (conv): Conv2d(64, 96, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
        )
    )
    # 第四个分支，使用最大值池化与 $1 \times 1$ 卷积，输出通道数为32
    (branch_pool): Sequential(
        (0): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
        (1): BasicConv2d(
            (conv): Conv2d(3, 32, kernel_size=(1, 1), stride=(1, 1))
        )
    )
)
>>> input = torch.randn(1, 3, 256, 256).cuda()
>>> input.shape
torch.Size([1, 3, 256, 256])
>>> output = net_inceptionv1(input)
# 可以看到输出的通道数是输入通道数的和，即 $256=64+64+96+32$ 
>>> output.shape
torch.Size([1, 256, 256])
```

---

在Inception v1网络的基础上，随后又出现了多个Inception版本。Inception v2进一步通过卷积分解与正则化实现更高效的计算，增加了BN层，同时利用两个级联的 $3 \times 3$ 卷积取代了Inception v1版本中的 $5 \times 5$ 卷积，如图3.15所示，这种方式既减少了卷积参数量，也增加了网络的非

线性能力。

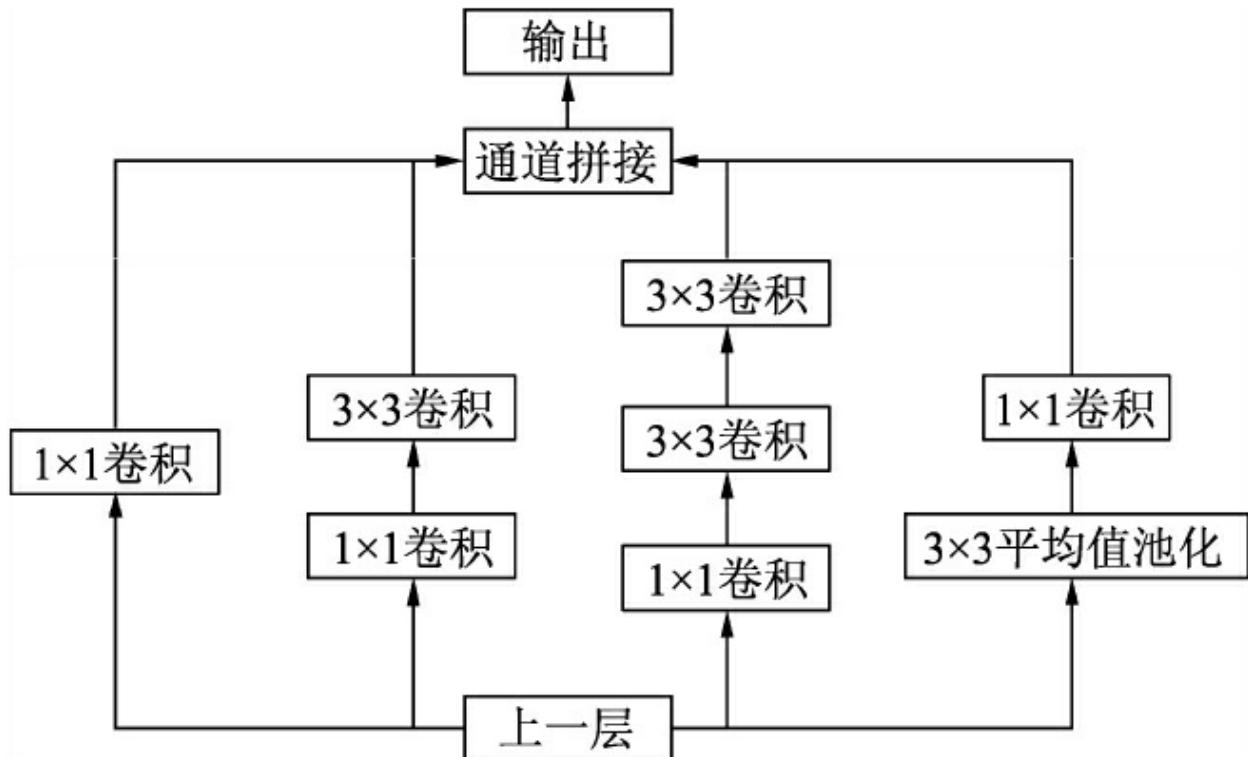


图3.15 Inception v2的基础模块结构

使用PyTorch来搭建一个单独的Inception v2模块， 默认输入的通道数为192， 新建一个inceptionv2.py文件， 代码如下：

---

```
import torch
from torch import nn
import torch.nn.functional as F
# 构建基础的卷积模块，与Inception v2的基础模块相比，增加了BN层
class BasicConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0):
        super(BasicConv2d, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding)
        self.bn = nn.BatchNorm2d(out_channels, eps=0.001)
    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        return F.relu(x, inplace=True)
class Inceptionv2(nn.Module):
```

```

def __init__(self):
    super(Inceptionv2, self).__init__()
    self.branch1 = BasicConv2d(192, 96, 1, 0) # 对应1x1卷积分支
    # 对应1x1卷积与3x3卷积分支
    self.branch2 = nn.Sequential(
        BasicConv2d(192, 48, 1, 0),
        BasicConv2d(48, 64, 3, 1)
    )
    # 对应1x1卷积、3x3卷积与3x3卷积分支
    self.branch3 = nn.Sequential(
        BasicConv2d(192, 64, 1, 0),
        BasicConv2d(64, 96, 3, 1),
        BasicConv2d(96, 96, 3, 1)
    )
    # 对应3x3平均池化与1x1卷积分支
    self.branch4 = nn.Sequential(
        nn.AvgPool2d(3, stride=1, padding=1, count_include_pad=False),
        BasicConv2d(192, 64, 1, 0)
    )
# 前向过程，将4个分支进行torch.cat()拼接起来
def forward(self, x):
    x0 = self.branch1(x)
    x1 = self.branch2(x)
    x2 = self.branch3(x)
    x3 = self.branch4(x)
    out = torch.cat((x0, x1, x2, x3), 1)
    return out

```

---

在终端中进入上述Inceptionv2.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用该模块。

---

```

>>> import torch
>>> from inceptionv2 import Inceptionv2
>>> net_inceptionv2 = Inceptionv2().cuda()
>>> net_inceptionv2
Inceptionv2(
    # 第1个分支，使用1x1卷积，输出通道数为96
    (branch1): BasicConv2d(
        (conv): Conv2d(192, 96, kernel_size=(1, 1), stride=(1, 1))
        (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_
            running_stats=True)
    )
    # 第2个分支，使用1x1卷积与3x3卷积，输出通道数为64
    (branch2): Sequential(

```

```
(0): BasicConv2d(
    (conv): Conv2d(192, 48, kernel_size=(1, 1), stride=(1, 1))
    (bn): BatchNorm2d(48, eps=0.001, momentum=0.1, affine=True, track_
        running_stats=True)
)
(1): BasicConv2d(
    (conv): Conv2d(48, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_
        running_stats=True)
)
)
# 第3个分支，使用1×1卷积与两个连续的3×3卷积，输出通道数为96
(branch3): Sequential(
    (0): BasicConv2d(
        (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1))
        (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_
            running_stats=True)
    )
    (1): BasicConv2d(
        (conv): Conv2d(64, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_
            running_stats=True)
    )
    (2): BasicConv2d(
        (conv): Conv2d(96, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (bn): BatchNorm2d(96, eps=0.001, momentum=0.1, affine=True, track_
            running_stats=True)
    )
)
)
# 第4个分支，使用平均池化与1×1卷积，输出通道数为64
(branch4): Sequential(
    (0): AvgPool2d(kernel_size=3, stride=1, padding=1)
    (1): BasicConv2d(
        (conv): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1))
        (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_
            running_stats=True)
    )
)
)
)
>>> input = torch.randn(1, 192, 32, 32).cuda()
>>> input.shape
torch.Size([1, 192, 32, 32])
>>> output = net_inceptionv2(input)      # 将输入传入实例的网络
>>> output.shape                      # 输出特征图的通道数为：96+64+96+64=320
torch.Size([1, 320, 32, 32])
```

---

更进一步，Inception v2将 $n \times n$ 的卷积运算分解为 $1 \times n$ 与 $n \times 1$ 两个卷积，如图3.16所示，这种分解的方式可以使计算成本降低33%。

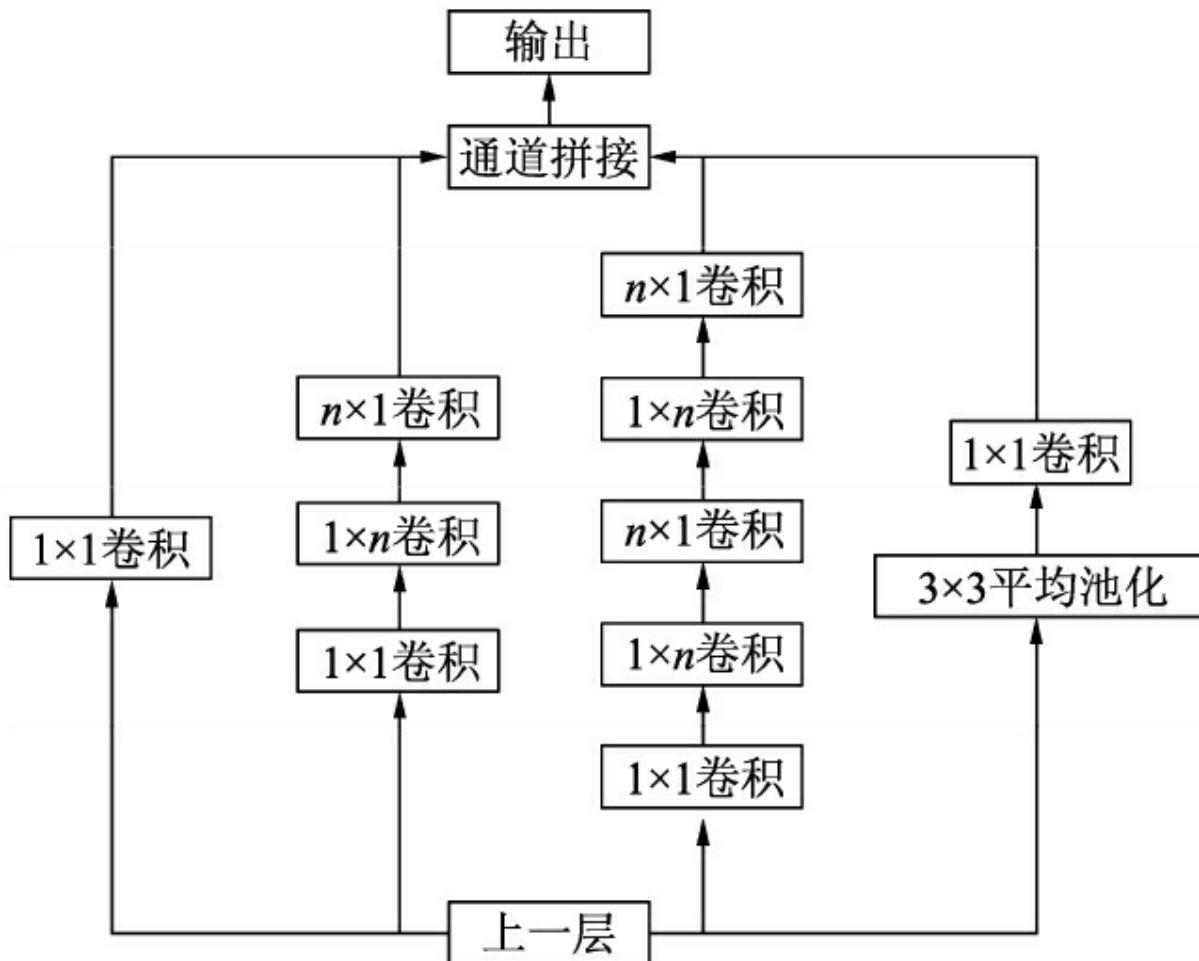


图3.16 改进的Inception v2模块结构

此外，Inception v2还将模块中的卷积核变得更宽而不是更深，形成第三个模块，以解决表征能力瓶颈的问题。Inception v2网络正是由上述的三种不同类型的模块组成的，其计算也更加高效。

Inception v3在Inception v2的基础上，使用了RMSProp优化器，在辅助的分类器部分增加了 $7 \times 7$ 的卷积，并且使用了标签平滑技术。

Inception v4则是将Inception的思想与残差网络进行了结合，显著提

升了训练速度与模型准确率，这里对于模块细节不再展开讲述。至于残差网络这一里程碑式的结构，正是由下一节的网络ResNet引出的。

## 3.4 里程碑：ResNet

VGGNet与Inception出现后，学者们将卷积网络不断加深以寻求更优越的性能，然而随着网络的加深，网络却越发难以训练，一方面会产生梯度消失现象；另一方面越深的网络返回的梯度相关性会越来越差，接近于白噪声，导致梯度更新也接近于随机扰动。

ResNet（Residual Network，残差网络）较好地解决了这个问题，并获得了2015年ImageNet分类任务的第一名。此后的分类、检测、分割等任务也大规模使用ResNet作为网络骨架。

ResNet的思想在于引入了一个深度残差框架来解决梯度消失问题，即让卷积网络去学习残差映射，而不是期望每一个堆叠层的网络都完整地拟合潜在的映射（拟合函数）。如图3.17所示，对于神经网络，如果我们期望的网络最终映射为 $H(x)$ ，左侧的网络需要直接拟合输出 $H(x)$ ，而右侧由ResNet提出的子模块，通过引入一个shortcut（捷径）分支，将需要拟合的映射变为残差 $F(x)$ ： $H(x)-x$ 。ResNet给出的假设是：相较于直接优化潜在映射 $H(x)$ ，优化残差映射 $F(x)$ 是更为容易的。

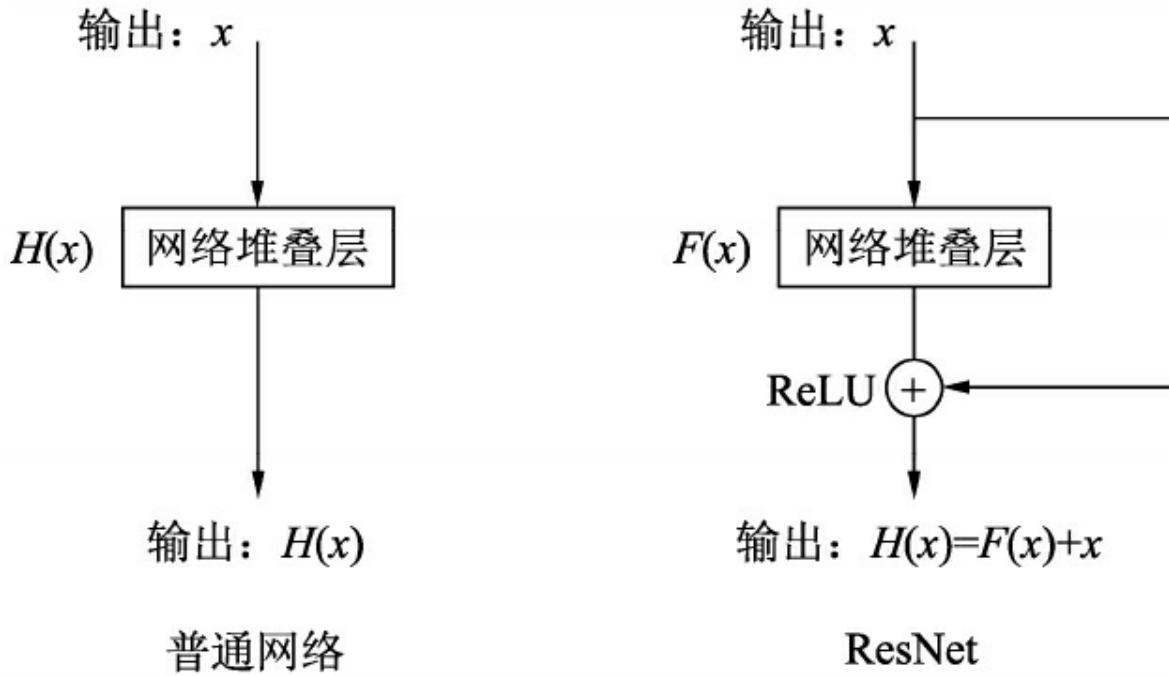


图3.17 普通网络与ResNet的残差映射

在ResNet中，上述的一个残差模块称为Bottleneck。ResNet有不同网络层数的版本，如18层、34层、50层、101层和152层，这里以常用的50层来讲解。ResNet-50的网络架构如图3.18所示，最主要的部分在于中间经历了4个大的卷积组，而这4个卷积组分别包含了3、4、6这3个Bottleneck模块。最后经过一个全局平均池化使得特征图大小变为 $1 \times 1$ ，然后进行1000维的全连接，最后经过Softmax输出分类得分。

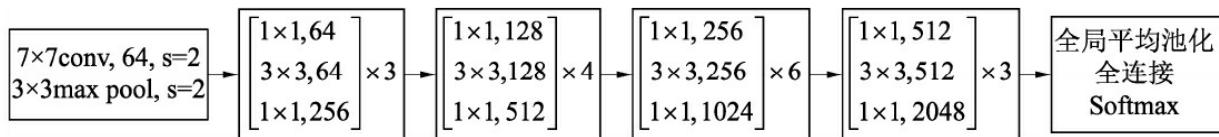


图3.18 ResNet-50网络结构图

由于 $F(x)+x$ 是逐通道进行相加，因此根据两者是否通道数相同，存在两种Bottleneck结构。对于通道数不同的情况，比如每个卷积组的第一个Bottleneck，需要利用 $1 \times 1$ 卷积对 $x$ 进行Downsample操作，将通道数

变为相同，再进行加操作。对于相同的情况下，两者可以直接进行相加。

利用PyTorch实现一个带有Downsample操作的Bottleneck结构，新建一个resnet\_bottleneck.py文件，代码如下：

---

```
import torch.nn as nn
class Bottleneck(nn.Module):
    def __init__(self, in_dim, out_dim, stride=1):
        super(Bottleneck, self).__init__()
        # 网路堆叠层是由1×1、3×3、1×1这3个卷积组成的，中间包含BN层
        self.bottleneck = nn.Sequential(
            nn.Conv2d(in_dim, in_dim, 1, bias=False),
            nn.BatchNorm2d(in_dim),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_dim, in_dim, 3, stride, 1, bias=False),
            nn.BatchNorm2d(in_dim),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_dim, out_dim, 1, bias=False),
            nn.BatchNorm2d(out_dim),
        )
        self.relu = nn.ReLU(inplace=True)
        # Downsample部分是由一个包含BN层的1×1卷积组成
        self.downsample = nn.Sequential(
            nn.Conv2d(in_dim, out_dim, 1, 1),
            nn.BatchNorm2d(out_dim),
        )
    def forward(self, x):
        identity = x
        out = self.bottleneck(x)
        identity = self.downsample(x)
        # 将identity（恒等映射）与网络堆叠层输出进行相加，并经过ReLU后输出
        out += identity
        out = self.relu(out)
        return out
```

---

在终端中进入上述resnet\_bottleneck.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用该模块。

---

```
>>> import torch
```

```
>>> from resnet_bottleneck import Bottleneck
# 实例化Bottleneck，输入通道数为64，输出为256，对应第一个卷积组的第一个Bottleneck
>>> bottleneck_1_1 = Bottleneck(64, 256).cuda()
>>> bottleneck_1_1
# Bottleneck作为卷积堆叠层，包含了1×1、3×3、1×1这3个卷积层
Bottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
            running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
            bias=False)
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
            stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
            running_stats=True)
    )
    (relu): ReLU(inplace)
    # 利用Downsample结构将恒等映射的通道数变为与卷积堆叠层相同，保证可以相加
    (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
            running_stats=True)
    )
)
>>> input = torch.randn(1, 64, 56, 56).cuda()
>>> output = bottleneck_1_1(input) # 将输入送到Bottleneck结构中
>>> input.shape
torch.Size([1, 64, 56, 56])
>>> output.shape
# 相比输入，输出的特征图分辨率没变，而通道数变为4倍
torch.Size([1, 256, 56, 56])
```

---

## 3.5 继往开来：DenseNet

上一节的ResNet通过前层与后层的“短路连接”（Shortcuts），加强了前后层之间的信息流通，在一定程度上缓解了梯度消失现象，从而可以将神经网络搭建得很深。更进一步，本节的主角DenseNet最大化了这种前后层信息交流，通过建立前面所有层与后面层的密集连接，实现了特征在通道维度上的复用，使其可以在参数与计算量更少的情况下实现比ResNet更优的性能，提出DenseNet的《Densely Connected Convolutional Networks》也一举拿下了2017年CVPR的最佳论文。

DenseNet的网络架构如图3.19所示，网络由多个Dense Block与中间的卷积池化组成，核心就在Dense Block中。Dense Block中的黑点代表一个卷积层，其中的多条黑线代表数据的流动，每一层的输入由前面的所有卷积层的输出组成。注意这里使用了通道拼接（Concatenate）操作，而非ResNet的逐元素相加操作。

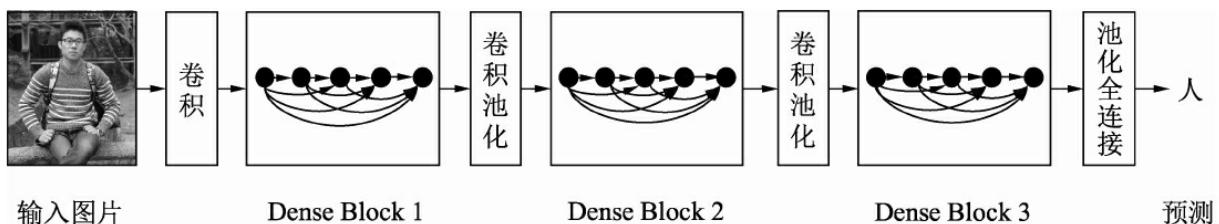


图3.19 DenseNet网络结构

DenseNet的结构有如下两个特性：

- 神经网络一般需要使用池化等操作缩小特征图尺寸来提取语义特征，而Dense Block需要保持每一个Block内的特征图尺寸一致来直接进行Concatenate操作，因此DenseNet被分成了多个Block。Block的数量一般为4。

·两个相邻的Dense Block之间的部分被称为Transition层，具体包括BN、ReLU、 $1\times 1$ 卷积、 $2\times 2$ 平均池化操作。 $1\times 1$ 卷积的作用是降维，起到压缩模型的作用，而平均池化则是降低特征图的尺寸，

具体的Block实现细节如图3.20所示，每一个Block由若干个Bottleneck的卷积层组成，对应图3.19中的黑点。Bottleneck由BN、ReLU、 $1\times 1$ 卷积、BN、ReLU、 $3\times 3$ 卷积的顺序构成。

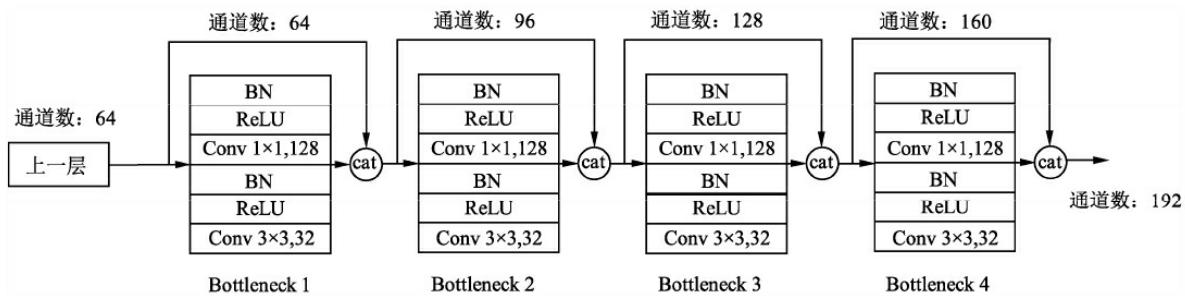


图3.20 DenseNet的Block结构

关于Block，有以下4个细节需要注意：

·每一个Bottleneck输出的特征通道数是相同的，例如这里的32。同时可以看到，经过Concatenate操作后的通道数是按32的增长量增加的，因此这个32也被称为GrowthRate。

·这里 $1\times 1$ 卷积的作用是固定输出通道数，达到降维的作用。当几十个Bottleneck相连接时，Concatenate后的通道数会增加到上千，如果不增加 $1\times 1$ 的卷积来降维，后续 $3\times 3$ 卷积所需的参数量会急剧增加。 $1\times 1$ 卷积的通道数通常是GrowthRate的4倍。

·图3.20中的特征传递方式是直接将前面所有层的特征Concatenate后传到下一层，这种方式与具体代码实现的方式是一致的，而不像图3.19中，前面层都要有一个箭头指向后面的所有层。

·Block采用了激活函数在前、卷积层在后的顺序，这与一般的网络上是不同的。

利用PyTorch来实现DenseNet的一个Block，新建一个densenet\_block.py文件，代码如下：

---

```
import torch
from torch import nn
import torch.nn.functional as F
# 实现一个Bottleneck的类，初始化需要输入通道数与GrowthRate这两个参数
class Bottleneck(nn.Module):
    def __init__(self, nChannels, growthRate):
        super(Bottleneck, self).__init__()
        # 通常1×1卷积的通道数为GrowthRate的4倍
        interChannels = 4*growthRate
        self.bn1 = nn.BatchNorm2d(nChannels)
        self.conv1 = nn.Conv2d(nChannels, interChannels, kernel_size=1,
                             bias=False)
        self.bn2 = nn.BatchNorm2d(interChannels)
        self.conv2 = nn.Conv2d(interChannels, growthRate, kernel_size=3,
                             padding=1, bias=False)
    def forward(self, x):
        out = self.conv1(F.relu(self.bn1(x)))
        out = self.conv2(F.relu(self.bn2(out)))
        # 将输入x同计算的结果out进行通道拼接
        out = torch.cat((x, out), 1)
        return out
class Denseblock(nn.Module):
    def __init__(self, nChannels, growthRate, nDenseBlocks):
        super(Denseblock, self).__init__()
        layers = []
        # 将每一个Bottleneck利用nn.Sequential()整合起来，输入通道数需要线性增长
        for i in range(int(nDenseBlocks)):
            layers.append(Bottleneck(nChannels, growthRate))
            nChannels += growthRate
        self.denseblock = nn.Sequential(*layers)
    def forward(self, x):
        return self.denseblock(x)
```

---

在终端中进入上述densenet\_block.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用该模块。

```
>>> import torch
>>> from densenet_block import Denseblock
# 实例化DenseBlock，包含了6个Bottleneck
>>> denseblock = Denseblock(64, 32, 6).cuda()
# 查看denseblock的网络结构，由6个Bottleneck组成
>>> denseblock
Denseblock(
(denseblock): Sequential(
    # 第1个Bottleneck的输入通道数为64，输出固定为32
    (0): Bottleneck(
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
        (conv1): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias
=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
    )
    # 第2个Bottleneck的输入通道数为96，输出固定为32
    (1): Bottleneck(
        (bn1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
        (conv1): Conv2d(96, 128, kernel_size=(1, 1), stride=(1, 1), bias
=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
    )
    # 第3个Bottleneck的输入通道数为128，输出固定为32
    (2): Bottleneck(
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (conv1): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias
=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1), bias=False)
    )
    # 第4个Bottleneck的输入通道数为160，输出固定为32
    (3): Bottleneck(
        (bn1): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (conv1): Conv2d(160, 128, kernel_size=(1, 1), stride=(1, 1), bias
=False)
    )
)
```

```

        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
                           track_running_stats=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding
                       =(1, 1), bias=False)
    )
# 第5个Bottleneck的输入通道数为192，输出固定为32
(4): Bottleneck(
    (bn1): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True,
                       track_running_stats=True)
    (conv1): Conv2d(192, 128, kernel_size=(1, 1), stride=(1, 1), bias
                   =False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
                       track_running_stats=True)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding
                   =(1, 1), bias=False)
)
# 第6个Bottleneck的输入通道数为224，输出固定为32
(5): Bottleneck(
    (bn1): BatchNorm2d(224, eps=1e-05, momentum=0.1, affine=True,
                       track_running_stats=True)
    (conv1): Conv2d(224, 128, kernel_size=(1, 1), stride=(1, 1), bias
                   =False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
                       track_running_stats=True)
    (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding
                   =(1, 1), bias=False)
)
)
)
)
>>> input = torch.randn(1, 64, 256, 256).cuda()
>>> output = denseblock(input) # 将输入传入denseblock结构中
# 输出的通道数为: 224+32=64+32×6=256
>>> output.shape
torch.Size([1, 256, 256, 256])

```

---

DenseNet网络的优势主要体现在以下两个方面：

- 密集连接的特殊网络，使得每一层都会接受其后所有层的梯度，而不是像普通卷积链式的反传，因此一定程度上解决了梯度消失的问题。

- 通过Concatenate操作使得大量的特征被复用，每个层独有的特征图

的通道是较少的，因此相比ResNet， DenseNet参数更少且计算更高效。

DenseNet的不足在于由于需要进行多次Concatenate操作，数据需要被复制多次，显存容易增加得很快，需要一定的显存优化技术。另外，DenseNet是一种更为特殊的网络，ResNet则相对一般化一些，因此ResNet的应用范围更广泛。

## 3.6 特征金字塔：FPN

为了增强语义性，传统的物体检测模型通常只在深度卷积网络的最后一个特征图上进行后续操作，而这一层对应的下采样率（图像缩小的倍数）通常又比较大，如16、32，造成小物体在特征图上的有效信息较少，小物体的检测性能会急剧下降，这个问题也被称为多尺度问题。

解决多尺度问题的关键在于如何提取多尺度的特征。传统的方法有图像金字塔（Image Pyramid），主要思路是将输入图片做成多个尺度，不同尺度的图像生成不同尺度的特征，这种方法简单而有效，大量使用在了COCO等竞赛上，但缺点是非常耗时，计算量也很大。

从前面几节内容可以知道，卷积神经网络不同层的大小与语义信息不同，本身就类似一个金字塔结构。2017年的FPN（Feature Pyramid Network）方法融合了不同层的特征，较好地改善了多尺度检测问题。

FPN的总体架构如图3.21所示，主要包含自下而上网络、自上而下网络、横向连接与卷积融合4个部分。

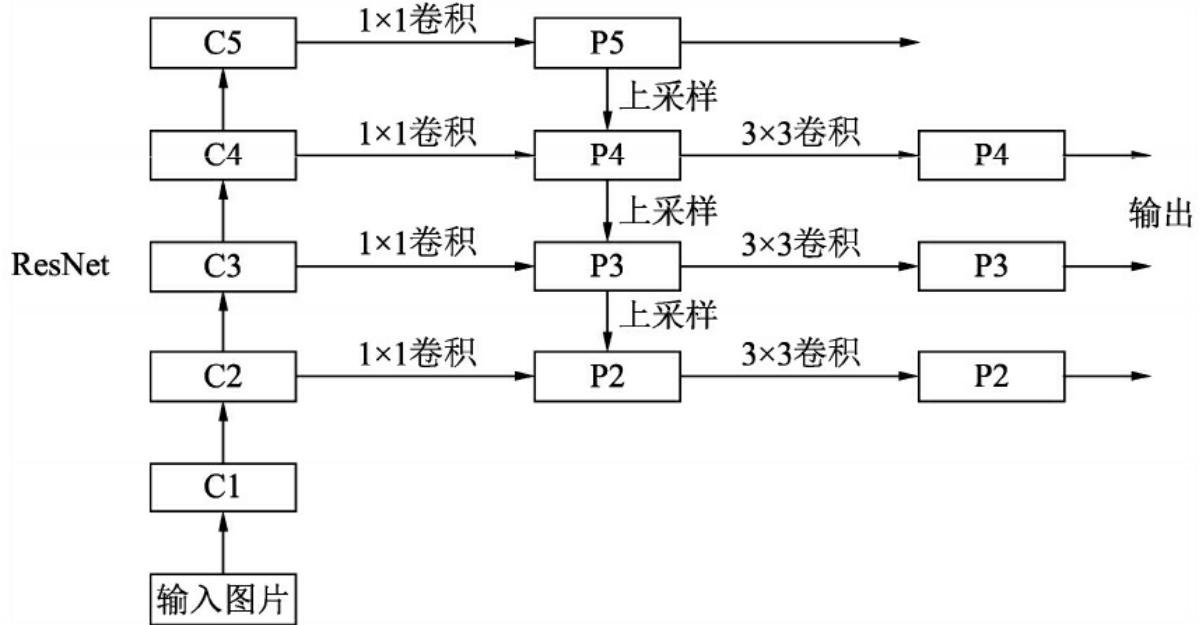


图3.21 FPN网络结构

**·自下而上：**最左侧为普通的卷积网络，默认使用ResNet结构，用作提取语义信息。C1代表了ResNet的前几个卷积与池化层，而C2至C5分别为不同的ResNet卷积组，这些卷积组包含了多个Bottleneck结构，组内的特征图大小相同，组间大小递减。

**·自上而下：**首先对C5进行 $1 \times 1$ 卷积降低通道数得到P5，然后依次进行上采样得到P4、P3和P2，目的是得到与C4、C3与C2长宽相同的特征，以方便下一步进行逐元素相加。这里采用2倍最邻近上采样，即直接对临近元素进行复制，而非线性插值。

**·横向连接（Lateral Connection）：**目的是为了将上采样后的高语义特征与浅层的定位细节特征进行融合。高语义特征经过上采样后，其长宽与对应的浅层特征相同，而通道数固定为256，因此需要对底层特征C2至C4进行 $1 \times 1$ 卷积使得其通道数变为256，然后两者进行逐元素相加得到P4、P3与P2。由于C1的特征图尺寸较大且语义信息不足，因此没有把C1放到横向连接中。

**·卷积融合：**在得到相加后的特征后，利用 $3 \times 3$ 卷积对生成的P2至P4再进行融合，目的是消除上采样过程带来的重叠效应，以生成最终的特征图。

对于实际的物体检测算法，需要在特征图上进行RoI（Region of Interests，感兴趣区域）提取，而FPN有4个输出的特征图，选择哪一个特征图上面的特征也是个问题。**FPN给出的解决方法是，对于不同大小的RoI，使用不同的特征图，大尺度的RoI在深层的特征图上进行提取，如P5，小尺度的RoI在浅层的特征图上进行提取，如P2，具体确定方法，感兴趣的读者可以自行查看。**

**FPN将深层的语义信息传到底层，来补充浅层的语义信息，从而获得了高分辨率、强语义的特征，在小物体检测、实例分割等领域有着非常不俗的表现。**

使用PyTorch来搭建一个完整的FPN网络，新建一个fpn.py，代码如下：

---

```
import torch.nn as nn
import torch.nn.functional as F
import math
# ResNet的基本Bottleneck类
class Bottleneck(nn.Module):
    expansion = 4 # 通道倍增数
    def __init__(self, in_planes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.bottleneck = nn.Sequential(
            nn.Conv2d(in_planes, planes, 1, bias=False),
            nn.BatchNorm2d(planes),
            nn.ReLU(inplace=True),
            nn.Conv2d(planes, planes, 3, stride, 1, bias=False),
            nn.BatchNorm2d(planes),
            nn.ReLU(inplace=True),
            nn.Conv2d(planes, self.expansion * planes, 1, bias=False),
            nn.BatchNorm2d(self.expansion * planes),
        )
        self.relu = nn.ReLU(inplace=True)
```

```

    self.downsample = downsample
def forward(self, x):
    identity = x
    out = self.bottleneck(x)
    if self.downsample is not None:
        identity = self.downsample(x)
    out += identity
    out = self.relu(out)
    return out
# FPN的类，初始化需要一个list，代表ResNet每一个阶段的Bottleneck的数量
class FPN(nn.Module):
    def __init__(self, layers):
        super(FPN, self).__init__()
        self.inplanes = 64
        # 处理输入的C1模块
        self.conv1 = nn.Conv2d(3, 64, 7, 2, 3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(3, 2, 1)
        # 搭建自下而上的C2、C3、C4、C5
        self.layer1 = self._make_layer(64, layers[0])
        self.layer2 = self._make_layer(128, layers[1], 2)
        self.layer3 = self._make_layer(256, layers[2], 2)
        self.layer4 = self._make_layer(512, layers[3], 2)
        # 对C5减少通道数，得到 P5
        self.toplayer = nn.Conv2d(2048, 256, 1, 1, 0)
        # 3x3卷积融合特征
        self.smooth1 = nn.Conv2d(256, 256, 3, 1, 1)
        self.smooth2 = nn.Conv2d(256, 256, 3, 1, 1)
        self.smooth3 = nn.Conv2d(256, 256, 3, 1, 1)
        # 横向连接，保证通道数相同
        self.latlayer1 = nn.Conv2d(1024, 256, 1, 1, 0)
        self.latlayer2 = nn.Conv2d(512, 256, 1, 1, 0)
        self.latlayer3 = nn.Conv2d(256, 256, 1, 1, 0)
    # 构建C2到C5，注意区分stride值为1和2的情况
    def _make_layer(self, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != Bottleneck.expansion * planes:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, Bottleneck.expansion * planes, 1,
                         stride, bias=False),
                nn.BatchNorm2d(Bottleneck.expansion * planes)
            )
        layers = []
        layers.append(Bottleneck(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * Bottleneck.expansion
        for i in range(1, blocks):
            layers.append(Bottleneck(self.inplanes, planes))

```

```

        return nn.Sequential(*layers)
    # 自上而下的上采样模块
    def _upsample_add(self, x, y):
        _,_,H,W = y.shape
        return F.upsample(x, size=(H,W), mode='bilinear') + y
    def forward(self, x):
        # 自下而上
        c1 = self.maxpool(self.relu(self.bn1(self.conv1(x))))
        c2 = self.layer1(c1)
        c3 = self.layer2(c2)
        c4 = self.layer3(c3)
        c5 = self.layer4(c4)
        # 自上而下
        p5 = self.toplayer(c5)
        p4 = self._upsample_add(p5, self.latlayer1(c4))
        p3 = self._upsample_add(p4, self.latlayer2(c3))
        p2 = self._upsample_add(p3, self.latlayer3(c2))
        # 卷积融合, 平滑处理
        p4 = self.smooth1(p4)
        p3 = self.smooth2(p3)
        p2 = self.smooth3(p2)
        return p2, p3, p4, p5

```

---

在终端中进入上述fpn.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用上述的FDN模块。

---

```

>>> import torch
>>> from fpn import FPN
# 利用list来初始化FPN网络
>>> net_fpn = FPN([3, 4, 6, 3]).cuda()
>>> net_fpn.conv1                               # 查看FPN的第一个卷积层
Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
>>> net_fpn.bn1                                # 查看FPN的第一个BN层
BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
>>> net_fpn.relu                                 # 查看FPN的第一个ReLU层
ReLU(inplace)
>>> net_fpn.maxpool                            # 查看FPN的第一个池化层, 使用最大值池化
MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
>>> net_fpn.layer1      # 查看FPN的第一个layer, 即前面的C2, 包含了3个Bottleneck
Sequential(
    # layer1中第1个Bottleneck模块
    (0): Bottleneck(
        (bottleneck): Sequential(
            (0): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)

```

```
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
(4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
(5): ReLU(inplace)
(6): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
)
# 这里存在一个通道增加模块
(relu): ReLU(inplace)
(downsample): Sequential(
    (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
)
)
# layer1中第2个Bottleneck模块
(1): Bottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
    )
    (relu): ReLU(inplace)
)
)
# layer1中第3个Bottleneck模块
(2): Bottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
_running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
```

```
_running_stats=True)
(5): ReLU(inplace)
(6): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
    _running_stats=True)
)
(relu): ReLU(inplace)
)
)
>>> net_fpn.layer2      # 查看fpn的layer2, 即上面的C3, 包含了4个Bottleneck
Sequential(
# layer2中第1个Bottleneck
(0): Bottleneck(
    bottleneck): Sequential(
        (0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
            bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
    )
    (relu): ReLU(inplace)
    (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
    )
)
# layer2中第2个Bottleneck
(1): Bottleneck(
    bottleneck): Sequential(
        (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
            bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
    )
)
```

```

        _running_stats=True)
    )
    (relu): ReLU(inplace)
)
# layer2中第3个Bottleneck
(2): Bottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
    )
    (relu): ReLU(inplace)
)
# layer2中第4个Bottleneck
(3): Bottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
        (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track
            _running_stats=True)
    )
    (relu): ReLU(inplace)
)
)
>>> net_fpn.toplayer # 1x1的卷积, 以得到P5
Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
>>> net_fpn.smooth1 # 对P4进行平滑的卷积层
Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
>>> net_fpn.latlayer1 # 对C4进行横向处理的卷积层
Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
>>> input = torch.randn(1, 3, 224, 224).cuda()

```

```
>>> output = fpn(input)
# 返回的P2、P3、P4、P5，这4个特征图通道数相同，但特征图尺寸递减
>>> output[0].shape
torch.Size([1, 256, 56, 56])
>>> output[1].shape
torch.Size([1, 256, 28, 28])
>>> output[2].shape
torch.Size([1, 256, 14, 14])
>>> output[3].shape
torch.Size([1, 256, 7, 7])
```

---

## 3.7 为检测而生：DetNet

前面几节的网络骨架，如VGGNet和ResNet等，虽从各个角度出发提升了物体检测性能，但究其根本是为ImageNet的图像分类任务而设计的。而图像分类与物体检测两个任务天然存在着落差，分类任务侧重于全图的特征提取，深层的特征图分辨率很低；而物体检测需要定位出物体位置，特征图分辨率不宜过小，因此造成了以下两种缺陷：

·大物体难以定位：对于FPN等网络，大物体对应在较深的特征图上检测，由于网络较深时下采样率较大，物体的边缘难以精确预测，增加了回归边界的难度。

·小物体难以检测：对于传统网络，由于下采样率大造成小物体在较深的特征图上几乎不可见；FPN虽从较浅的特征图来检测小物体，但浅层的语义信息较弱，且融合深层特征时使用的上采样操作也会增加物体检测的难度。

针对以上问题，旷视科技提出了专为物体检测设计的DetNet结构，引入了空洞卷积，使得模型兼具较大感受野与较高分辨率，同时避免了3.6节中FPN的多次上采样，实现了较好的检测效果。

DetNet的网络结构如图3.22所示，仍然选择性能优越的ResNet-50作为基础结构，并保持前4个stage与ResNet-50相同，具体的结构细节有以下3点：

·引入了一个新的Stage 6，用于物体检测。Stage 5与Stage 6使用了DetNet提出的Bottleneck结构，最大的特点是利用空洞数为2的3×3卷积取代了步长为2的3×3卷积。

·Stage 5与Stage 6的每一个Bottleneck输出的特征图尺寸都为原图的 $\frac{1}{16}$ ，通道数都为256，而传统的Backbone通常是特征图尺寸递减，通道数递增。

·在组成特征金字塔时，由于特征图大小完全相同，因此可以直接从右向左传递相加，避免了上一节的上采样操作。为了进一步融合各通道的特征，需要对每一个阶段的输出进行 $1\times 1$ 卷积后再与后一Stage传回的特征相加。

DetNet这种精心设计的结构，在增加感受野的同时，获得了较大的特征图尺寸，有利于物体的定位。与此同时，由于各Stage的特征图尺寸相同，避免了上一节的上采样，既一定程度上降低了计算量，又有利  
于小物体的检测。

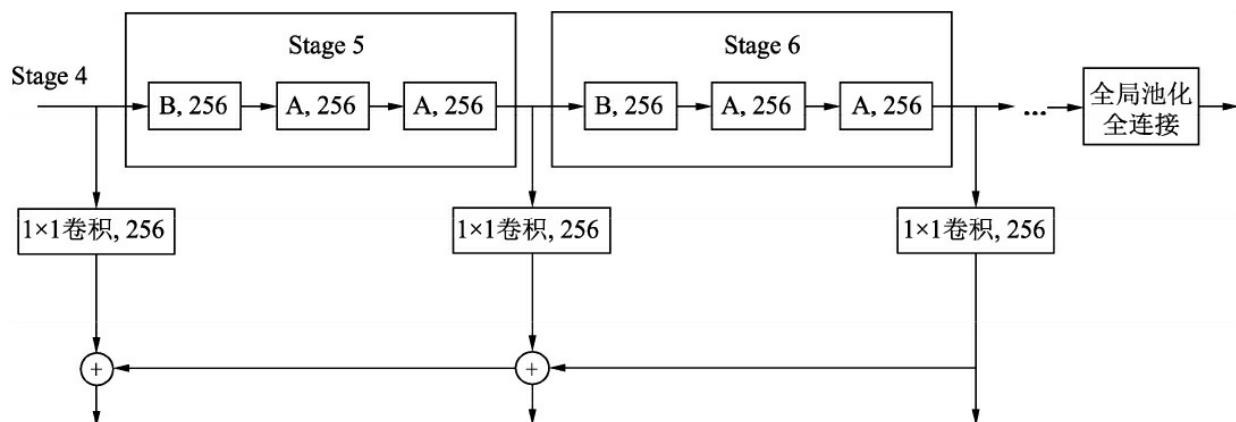


图3.22 DetNet网络结构

DetNet中Bottleneck的细节如图3.23所示，左侧的两个Bottleneck A与Bottleneck B分别对应图3.22的A与B，右侧的为原始的ResNet残差结构。DetNet与ResNet两者的基本思想都是卷积堆叠层与恒等映射的相加，区别在于DetNet使用了空洞数为2的 $3\times 3$ 卷积，这样使得特征图尺寸保持不变，而ResNet是使用了步长为2的 $3\times 3$ 卷积。B相比于A，在恒等映射部分增加了一个 $1\times 1$ 卷积，这样做可以区分开不同的Stage，并且实

验发现这种做法对于特征金字塔式的检测非常重要。

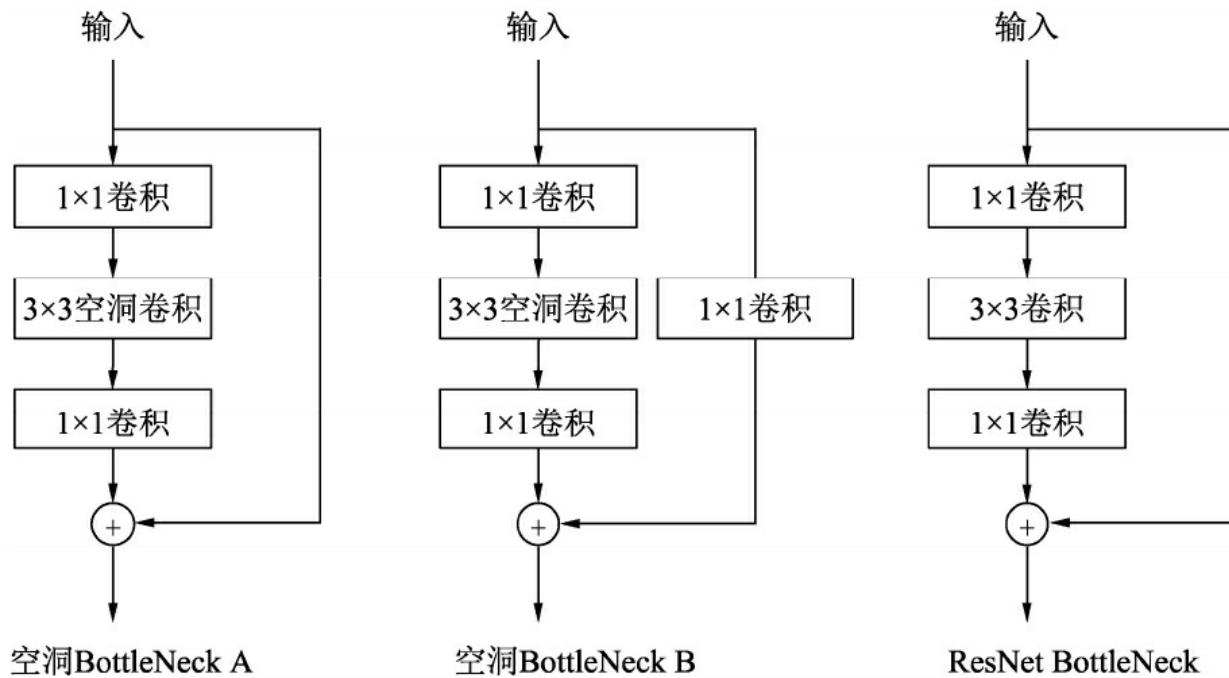


图3.23 DetNet与ResNet的残差对比

使用PyTorch来实现DetNet的两个Bottleneck结构A和B，新建一个detnet\_bottleneck.py文件，代码如下：

```
from torch import nn
class DetBottleneck(nn.Module):
    # 初始化时extra为False时为Bottleneck A, 为True时则为Bottleneck B
    def __init__(self, inplanes, planes, stride=1, extra=False):
        super(DetBottleneck, self).__init__()
        # 构建连续3个卷积层的Bottleneck
        self.bottleneck = nn.Sequential(
            nn.Conv2d(inplanes, planes, 1, bias=False),
            nn.BatchNorm2d(planes),
            nn.ReLU(inplace=True),
            nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=2,
                     dilation=2, bias=False),
            nn.BatchNorm2d(planes),
            nn.ReLU(inplace=True),
            nn.Conv2d(planes, planes, 1, bias=False),
            nn.BatchNorm2d(planes),
        )
        if extra:
            self.extra = nn.Sequential(
                nn.Conv2d(inplanes, planes, 1, bias=False),
                nn.BatchNorm2d(planes),
            )
```

```

        self.relu = nn.ReLU(inplace=True)
        self.extra = extra
        # Bottleneck B的1x1卷积
        if self.extra:
            self.extra_conv = nn.Sequential(
                nn.Conv2d(inplanes, planes, 1, bias=False),
                nn.BatchNorm2d(planes)
            )
    def forward(self, x):
        # 对于Bottleneck B来讲，需要对恒等映射增加卷积处理，与ResNet类似
        if self.extra:
            identity = self.extra_conv(x)
        else:
            identity = x
        out = self.bottleneck(x)
        out += identity
        out = self.relu(out)
        return out

```

---

在终端中进入上述detnet\_bottleneck.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用上述DetNet网络的Bottleneck结构：

---

```

>>> import torch
>>> from detnet_bottleneck import DetBottleneck
# 完成一个Stage 5，即B-A-A的结构，Stage 4输出通道数为1024
>>> bottleneck_b = DetBottleneck(1024, 256, 1, True).cuda()
>>> bottleneck_b           # 查看Bottleneck B的结构，带有extra的卷积层
DetBottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
              _running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
                   dilation=(2, 2), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
              _running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track
              _running_stats=True)
    )
    (relu): ReLU(inplace)

```

```

(extra_conv): Sequential(
    (0): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
)
>>> bottleneck_a1 = DetBottleneck(256, 256).cuda()
>>> bottleneck_a1      # 查看Bottleneck A1的结构
DetBottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
dilation=(2, 2), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (relu): ReLU(inplace)
)
>>> bottleneck_a2 = DetBottleneck(256, 256).cuda()
>>> bottleneck_a2      # 查看Bottleneck A2的结构, 与Bottleneck A1相同
DetBottleneck(
    (bottleneck): Sequential(
        (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace)
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2),
dilation=(2, 2), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace)
        (6): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (relu): ReLU(inplace)
)
>>> input = torch.randn(1, 1024, 14, 14).cuda()
# 将input作为某一层的特征图, 依次传入Bottleneck B、A1与A2三个模块
>>> output1 = bottleneck_b(input)
>>> output2 = bottleneck_a1(output1)
>>> output3 = bottleneck_a2(output2)

```

```
# 三个Bottleneck输出的特征图大小完全相同
>>> output1.shape, output2.shape, output3.shape
(torch.Size([1, 256, 14, 14]), torch.Size([1, 256, 14, 14]), torch.Size([1, 256,
14, 14]))
```

---

## 3.8 总结

作为物体检测技术的基础部分，网络Backbone发展迅速，涌现出了众多优秀的网络结构，如VGGNet和ResNet等。在本章中，笔者利用PyTorch从代码角度讲解了每一个基础卷积结构的搭建方法，希望读者能够领悟每一个结构背后的出发点及创新点，这对于深度学习的学习是至关重要的。

接下来，我们就可以接触完整的物体检测算法了。在下一章中，我们将学习物体检测领域大名鼎鼎的算法：Faster RCNN。

## 第2篇 物体检测经典框架

·第4章 两阶经典检测器：Faster RCNN

·第5章 单阶多层检测器：SSD

·第6章 单阶经典检测器：YOLO

## 第4章 两阶经典检测器：Faster RCNN

RCNN全称为Regions with CNN Features，是将深度学习应用到物体检测领域的经典之作，并凭借卷积网络出色的特征提取能力，大幅度提升了物体检测的效果。而随后基于RCNN的Fast RCNN及Faster RCNN将物体检测问题进一步优化，在实现方式、速度、精度上均有了大幅度提升。

物体检测领域出现的新成果很大一部分也是基于RCNN系列的思想，尤其是Faster RCNN，并且在解决小物体、拥挤等较难任务时，RCNN系列仍然具有较强的优势。因此，想要学习物体检测，RCNN系列是第一个需要全面掌握的算法。

本章首先简要介绍RCNN系列算法的发展历程，然后针对Faster RCNN从宏观角度探索该算法的设计思想与逻辑，从代码角度详细讲述每一个模块的细节与实现方法，让你知其然且知其所以然，并能启发你的思考。

## 4.1 RCNN系列发展历程

在2014年RCNN算法问世之后，经历了众多版本的改进，但具有里程碑式意义的当属Fast RCNN与Faster RCNN算法。本节主要讲述这3种算法的思想、优点及存在的问题。

### 4.1.1 开山之作：RCNN

RCNN算法由Ross Girshick等人发表在CVPR 2014，将卷积神经网络应用于特征提取，并借助于CNN良好的特征提取性能，一举将PASCAL VOC数据集的检测率从35.1%提升到了53.7%。

RCNN算法流程如图4.1所示，RCNN仍然延续传统物体检测的思想，将物体检测当做分类问题处理，即先提取一系列的候选区域，然后对候选区域进行分类。

具体过程主要包含4步：

(1) 候选区域生成。采用Region Proposal提取候选区域，例如Selective Search算法，先将图像分割成小区域，然后合并包含同一物体可能性高的区域，并输出，在这一步需要提取约2000个候选区域。在提取完后，还需要将每一个区域进行归一化处理，得到固定大小的图像。

(2) CNN特征提取。将上述固定大小的图像，利用CNN网络得到固定维度的特征输出。

(3) SVM分类器。使用线性二分类器对输出的特征进行分类，得到是否属于此类的结果，并采用难样本挖掘来平衡正负样本的不平衡。

(4) 位置精修。通过一个回归器，对特征进行边界回归以得到更为精确的目标区域。

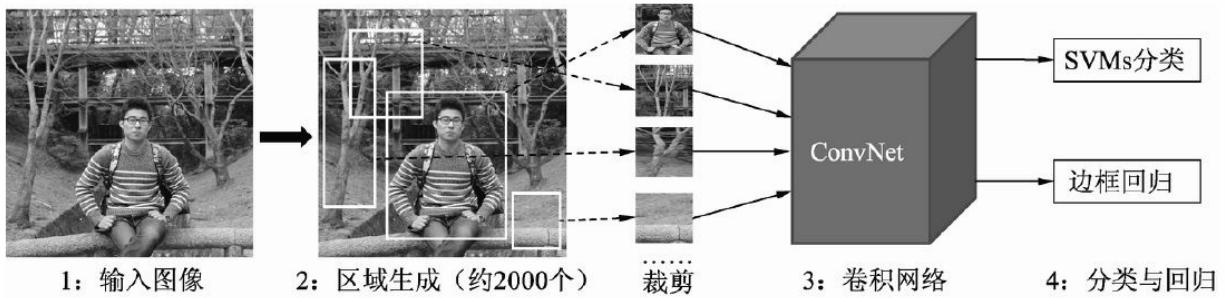


图4.1 RCNN算法流程图

RCNN虽然显著提升了物体检测的效果，但仍存在3个较大的问题。首先RCNN需要多步训练，步骤繁琐且训练速度较慢；其次，由于涉及分类中的全连接网络，因此输入尺寸是固定的，造成了精度的降低；最后，候选区域需要提前提取并保存，占用空间较大。

## 4.1.2 端到端：Fast RCNN

在RCNN之后，SPPNet算法解决了重复卷积计算与固定输出尺度的两个问题，但仍然存在RCNN的其他弊端。在2015年，Ross Girshick独自提出了更快、更强的Fast RCNN算法，不仅训练的步骤可以实现端到端，而且算法基于VGG16网络，在训练速度上比RCNN快了近9倍，在测试速度上快了213倍，并在VOC 2012数据集上达到了68.4%的检测率。

Fast RCNN算法框架图如图4.2所示，相比起RCNN，主要有3点改进：

·共享卷积：将整幅图送到卷积网络中进行区域生成，而不是像RCNN那样一个个的候选区域，虽然仍采用Selective Search方法，但共享卷积的优点使得计算量大大减少。

·RoI Pooling：利用特征池化（RoI Pooling）的方法进行特征尺度变换，这种方法可以有任意大小图片的输入，使得训练过程更加灵活、准确。

·多任务损失：将分类与回归网络放到一起训练，并且为了避免SVM分类器带来的单独训练与速度慢的缺点，使用了Softmax函数进行分类。

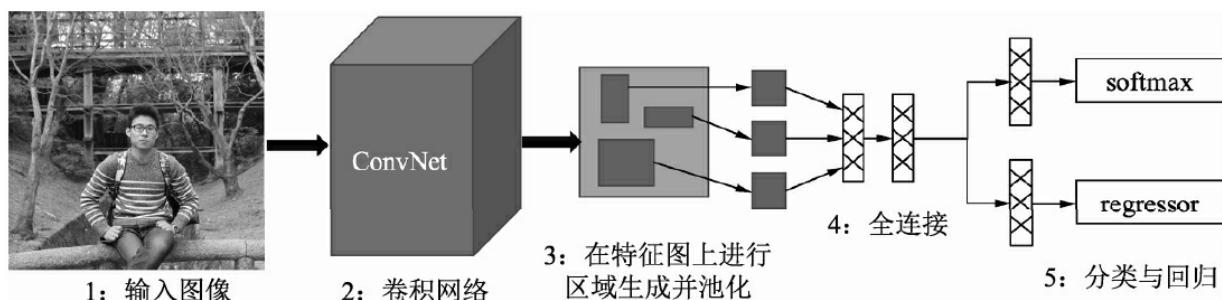


图4.2 Fast RCNN算法流程图

Fast RCNN算法虽然取得了显著的成果，但在该算法中，Selective Search需要消耗2~3秒，而特征提取仅需要0.2秒，因此这种区域生成方法限制了Fast RCNN算法的发挥空间，这也为后来的Faster RCNN算法提供了改进方向。

### 4.1.3 走向实时：Faster RCNN

Faster RCNN算法发表于NIPS 2015，该算法最大的创新点在于提出了RPN（Region Proposal Network）网络，利用Anchor机制将区域生成与卷积网络联系到一起，将检测速度一举提升到了17 FPS（Frames Per Second），并在VOC 2012测试集上实现了70.4%的检测结果。

Anchor可以看做是图像上很多固定大小与宽高的方框，由于需要检测的物体本身也都是一一个个大小宽高不同的方框，因此Faster RCNN将Anchor做强先验的知识，接下来只需要将Anchor与真实物体进行匹配，进行分类与位置的微调即可。相比起没有Anchor的物体检测算法，这样的先验无疑降低了网络收敛的难度，再加上一系列的工程优化，使得Faster RCNN达到了物体检测中的一个高峰。

只谈思想不谈代码实现，终究是纸上谈兵，因此本章接下来将会针对Faster RCNN，从代码角度全面讲解每一步的思想与实现过程，探索该算法的精妙与乐趣。

## 4.2 准备工作

本章代码所使用的环境在第1章已有说明，如果还没配置好PyTorch环境的读者，还请前往第1章按相关说明进行配置。本章使用的Faster RCNN源码来自于GitHub上jwyang的高质量复现，笔者迁移到了本书的代码中，使用方式具体如下：

(1) 首先从GitHub上下载本章所用的代码，地址如下：

```
git clone git@github.com:dongdonghy/Detection-PyTorch-Notebook.git
```

(2) 然后创建data文件夹。

```
cd Detection-PyTorch-Notebook/chapter4/faster-rcnn-pytorch && mkdir data
```

(3) 下载PASCAL VOC 2012数据集，在data文件夹下创建软连接。

```
cd data && ln -s "your data directory" VOC2012
```

(4) 从下面网址下载预训练模型并放到data/pretrained\_model/文件夹下。

```
https://www.dropbox.com/s/s3brpk0bdq60nyb/vgg16\_caffe.pth?dl=0
```

(5) 安装所需依赖包。

```
pip install -r requirements.txt
```

---

(6) 根据自己的GPU型号，修改lib/make.sh文件中的CUDA\_ARCH。

(7) 由于在NMS与RoI Pooling中使用了CUDA进行加速，因此还需要对此进行编译。

---

```
cd lib && sh make.sh
```

---

在运行程序的过程中，得益于PyTorch动态图的特性，可以在任何地方使用pdb工具插入到代码块中进行debug，理解每一步网络的数据变化。 接下来开始Faster RCNN学习之旅！

## 4.3 Faster RCNN总览

如图4.3所示为Faster RCNN算法的基本流程，从功能模块来讲，主要包括4部分：特征提取网络、RPN模块、RoI Pooling（Region of Interest）模块与RCNN模块，虚线表示仅仅在训练时有的步骤。Faster RCNN延续了RCNN系列的思想，即先进行感兴趣区域RoI的生成，然后再把生成的区域分类，最后完成物体的检测，这里的RoI使用的即是RPN模块，区域分类则是RCNN网络。

特征提取网络Backbone：输入图像首先经过Backbone得到特征图，在此以VGGNet为例，假设输入图像的维度为 $3 \times 600 \times 800$ ，由于VGGNet包含4个Pooling层（物体检测使用VGGNet时，通常不使用第5个Pooling层），下采样率为16，因此输出的feature map的维度为 $512 \times 37 \times 50$ 。

RPN模块：区域生成模块，如图4.3的中间部分，其作用是生成较好的建议框，即Proposal，这里用到了强先验的Anchor。RPN包含5个子模块：

·Anchor生成：RPN对feature map上的每一个点都对应了9个Anchors，这9个Anchors大小宽高不同，对应到原图基本可以覆盖所有可能出现的物体。因此，有了数量庞大的Anchors，RPN接下来的工作就是从中筛选，并调整出更好的位置，得到Proposal。

·RPN卷积网络：与上面的Anchor对应，由于feature map上每个点对应了9个Anchors，因此可以利用 $1 \times 1$ 的卷积在feature map上得到每一个Anchor的预测得分与预测偏移值。

·计算RPN loss：这一步只在训练中，将所有的Anchors与标签进行匹配，匹配程度较好的Anchors赋予正样本，较差的赋予负样本，得到

分类与偏移的真值，与第二步中的预测得分与预测偏移值进行loss的计算。

·生成Proposal：利用第二步中每一个Anchor预测的得分与偏移量，可以进一步得到一组较好的Proposal，送到后续网络中。

·筛选Proposal得到RoI：在训练时，由于Proposal数量还是太多（默认是2000），需要进一步筛选Proposal得到RoI（默认数量是256）。在测试阶段，则不需要此模块，Proposal可以直接作为RoI，默认数量为300。

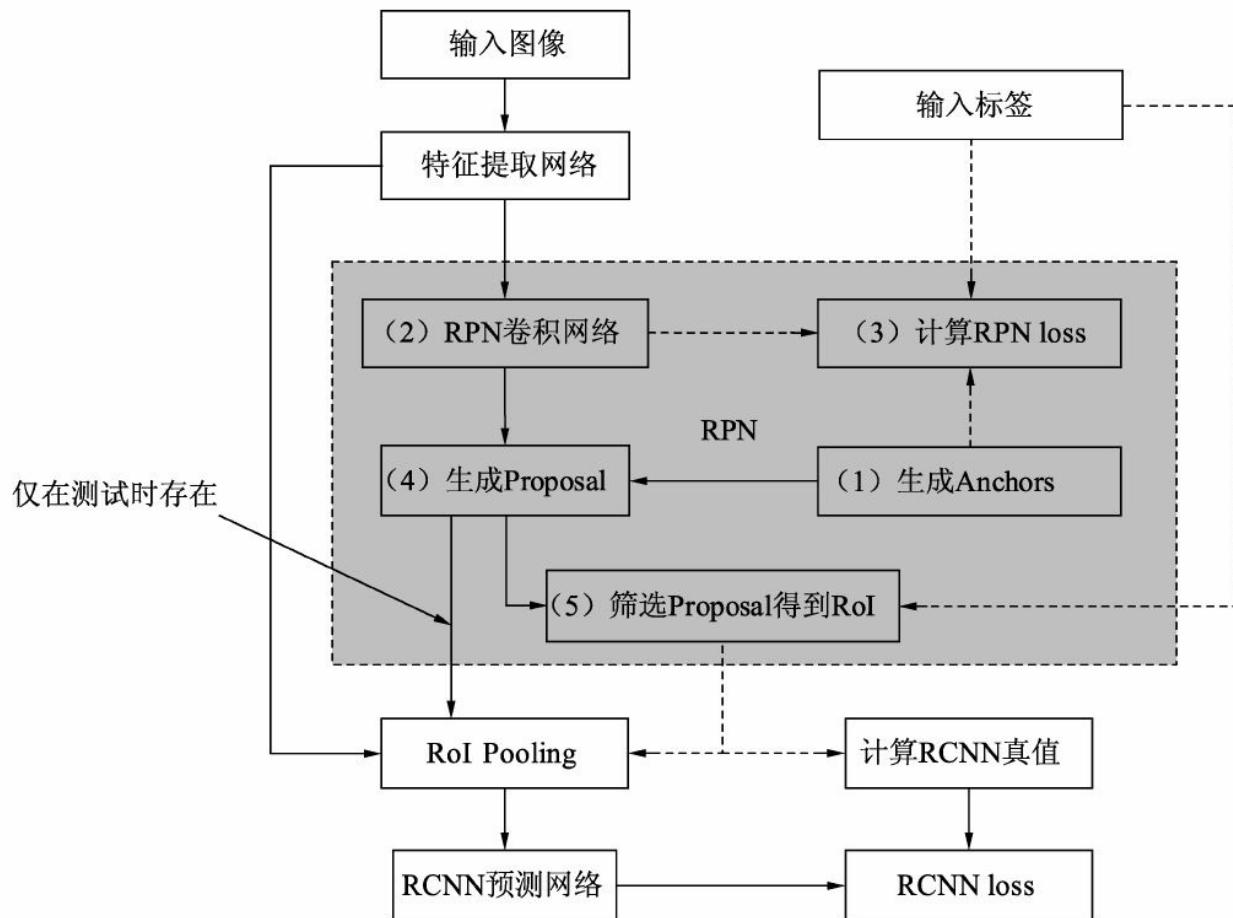


图4.3 Faster RCNN算法过程示意图

RoI Pooling模块：这部分承上启下，接受卷积网络提取的feature

map和RPN的RoI，输出送到RCNN网络中。由于RCNN模块使用了全连接网络，要求特征的维度固定，而每一个RoI对应的特征大小各不相同，无法送入到全连接网络，因此RoI Pooling将RoI的特征池化到固定的维度，方便送到全连接网络中。

RCNN模块：将RoI Pooling得到的特征送入全连接网络，预测每一个RoI的分类，并预测偏移量以精修边框位置，并计算损失，完成整个Faster RCNN过程。**主要包含3部分：**

·RCNN全连接网络：将得到的固定维度的RoI特征接到全连接网络中，输出为RCNN部分的预测得分与预测回归偏移量。

·计算RCNN的真值：对于筛选出的RoI，需要确定是正样本还是负样本，同时计算与对应真实物体的偏移量。在实际实现时，为实现方便，这一步往往与RPN最后筛选RoI那一步放到一起。

·RCNN loss：通过RCNN的预测值与RoI部分的真值，计算分类与回归loss。

从整个过程可以看出，Faster RCNN是一个两阶的算法，即RPN与RCNN，这两步都需要计算损失，只不过前者还要为后者提供较好的感兴趣区域。具体每一个模块的细节实现，将在后面章节中详细讲解。

## 4.4 详解RPN

RPN部分的输入、输出如下：

·输入：feature map、物体标签，即训练集中所有物体的类别与边框位置。

·输出：Proposal、分类Loss、回归Loss，其中，Proposal作为生成的区域，供后续模块分类与回归。两部分损失用作优化网络。

RPN模块的总体代码逻辑如下，源代码文件见  
lib/model/faster\_rcnn/faster\_rcnn.py。

---

```
def forward(self, im_data, im_info, gt_boxes, num_boxes):
    # 输入数据的第一维是batch数
    batch_size = im_data.size()
    im_info = im_info.data
    gt_boxes = gt_boxes.data
    num_boxes = num_boxes.data
    # 从VGG的Backbone中获取feature map
    base_feat = self.RCNN_base(im_data)
    # 将feature map送入RPN，得到Proposal与分类与回归loss
    rois, rpn_loss_cls, rpn_loss_bbox = self.RCNN_rpn(base_feat, im_info,
    gt_boxes, num_boxes)
    .....
```

---

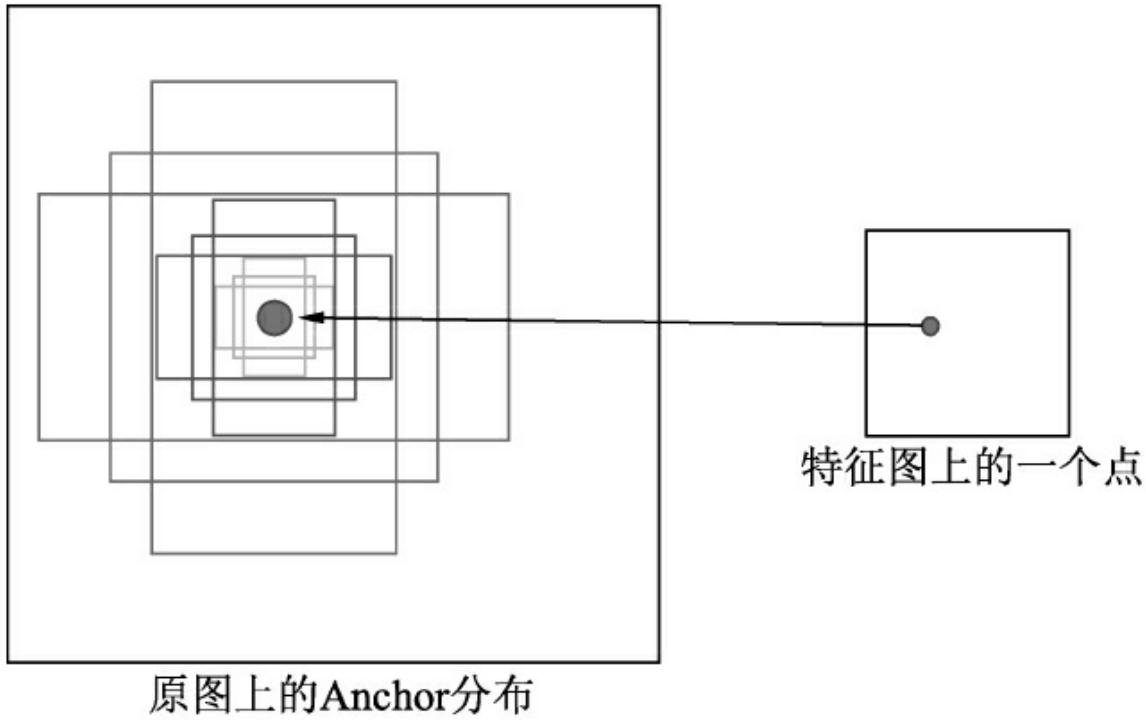
说明：本书中的源代码文件获取方式请参见前言中的说明。

#### 4.4.1 理解Anchor

理解Anchor是理解RPN乃至Faster RCNN的关键。Faster RCNN先提供一些先验的边框，然后再去筛选与修正，这样在Anchor的基础上做物体检测要比从无到有的直接拟合物体的边框容易一些。

Anchor的本质是在原图大小上的一系列的矩形框，但Faster RCNN将这一系列的矩形框和feature map进行了关联。具体做法是，首先对feature map进行 $3 \times 3$ 的卷积操作，得到的每一个点的维度是512维，这512维的数据对应着原始图片上的很多不同的大小与宽高区域的特征，这些区域的中心点都相同。如果下采样率为默认的16，则每一个点的坐标乘以16即可得到对应的原图坐标。

为适应不同物体的大小与宽高，在作者的论文中，默认在每一个点上抽取了9种Anchors，具体Scale为{8,16,32}，Ratio为{0.5,1,2}，将这9种Anchors的大小反算到原图上，即得到不同的原始Proposal，如图4.4所示。由于feature map大小为 $37 \times 50$ ，因此一共有 $37 \times 50 \times 9 = 16650$ 个Anchors。而后通过分类网络与回归网络得到每一个Anchor的前景背景概率和偏移量，前景背景概率用来判断Anchor是前景的概率，回归网络则是将预测偏移量作用到Anchor上使得Anchor更接近于真实物体坐标。



原图上的Anchor分布

图4.4 Anchor原理示意图

在具体的代码实现时，lib/model/rpn下的anchor\_target\_layer.py与proposal\_layer.py在类的初始化中均生成了所需的Anchor，下面从代码角度简单讲解一下生成过程，源代码文件见lib/model/rpn/generate\_anchors.py。

---

```

def generate_anchors(base_size=16, ratios=[0.5, 1, 2], scales=2**np.arange(3, 6)):
    # 首先创建一个基本Anchor为[0, 0, 15, 15]
    base_anchor = np.array([1, 1, base_size, base_size]) - 1
    # 将基本Anchor进行宽高变化，生成三种宽高比的s:Anchor
    ratio_anchors = _ratio_enum(base_anchor, ratios)
    # 将上述Anchor再进行尺度变化，得到最终的9种Anchors
    anchors = np.vstack([_scale_enum(ratio_anchors[i, :], scales)
                        for i in xrange(ratio_anchors.shape[0])])
    # 返回对应于feature map大小的Anchors
    return anchors

```

---

#### 4.4.2 RPN的真值与预测量

理解RPN的预测量与真值分别是什么，也是理解RPN原理的关键。对于物体检测任务来讲，模型需要预测每一个物体的类别及其出现的位置，即类别、中心点坐标 $x$ 与 $y$ 、宽 $w$ 与高 $h$ 这5个量。由于有了Anchor这个先验框，RPN可以预测Anchor的类别作为预测边框的类别，并且可以预测真实的边框相对于Anchor的偏移量，而不是直接预测边框的中心点坐标 $x$ 与 $y$ 、宽高 $w$ 与 $h$ 。

举个例子，如图4.5所示，输入图像中有3个Anchors与两个标签，从位置来看，Anchor A、C分别和标签M、N有一定的重叠，而Anchor B位置更像是背景。

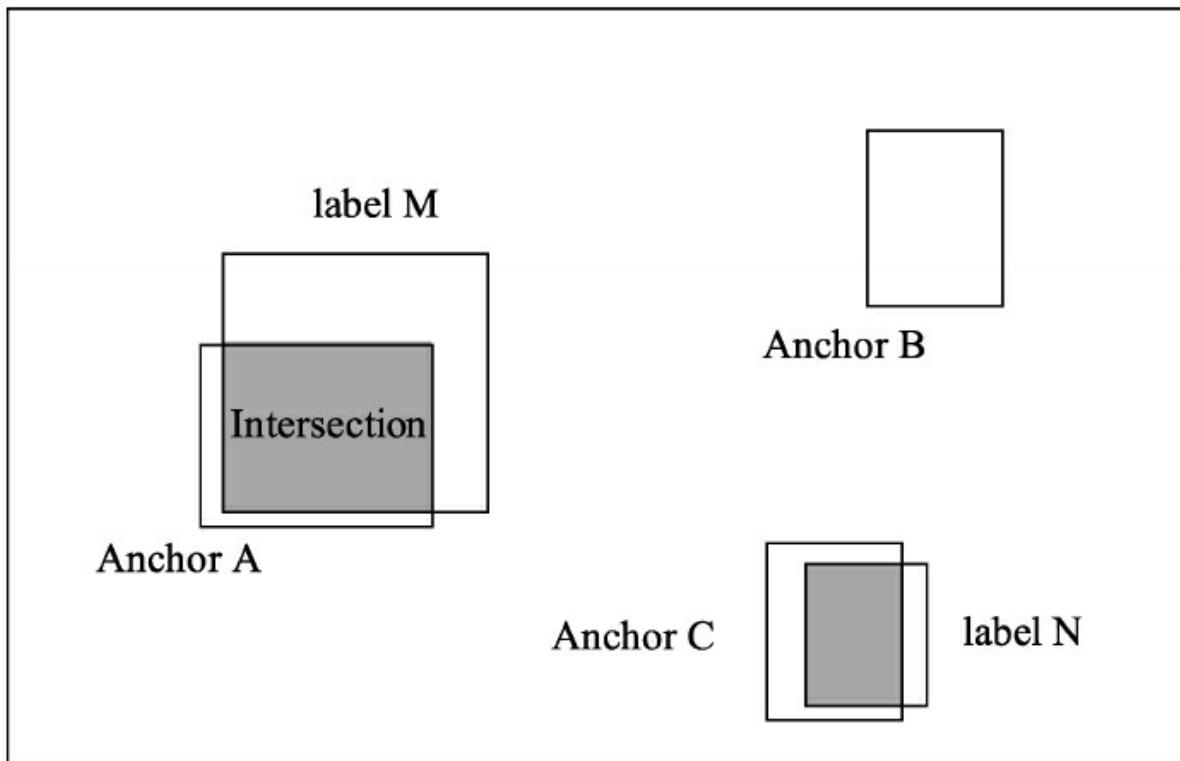


图4.5 图像中Anchor与标签的关系

首先介绍模型的真值。对于类别的真值，由于RPN只负责区域生成，保证recall，而没必要细分每一个区域属于哪一个类别，因此只需要前景与背景两个类别，前景即有物体，背景则没有物体。

RPN通过计算Anchor与标签的IoU来判断一个Anchor是属于前景还是背景。IoU的含义是两个框的公共部分占所有部分的比例，即重合比例。在图4.5中，Anchor A与标签M的IoU计算公式如式（4-1）所示。

$$IoU(A, M) = \frac{A \cap M}{A \cup M} \quad (4-1)$$

当IoU大于一定值时，该Anchor的真值为前景，低于一定值时，该Anchor的真值为背景。

然后是偏移量的真值。仍以图4.5中的Anchor A与标签M为例，假设Anchor A的中心坐标为 $x_a$ 与 $y_a$ ，宽高分别为 $w_a$ 与 $h_a$ ，标签M的中心坐标为x与y，宽高分别为w与h，则对应的偏移真值计算公式如式（4-2）所示。

$$\begin{cases} t_x = (x - x_a) / w_a \\ t_y = (y - y_a) / h_a \\ t_w = \log\left(\frac{w}{w_a}\right) \\ t_h = \log\left(\frac{h}{h_a}\right) \end{cases} \quad (4-2)$$

从式（4-2）中可以看到，位置偏移 $t_x$ 与 $t_y$ 利用宽与高进行了归一化，而宽高偏移 $t_w$ 与 $t_h$ 进行了对数处理，这样的好处是进一步限制了偏

移量的范围，便于预测。

有了上述的真值，为了求取损失，RPN通过卷积网络分别得到了类别与偏移量的预测值。具体来讲，RPN需要预测每一个Anchor属于前景与背景的概率，同时也需要预测真实物体相对于Anchor的偏移量，记为 $t_x^*$ 、 $t_y^*$ 、 $t_w^*$ 和 $t_h^*$ 。具体的网络下一节细讲。

另外，在得到预测偏移量后，可以使用式（4-3）的公式将预测偏移量作用到对应的Anchor上，得到预测框的实际位置 $x^*$ 、 $y^*$ 、 $w^*$ 和 $h^*$ 。

$$\left\{ \begin{array}{l} t_x^* = (x^* - x_a) / w_a \\ t_y^* = (y^* - y_a) / h_a \\ t_w^* = \log\left(\frac{w^*}{w_a}\right) \\ t_h^* = \log\left(\frac{h^*}{h_a}\right) \end{array} \right. \quad (4-3)$$

如果没有Anchor，做物体检测需要直接预测每个框的坐标，由于框的坐标变化幅度大，使网络很难收敛与准确预测，而Anchor相当于提供了一个先验的阶梯，使得模型去预测Anchor的偏移量，即可更好地接近真实物体。

实际上，Anchor是我们想要预测属性的先验参考值，并不局限于矩形框。如果需要，我们也可以增加其他类型的先验，如多边形框、角度和速度等。

### 4.4.3 RPN卷积网络

为了实现上述的预测，RPN搭建了如图4.6所示的网络结构。具体实现时，在feature map上首先用 $3\times 3$ 的卷积进行更深的特征提取，然后利用 $1\times 1$ 的卷积分别实现分类网络和回归网络。

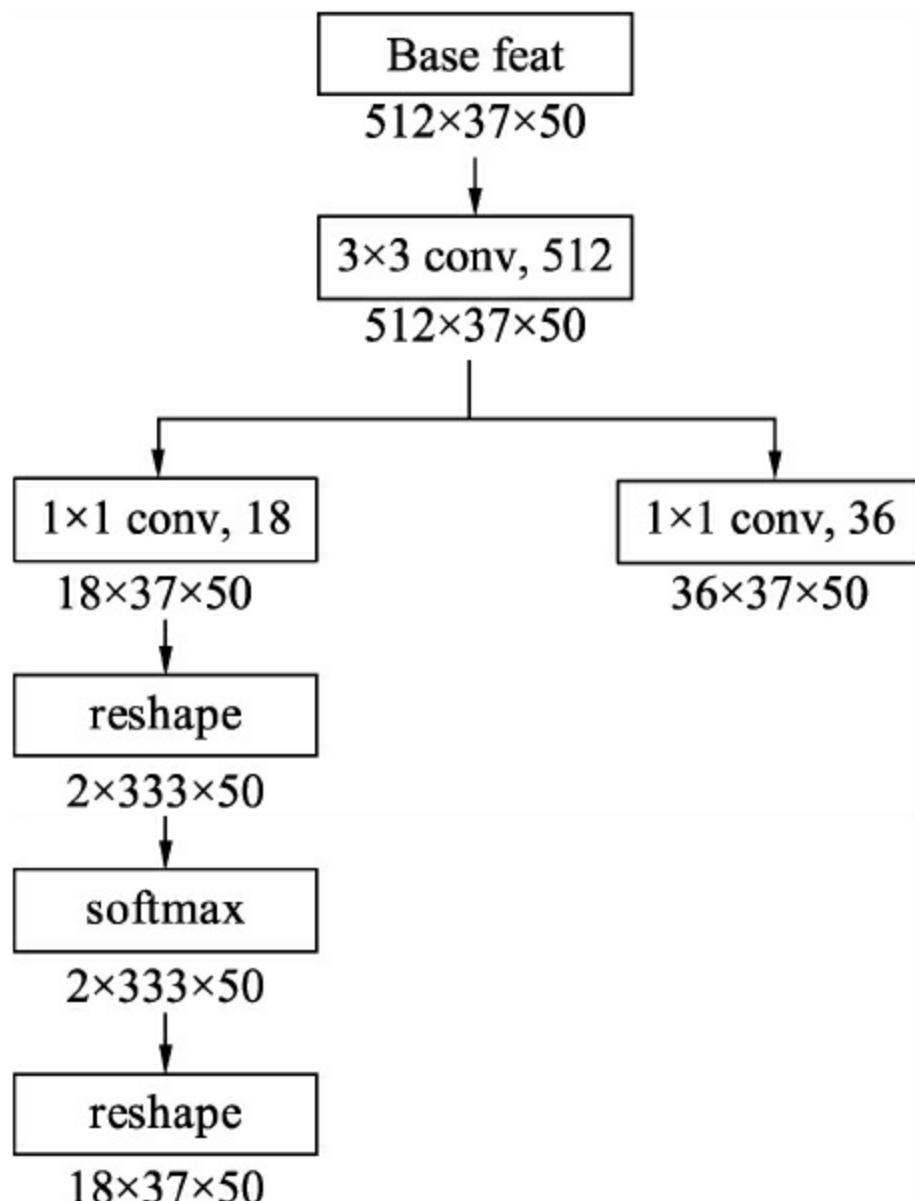


图4.6 RPN网络计算流程

在物体检测中，通常我们将有物体的位置称为前景，没有物体的位置称为背景。在分类网络分支中，首先使用 $1 \times 1$ 卷积输出 $18 \times 37 \times 50$ 的特征，由于每个点默认有9个Anchors，并且每个Anchor只预测其属于前景还是背景，因此通道数为18。随后利用torch.view()函数将特征映射到 ~~$2 \times 33 \times 75$~~ ，这样第一维仅仅是一个Anchor的前景背景得分，并送到Softmax函数中进行概率计算，得到的特征再变换到 $18 \times 37 \times 50$ 的维度，最终输出的是每个Anchor属于前景与背景的概率。

在回归分支中，利用 $1 \times 1$ 卷积输出 $36 \times 37 \times 50$ 的特征，第一维的36包含9个Anchors的预测，每一个Anchor有4个数据，分别代表了每一个Anchor的中心点横纵坐标及宽高这4个量相对于真值的偏移量。RPN的网络部分代码如下，源代码文件见lib/model/rpn/rpn.py。

---

```
def forward(self, base_feat, im_info, gt_boxes, num_boxes):
    # 输入数据的第一维是batch值
    batch_size = base_feat.size(0)
    # 首先利用 $3 \times 3$ 卷积进一步融合特征
    rpn_conv1 = F.relu(self.RPN_Conv(base_feat), inplace=True)
    # 利用 $1 \times 1$ 卷积得到分类网络，每个点代表Anchor的前景背景得分
    rpn_cls_score = self.RPN_cls_score(rpn_conv1)
    # 利用reshape与softmax得到Anchor的前景背景概率
    rpn_cls_score_reshape = self.reshape(rpn_cls_score, 2)
    rpn_cls_prob_reshape = F.softmax(rpn_cls_score_reshape, 1)
    rpn_cls_prob = self.reshape(rpn_cls_prob_reshape, 18)
    # 利用 $1 \times 1$ 卷积得到回归网络，每一个点代表Anchor的偏移
    rpn_bbox_pred = self.RPN_bbox_pred(rpn_conv1)
```

---

#### 4.4.4 RPN真值的求取

上一节的RPN分类与回归网络得到的是模型的预测值，而为了计算预测的损失，还需要得到分类与偏移预测的真值，具体指的是每一个Anchor是否对应着真实物体，以及每一个Anchor对应物体的真实偏移值。求真值的具体实现过程如图4.7所示，主要包含4步，下面具体介绍。

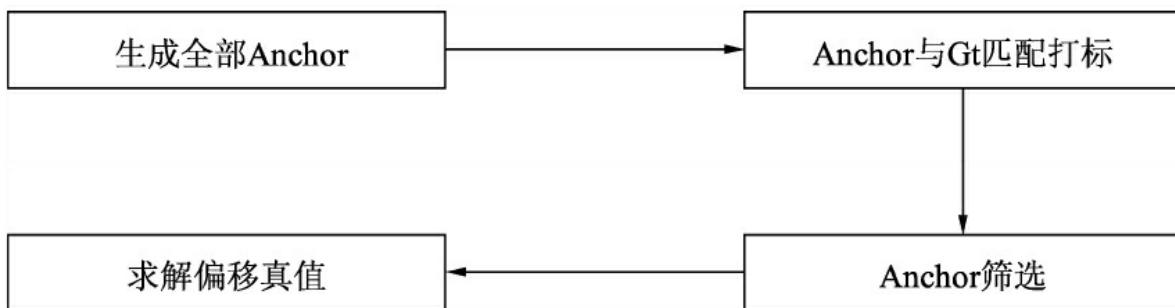


图4.7 RPN真值求取过程

##### 1. Anchor生成

这部分与前面Anchor的生成过程一样，可以得到 $37 \times 50 \times 9 = 16650$ 个Anchors。由于按照这种方式生成的Anchor会有一些边界在图像边框外，因此还需要把这部分超过图像边框的Anchors过滤掉，具体生成过程如下，源代码文件见lib/model/rpn/anchor\_target\_layer.py。

---

```
def forward(self, input):
    .....
    # 利用NumPy首先得到原图上的中心点坐标，并利用contiguous保证内存连续
    shifts = torch.from_numpy(np.vstack((shift_x.ravel(), shift_y.ravel(),
                                         shift_x.ravel(), shift_y.ravel()))).transpose()
    shifts = shifts.contiguous().type_as(rpn_cls_score).float()
    .....
    # 调用基础Anchor生成所有Anchors
    self._anchors = self._anchors.type_as(gt_boxes)
```

```
all_anchors = self._anchors.view(1, A, 4) + shifts.view(K, 1, 4)
.....
# 保留边框内的Anchors
inds_inside = torch.nonzero(keep).view(-1)
anchors = all_anchors[inds_inside, :]
```

---

## 2. Anchor与标签的匹配

为了计算Anchor的损失，在生成Anchor之后，我们还需要得到每个Anchor的类别，由于RPN的作用是建议框生成，而非详细的分类，因此只需要区分正样本与负样本，即每个Anchor是属于正样本还是负样本。

前面已经介绍了通过计算Anchor与标签的IoU来判断是正样本还是负样本。在具体实现时，需要计算每一个Anchor与每一个标签的IoU，因此会得到一个IoU矩阵，具体的判断标准如下：

- 对于任何一个Anchor，与所有标签的最大IoU小于0.3，则视为负样本。
- 对于任何一个标签，与其有最大IoU的Anchor视为正样本。
- 对于任何一个Anchor，与所有标签的最大IoU大于0.7，则视为正样本。

匹配与筛选的代码示例如下，源代码文件见  
lib/model/rpn/anchor\_target\_layer.py。

---

```
def forward(self, input):
    # 生成标签向量，对应每一个Anchor的状态，1为正，0为负，初始化为-1
    labels = gt_boxes.new(batch_size, inds_inside.size(0)).fill_(-1)
    # 生成IoU矩阵，每一行代表一个Anchor，每一列代表一个标签
    overlaps = bbox_overlaps_batch(anchors, gt_boxes)
    # 对每一行求最大值，返回的第一个为最大值，第二个为最大值的位置
    max_overlaps, argmax_overlaps = torch.max(overlaps, 2)
    # 对每一列取最大值，返回的是每一个标签对应的IoU最大值
    gt_max_overlaps, _ = torch.max(overlaps, 1)
```

```
# 如果一个Anchor最大的IoU小于0.3, 视为负样本
labels[max_overlaps < 0.3] = 0
# 与所有Anchors的最大IoU为0的标签要过滤掉
gt_max_overlaps[gt_max_overlaps==0] = 1e-5
# 将与标签有最大IoU的Anchor赋予正样本
keep = torch.sum(overlaps.eq(gt_max_overlaps.view(batch_size,
1, -1).expand_as(overlaps)), 2)
if torch.sum(keep)>0:
    labels[keep>0] = 1
# 如果一个Anchor最大的IoU大于0.7, 视为正样本
labels[max_overlaps >= 0.7] = 1
```

---

需要注意的是，上述三者的顺序不能随意变动，要保证一个Anchor既符合正样本，也符合负样本时，赋予正样本。并且为了保证这一阶段的召回率，允许多个Anchors对应一个标签，而不允许一个标签对应多个Anchors。

### 3. Anchor的筛选

由于Anchor的总数量接近于2万，并且大部分Anchor的标签都是背景，如果都计算损失的话则正、负样本失去了均衡，不利于网络的收敛。在此，RPN默认选择256个Anchors进行损失的计算，其中最多不超过128个的正样本。如果数量超过了限定值，则进行随机选取。当然，这里的256与128都可以根据实际情况进行调整，而不是固定死的。

Anchor筛选的代码示例如下，源代码文件见  
lib/model/rpn/anchor\_target\_layer.py。

---

```
def forward(self, input):
    .....
    for i in range(batch_size):
        # 如果正样本数量太多，则进行下采样随机选取
        if sum_fg[i] > 128:
            fg_inds = torch.nonzero(labels[i] == 1).view(-1)
            rand_num = torch.from_numpy(np.random.permutation(
                (fg_inds.size(0))).type_as(gt_boxes).long())
            disable_inds = fg_inds[rand_num[:fg_inds.size(0)-num_fg]]
```

```
    labels[i][disable_inds] = -1
    # 负样本同上
    ....
```

---

#### 4. 求解回归偏移真值

上一步将每个Anchor赋予正样本或者负样本代表了预测类别的真值，而回归部分的偏移量真值还需要利用Anchor与对应的标签求解得到，具体公式见式4-2。

得到偏移量的真值后，将其保存在bbox\_targets中。与此同时，还需要解两个权值矩阵bbox\_inside\_weights和bbox\_outside\_weights，前者是用来设置正样本回归的权重，正样本设置为1，负样本设置为0，因为负样本对应的是背景，不需要进行回归；后者的作用则是平衡RPN分类损失与回归损失的权重，在此设置为1/256。

求解回归的偏移真值示例如下，源代码文件见  
lib/model/rpn/anchor\_target\_layer.py。

---

```
def forward(self, input):
    .....
    # 选择每一个Anchor对应最大IoU的标签进行偏移计算
    bbox_targets = _compute_targets_batch(anchors,
                                          gt_boxes.view(-1, 5)[argmax_overlaps.view(-1), :].view(batch_
size, -1, 5))
    # 设置两个权重向量
    bbox_inside_weights[labels==1] = 1
    num_examples = torch.sum(labels[i] >=0)
    bbox_outside_weights[labels == 1] = 1.0 / examples.item()
    bbox_outside_weights[labels == 0] = 1.0 / examples.item()
```

---

真值的求取部分最后的输出包含了分类的标签label、回归偏移的真值bbox\_targets，以及两个权重向量bbox\_inside\_weights与bbox\_outside\_weights。

#### 4.4.5 损失函数设计

有了网络预测值与真值，接下来就可以计算损失了。RPN的损失函数包含分类与回归两部分，具体公式如式（4-4）所示。

$$L(\{P_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i P_i^* L_{reg}(t_i, t_i^*) \quad (4-4)$$

$\sum_i L_{cls}(p_i, p_i^*)$ 代表了256个筛选出的Anchors的分类损失， $p_i$ 为每一个Anchor的类别真值， $p_i^*$ 为每一个Anchor的预测类别。由于RPN的作用是选择出Proposal，并不要求细分出是哪一类前景，因此在这一阶段是二分类，使用的是交叉熵损失。值得注意的是，在F.cross\_entropy()函数中集成了Softmax的操作，因此应该传入得分，而非经过Softmax之后的预测值。

$\sum_i P_i^* L_{reg}(t_i, t_i^*)$ 代表了回归损失，其中bbox inside weights实际上起到了

$p_i^*$ 进行筛选的作用，bbox outside weights起到了 $\lambda \frac{1}{N_{reg}}$ 来平衡两部分损失的作用。回归损失使用了smooth<sub>L1</sub>函数，具体公式如式（4-5）与式（4-6）所示。

$$L_{reg}(t_i, t_i^*) = \sum_{i \in x, y, w, h} smooth_{L1}(t_i - t_i^*) \quad (4-5)$$

$$smooth_{L1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \quad (4-6)$$

从式（4-6）中可以看到，smooth<sub>L1</sub>函数结合了1阶与2阶损失函数，原因在于，当预测偏移量与真值差距较大时，使用2阶函数时导数太

大，模型容易发散而不容易收敛，因此在大于1时采用了导数较小的1阶损失函数。

损失函数的代码接口如下，源代码文件见ib/model/rpn/rpn.py。

```
# 先对scores进行筛选得到256个样本的得分，随后进行交叉熵求解
self.rpn_loss_cls = F.cross_entropy(rpn_cls_score, rpn_label)
# 利用smoothL1损失函数进行loss计算
self.rpn_loss_box = _smooth_l1_loss(rpn_bbox_pred, rpn_bbox_targets,
                                     rpn_bbox_inside_weights, rpn_bbox_outside_weights, sigma=3,
                                     dim=[1, 2, 3])
```

#### 4.4.6 NMS与生成Proposal

完成了损失的计算，RPN的另一个功能就是区域生成，即生成较好的Proposal，以供下一个阶段进行细分类与回归。

NMS生成Proposal的主要过程如图4.8所示，首先生成大小固定的全部Anchors，然后将网络中得到的回归偏移作用到Anchor上使Anchor更加贴近于真值，并修剪超出图像尺寸的Proposal，得到最初的建议区域。在这之后，按照分类网络输出的得分对Anchor排序，保留前12000个得分高的Anchors。由于一个物体可能会有多个Anchors重叠对应，因此再应用非极大值抑制（NMS）将重叠的框去掉，最后在剩余的Proposal中再次根据RPN的预测得分选择前2000个，作为最终的Proposal，输出到下一个阶段。

NMS与Proposal的筛选过程如下，源代码文件见  
lib/model/rpn/proposal\_layer.py。

---

```
def forward(self, input):
    # 生成Anchor后，首先利用回归网络对Anchor进行偏移修整
    proposals = bbox_transform_inv(anchors, bbox_deltas, batch_size)
    # 将超出图像范围的边框修整到图像边界
    proposals = clip_boxes(proposals, im_info, batch_size)
    # 利用分类网络的得分对proposal进行排序
    _, order = torch.sort(scores_keep, 1, True)
    # 选取前12000个
    order_single = order_single[:12000]
    # 进行NMS，在此利用GPU进行计算，提高效率
    keep_idx_i = nms(torch.cat((proposals_single, scores_single), 1), 0.7,
                     force_cpu=False)
    # 最终选择前2000个作为最终的Proposal输出
    keep_idx_i = keep_idx_i[:2000]
```

---

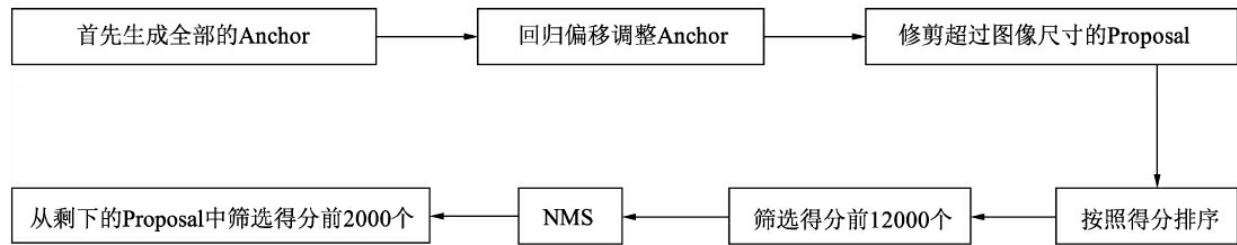


图4.8 RPN生成Proposal的过程

#### 4.4.7 筛选Proposal得到RoI

在训练时，上一步生成的Proposal数量为2000个，其中仍然有很多背景框，真正包含物体的仍占少数，因此完全可以针对Proposal进行再一步筛选，过程与RPN中筛选Anchor的过程类似，利用标签与Proposal构建IoU矩阵，通过与标签的重合程度选出256个正负样本。这一步有3个作用：

·筛选出了更贴近真实物体的RoI，使送入到后续网络的物体正、负样本更均衡，避免了负样本过多，正样本过少的情况。

·减少了送入后续全连接网络的数量，有效减少了计算量。

·筛选Proposal得到RoI的过程中，由于使用了标签来筛选，因此也为每一个RoI赋予了正、负样本的标签，同时可以在此求得RoI变换到对应标签的偏移量，这样就求得了RCNN部分的真值。

具体实现时，首先计算Proposal与所有的物体标签的IoU矩阵，然后根据IoU矩阵的值来筛选出符合条件的正负样本。筛选标准如下：

·对于任何一个Proposal，其与所有标签的最大IoU如果大于等于0.5，则视为正样本。

·对于任何一个Proposal，其与所有标签的最大IoU如果大于等于0且小于0.5，则视为负样本。

经过上述标准的筛选，选出的正、负样本数量不一，在此设定正、负样本的总数为256个，其中正样本的数量为p个。为了控制正、负样本的比例基本满足1:3，在此正样本数量p不超过64，如果超过了64则从正样本中随机选取64个。剩余的数量256-p为负样本的数量，如果超过了

256-p则从负样本中随机选取256-p个。

经过上述操作后，选出了最终的256个RoI，并且每一个RoI都赋予了正样本或者负样本的标签。在此也可以进一步求得每一个RoI的真值，即属于哪一个类别及对应真值物体的偏移量。

筛选Proposal过程的代码示例如下，源代码文件见  
lib/model/rpn/proposal\_target\_layer\_cascade.py。

---

```
def _sample_rois_pytorch(self, all_rois, gt_boxes,
                        fg_rois_per_image, rois_per_image, num_classes):
    # 利用Proposal与标签生成IoU矩阵
    overlaps = bbox_overlaps_batch(all_rois, gt_boxes)
    # 选择满足条件的正负样本
    fg_inds = torch.nonzero(max_overlaps[i] >= 0.5).view(-1)
    bg_inds = torch.nonzero((max_overlaps[i] < 0.5 & max_overlaps[i] >= 0)).
        view(-1)
    # 如果正样本超过64个，负样本超过(256-正样本)的数量，则进行下采样随机选取
    rand_num = torch.from_numpy(np.random.permutation(
        (fg_num_rois)).type_as(gt_boxes).long())
    fg_inds = fg_inds[rand_num[:fg_rois_per_this_image]]
    # 计算每一个Proposal相对于其标签的偏移量，并记录权重
    bbox_target_data = self._compute_targets_pytorch(rois_batch[:, :, 1:5],
                                                    gt_rois_batch[:, :, :4])
    bbox_targets, bbox_inside_weights =
        self._get_bbox_regression_labels_pytorch(bbox_target_data, labels_
batch, num_classes)
```

---

最终返回分类的真值label，回归的偏移真值bbox\_targets，以及每一个Proposal对应的权重bbox\_inside\_weights与bbox\_outside\_weights。

## 4.5 RoI Pooling层

上述步骤得到了256个RoI，以及每一个RoI对应的类别与偏移量真值，为了计算损失，还需要计算每一个RoI的预测量。

前面的VGGNet网络已经提供了整张图像的feature map，因此自然联想到可以利用此feature map，将每一个RoI区域对应的特征提取出来，然后接入一个全连接网络，分别预测其RoI的分类与偏移量。

然而，由于RoI是由各种大小宽高不同的Anchors经过偏移修正、筛选等过程生成的，因此其大小不一且带有浮点数，然后续相连的全连接网络要求输入特征大小维度固定，这就需要有一个模块，能够把各种维度不同的RoI变换到维度相同的特征，以满足后续全连接网络的要求，于是RoI Pooling就产生了。

对RoI进行池化的思想在SPPNet中就已经出现了，只不过在Fast RCNN中提出的RoI Pooling算法利用最近邻差值算法将池化过程进行了简化，而在随后的Mask RCNN中进一步提出了RoI Align的算法，利用双线性插值，进一步提升了算法精度。

在此我们举一个例子来讲解这几种算法的思想，假设当前RoI大小为 $332 \times 332$ ，使用VGGNet的全连接层，其所需的特征向量维度为 $512 \times 7 \times 7$ ，由于目前的特征图通道数为512，Pooling的过程就是如何获得 $7 \times 7$ 大小区域的特征。

### 1. RoI Pooling简介

RoI Pooling的实现过程如图4.9所示，假设当前的RoI为图4.9中左侧图像的边框，大小为 $332 \times 332$ ，为了得到这个RoI的特征图，首先需要将

该区域映射到全图的特征图上，由于下采样率为16，因此该区域在特征图上的坐标直接除以16并取整，而对应的大小为 $332/16=20.75$ 。在此，RoI Pooling的做法是直接将浮点数量化为整数，取整为 $20\times 20$ ，也就得到了该RoI的特征，即图4.9中第3步的边框。

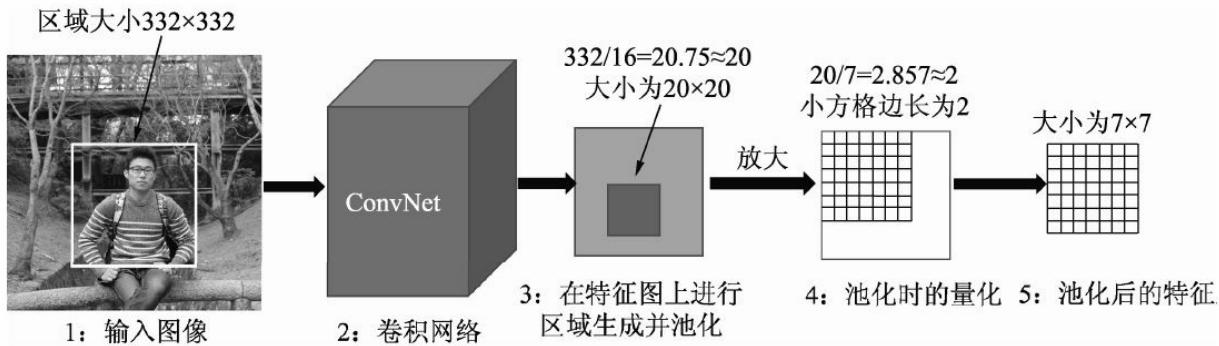


图4.9 RoI Pooling的实现过程示例

下一步还要将该 $20\times 20$ 区域处理为 $7\times 7$ 的特征，然而 $20/7\approx 2.857$ ，再次出现浮点数，RoI Pooling的做法是再次量化取整，将2.857取整为2，然后以2为步长从左上角开始选取7×7的区域，这样每个小方格在特征图上都对应 $2\times 2$ 的大小，如图4.9中第4步所示。

最后，取每个小方格内的最大特征值，作为这个小方格的输出，最终实现了 $7\times 7$ 的输出，也完成了池化的过程，如图4.9中第5步所示。

从实现过程中可以看到，RoI本来对应于 $20.75\times 20.75$ 的特征图区域，最后只取了 $14\times 14$ 的区域，因此RoI Pooling算法虽然简单，但量化取整带来的偏差势必会影响网络，尤其是回归物体位置的准确率。

## 2. RoI Align简介

RoI Align的思想是使用双线性插值获得坐标为浮点数的点的值，主要过程如图4.10所示，依然将RoI对应到特征图上，但坐标与大小都保留着浮点数，大小为 $20.75\times 20.75$ ，不做量化。

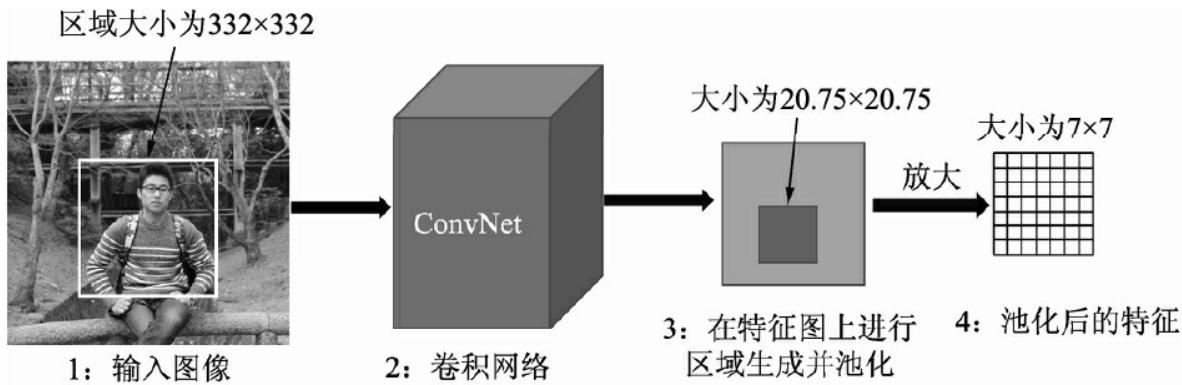


图4.10 ROI Align的实现过程示例

接下来，将特征图上的 $20.75 \times 20.75$ 大小均匀分成 $7 \times 7$ 方格的大小，中间的点依然保留浮点数。在此选择其中 $2 \times 2$ 方格为例，如图4.11所示，在每一个小方格内的特定位置选取4个采样点进行特征采样，如图4.11中每个小方格选择了4个小黑点，然后对这4个黑点的值选择最大值，作为这个方格最终的特征。这4个小黑点的位置与值该如何计算呢？

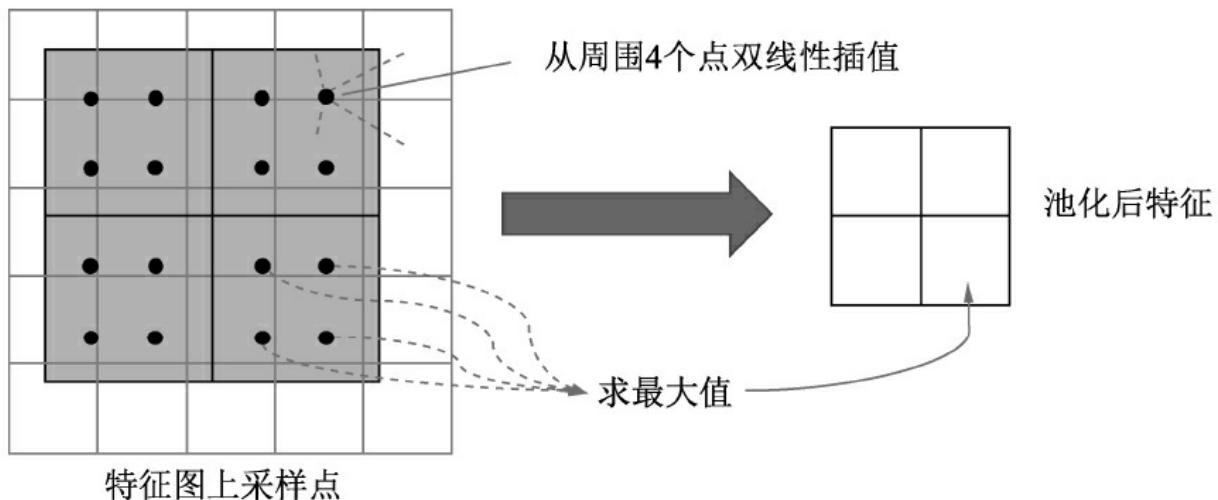


图4.11 ROI Align双线性插值与池化示例

对于黑点的位置，可以将小方格平均分成 $2 \times 2$ 的4份，然后这4份更小单元的中心点可以作为小黑点的位置。

至于如何计算这4个小黑点的值，RoI Align使用了双线性插值的方法。小黑点周围会有特征图上的4个特征点，利用这4个特征点双线性插值出该黑点的值。

由于Align算法最大可能地保留了原始区域的特征，因此Align算法对检测性能有显著的提升，尤其是对于受RoI Pooling影响大的情形，如本身特征区域较小的小物体，改善更为明显。

## 4.6 全连接RCNN模块

在经过RoI Pooling层之后，特征被池化到了固定的维度，因此接下来可以利用全连接网络进行分类与回归预测量的计算。在训练阶段，最后需要计算预测量与真值的损失并反传优化，而在前向测试阶段，可以直接将预测量加到RoI上，并输出预测量。

#### 4.6.1 RCNN全连接网络

RCNN全连接网络如图4.12所示，4.5节中256个RoI经过池化之后得到固定维度为 $512 \times 7 \times 7$ 的特征，在此首先将这三个维度延展为一维，因为全连接网络需要将一个RoI的特征全部连接起来。

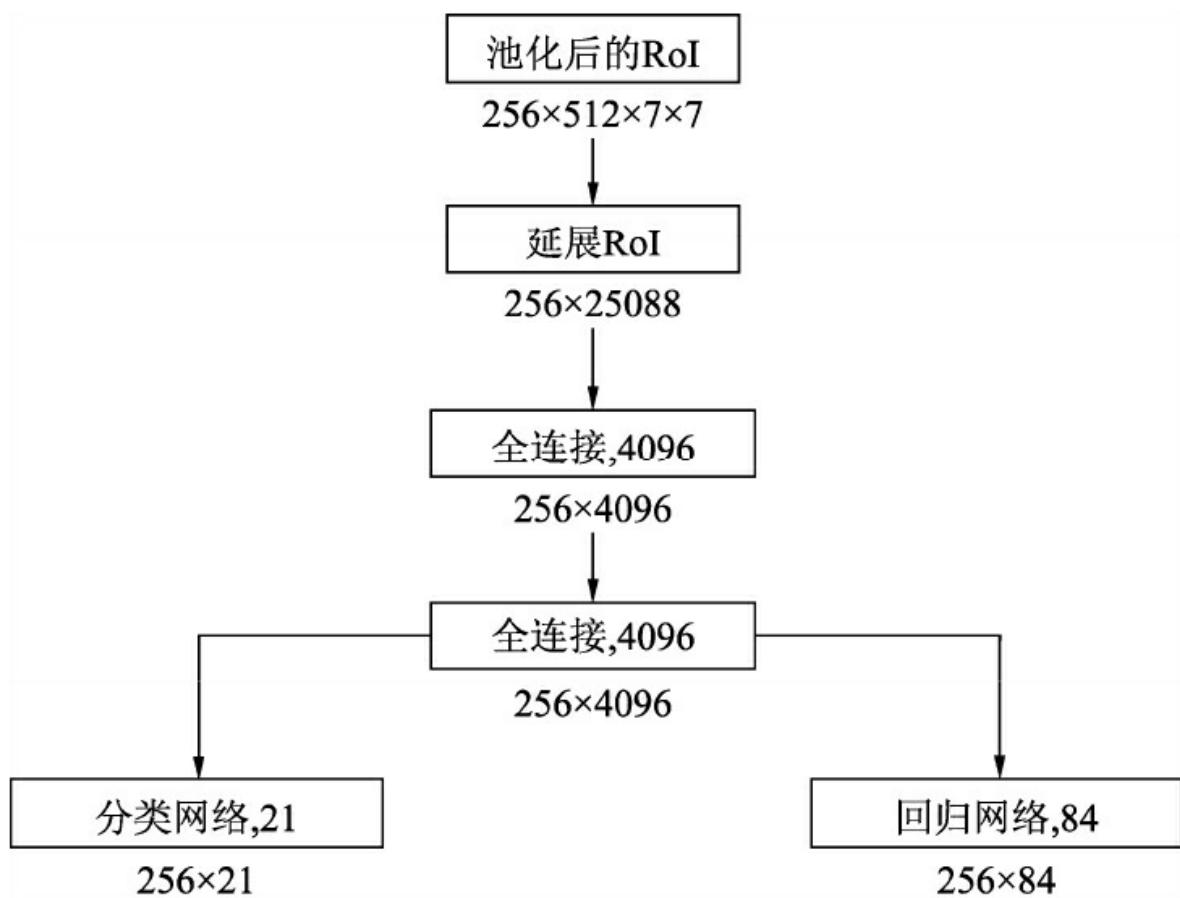


图4.12 RCNN部分网络计算流程

接下来利用VGGNet的两个全连接层，得到长度为4096的256个RoI特征。为了输出类别与回归的预测，将上述特征分别接入分类与回归的全连接网络。在此默认为21类物体，因此分类网络输出维度为21，回归网络则输出每一个类别下的4个位置偏移量，因此输出维度为84。

值得注意的是，虽然是256个RoI放到了一起计算，但相互之间是独立的，并没有使用到共享特征，因此造成了重复计算，这也是Faster RCNN的一个缺点。RCNN全连接部分的代码接口如下，源代码文件见lib/model/faster\_rcnn/faster\_rcnn.py。

---

```
def forward (self, im_data, im_info, gt_boxes, num_boxes):
    # 利用VGG的两层分类全连接网络进一步计算
    pooled_feat = self.head_to_tail(pooled_feat)
    # 分别计算分类与回归预测值
    cls_score = self.RCNN_cls_score(pooled_feat)
    bbox_pred = self.RCNN_bbox_pred(pooled_feat)
```

---

## 4.6.2 损失函数设计

RCNN部分的损失函数计算方法与RPN部分相同，不再重复。只不过在此为21个类别的分类，而RPN部分则是二分类，需要注意回归时至多有64个正样本参与回归计算，负样本不参与回归计算。

计算RCNN损失的代码接口如下，源代码文件见  
lib/model/faster\_rcnn/faster\_rcnn.py。

---

```
# 代码在lib/model/faster_rcnn/faster_rcnn.py文件中
# 对256个RoI进行分类损失求解，在此使用交叉熵损失
RCNN_loss_cls = F.cross_entropy(cls_score, rois_label)
# 利用smoothL1损失函数进行回归loss计算
RCNN_loss_box=_smooth_l1_loss(bbox_pred, rois_targets, rois_inside_
ws,rois_outside_ws)
```

---

## 4.7 Faster RCNN的改进算法

Faster RCNN从问世到现在，期间诞生了众多优秀的物体检测算法，但凭借其优越的性能，目前依然是物体检测领域主流的框架之一。尤其是在高精度、多尺度和小物体等物体检测领域的难点问题上，新型算法基本都是在Faster RCNN的基础上优化完善的。

本节将首先分析Faster RCNN的特点及可以优化的方向，然后从特征提取网络、RoI Pooling等多个角度，陆续讲解几个在Faster RCNN基础上优化改进的经典算法，如HyperNet、Mask RCNN、R-FCN及Cascade RCNN算法，如图4.13所示。

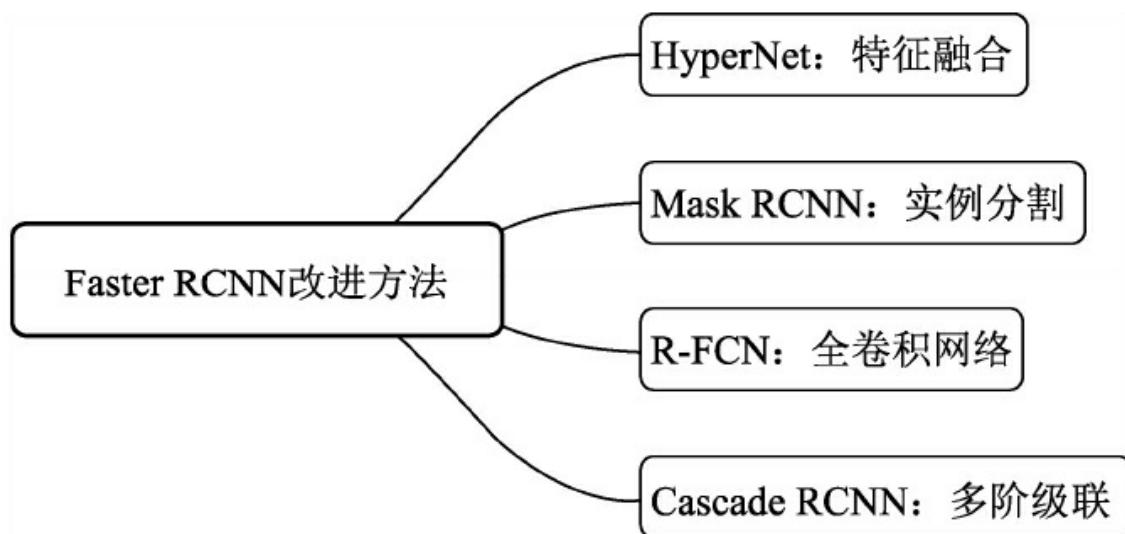


图4.13 Faster RCNN的多种改进算法

#### 4.7.1 审视Faster RCNN

Faster RCNN之所以生命力如此强大，应用如此广泛，离不开以下几个特点：

- 性能优越**：Faster RCNN通过两阶网络与RPN，实现了精度较高的物体检测性能。

- 两阶网络**：相比起其他一阶网络，两阶更为精准，尤其是针对高精度、多尺度以及小物体问题上，两阶网络优势更为明显。

- 通用性与鲁棒性**：Faster RCNN在多个数据集及物体任务上效果都很好，对于个人的数据集，往往Fine-tune后就能达到较好的效果。

- 可优化点很多**：Faster RCNN的整个算法框架中可以进行优化的点很多，提供了广阔的算法优化空间。

- 代码全面**：各大深度学习框架都有较好的Faster RCNN源码实现，使用方便。

当然，原始的Faster RCNN也存在一些缺点，而这些缺点也恰好成为了后续学者优化改进的方向，总体来看，可以从以下6个方面考虑：

- 卷积提取网络**：无论是VGGNet还是ResNet，其特征图仅仅是单层的，分辨率通常也较小，这些都不利于小物体及多尺度的物体检测，因此**多层融合的特征图、增大特征图的分辨率等都是可以优化的方向**。

- NMS**：在RPN产生Proposal时为了避免重叠的框，使用了NMS，并以分类得分为筛选标准。但**NMS本身的过滤对于遮挡物体不是特别友好**，本身属于两个物体的Proposal有可能因为NMS而过滤为1个，造成漏

检，因此改进优化NMS是可以带来检测性能提升的。

·RoI Pooling: Faster RCNN的原始RoI Pooling两次取整带来了精度的损失，因此后续Mask RCNN针对此Pooling进行了改进，提升了定位的精度。

·全连接: 原始Faster RCNN最后使用全连接网络，这部分全连接网络占据了网络的大部分参数，并且RoI Pooling后每一个RoI都要经过一遍全连接网络，没有共享计算，而如今全卷积网络是一个发展趋势，如何取代这部分全连接网络，实现更轻量的网络是需要研究的方向。

·正负样本: 在RPN及RCNN部分，都是通过超参数来限制正、负样本的数量，以保证正、负样本的均衡。而对于不同任务与数据，这种正、负样本均衡方法是否都是最有效的，也是一个研究的方向。

·两阶网络: Faster RCNN的RPN与RCNN两个阶段分工明确，带来了精度的提升，但速度相对较慢，实际实现上还没有达到实时。因此，网络阶数也是一个值得探讨的问题，如单阶是否可以使网络的速度更快，更多阶的网络是否可以进一步提升网络的精度等。

## 4.7.2 特征融合：HyperNet

卷积神经网络的特点是，深层的特征体现了强语义特征，有利于进行分类与识别，而浅层的特征分辨率高，有利于进行目标的定位。原始的Faster RCNN方法仅仅利用了单层的feature map（例如VGGNet的conv5-3），对于小尺度目标的检测较差，同时高IoU阈值时，边框定位的精度也不高。

在2016 CVPR上发表的HyperNet方法认为单独一个feature map层的特征不足以覆盖RoI的全部特性，因此提出了一个精心设计的网络结构，融合了浅、中、深3个层次的特征，取长补短，在处理好区域生成的同时，实现了较好的物体检测效果。

HyperNet提出的特征提取网络结构如图4.14所示，以VGGNet作为基础网络，分别从第1、3、5个卷积组后提取出特征，这3个特征分别对应着浅层、中层与深层的信息。然后，对浅层的特征进行最大值池化，对深层的特征进行反卷积，使得二者的分辨率都为原图大小的 $1/4$ ，与中层的分辨率相同，方便进行融合。得到3个特征图后，再接一个 $5 \times 5$ 的卷积以减少特征通道数，得到通道数为42的特征。

在三层的特征融合前，需要先经过一个LRN（Local Response Normalization）处理，LRN层借鉴了神经生物学中的侧抑制概念，即激活的神经元抑制周围的神经元，在此的作用是增加泛化能力，做平滑处理。

最后将特征沿着通道数维度拼接到一起。3个通道数为42的特征拼接一起后形成通道数为126的特征，作为最终输出。

HyperNet融合了多层特征的网络有如下3点好处：

·深层、中层、浅层的特征融合到一起，优势互补，利于提升检测精度。

·特征图分辨率为 $1/4$ ，特征细节更丰富，利于检测小物体。

·在区域生成与后续预测前计算好了特征，没有任何的冗余计算。

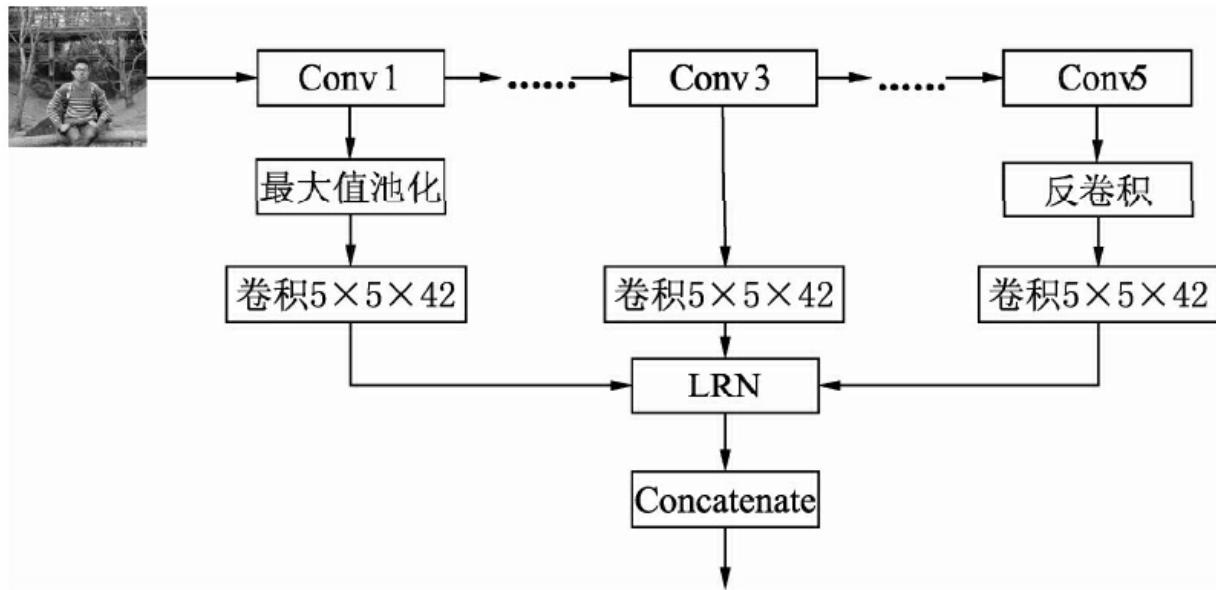


图4.14 HyperNet卷积提取网络结构

HyperNet实现了一个轻量化网络来实现候选区域生成。具体方法是，首先在特征图上生成3万个不同大小与宽高的候选框，经过RoI Pooling获得候选框的特征，再接卷积及相应的分类回归网络，进而可以得到预测值，结合标签就可以筛选出合适的Proposal。可以看出，这里的实现方法与Faster RCNN的RPN方法很相似，只不过先进行了RoI Pooling，再选择候选区域。

HyperNet后续的网络与Faster RCNN也基本相同，接入全连接网络完成最后的分类与回归。不同的地方是，HyperNet先使用了一个卷积降低通道数，并且Dropout的比例从0.5调整到了0.25。

由于提前使用了RoI Pooling，导致众多候选框特征都要经过一遍此Pooling层，计算量较大，为了加速，可以在Pooling前使用一个 $3 \times 3$ 卷积降低通道数为4，这种方法在大幅度降低计算量的前提下，基本没有精度的损失。

总体来看，HyperNet最大的特点还是提出了多层融合的特征，因此，其检测小物体的能力更加出色，并且由于特征图分辨率较大，物体的定位也更精准。此外，由于其出色的特征提取，HyperNet的Proposal的质量很高，前100个Proposal就可以实现97%的召回率。

值得注意，HyperNet使用到了反卷积来实现上采样，以扩大尺寸。通常来讲，上采样可以有3种实现方法：双线性插值、反池化（Unpooling）与反卷积。反卷积也叫转置卷积，但并非正常卷积的完全可逆过程。具体实现过程是，先按照一定的比例在特征图上补充0，然后旋转卷积核，再进行正向的卷积。反卷积方法经常被用在图像分割中，以扩大特征图尺寸。

### 4.7.3 实例分割：Mask RCNN

基于Faster RCNN，何凯明进一步提出了新的实例分割网络Mask RCNN，该方法在高效地完成物体检测的同时也实现了高质量的实例分割，获得了ICCV 2017的最佳论文。

Mask RCNN的网络结构如图4.15所示，可以看到其结构与Faster RCNN非常类似，但有3点主要区别：

- 在基础网络中采用了较为优秀的ResNet-FPN结构，多层次特征图有利于多尺度物体及小物体的检测。
- 提出了RoI Align方法来替代RoI Pooling，原因是RoI Pooling的取整做法损失了一些精度，而这对于分割任务来说较为致命。
- 得到感兴趣区域的特征后，在原来分类与回归的基础上，增加了一个Mask分支来预测每一个像素的类别。

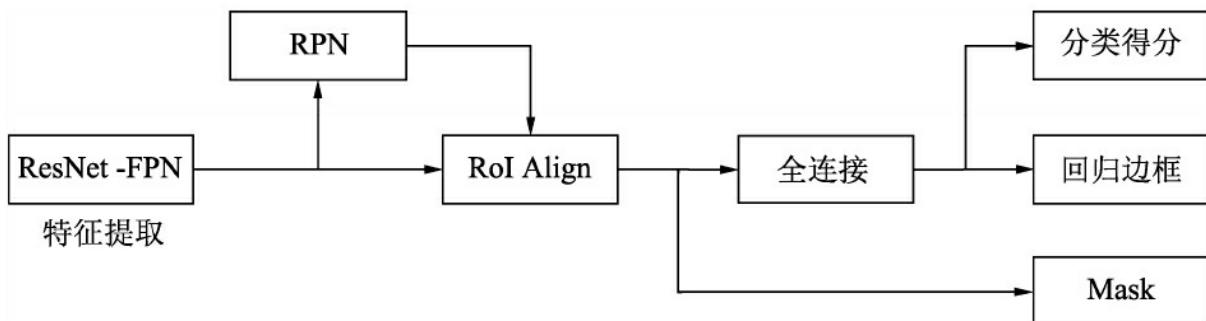


图4.15 Mask RCNN网络结构图

下面将详细介绍这3部分。

#### 1. 特征提取网络

当前多尺度的物体检测越来越重要，对于基础的特征提取网络来说，不同层的特征图恰好拥有着不同的感受野与尺度，因此可以利用多层次特征图来做多尺度的检测，例如上一节的HyperNet方法。

基于ResNet的FPN基础网络也是一个多层次特征结合的结构，包含了自下而上、自上而下及横向连接3个部分，这种结构可以将浅层、中层、深层的特征融合起来，使得特征同时具备强语义性与强空间性，具体结构参见第3章，在此不再重复。

原始的FPN会输出P2、P3、P4与P5 4个阶段的特征图，但在Mask RCNN中又增加了一个P6。将P5进行最大值池化即可得到P6，目的是获得更大感受野的特征，该阶段仅仅用在RPN网络中。

原始的Faster RCNN使用单层的特征图时，筛选出的RoI可以直接作用到特征图上进行RoI Pooling，但如果使用FPN作为基础网络时，由于其包含了P2、P3、P4、P5这4个阶段的特征图，筛选出的RoI应该对应到哪一个特征图呢？

对此问题，Mask RCNN的做法与FPN论文中的方法一致，从合适尺度的特征图中切出RoI，大的RoI对应到高语义的特征图中，如P5，小的RoI对应到高分辨率的特征图中，如P3。式（4-7）中给出了精确的计算公式：

$$k = \left\lfloor k_0 + \log_2 \left( \sqrt{wh} / 224 \right) \right\rfloor \quad (4-7)$$

公式中的224代表着ImageNet预训练图片的大小， $k_0$ 默认值为4，表示大小为 $224 \times 224$ 的RoI应该对应的层级为4，计算后做取整处理。

这样的分配方法保证了大的RoI从高语义的特征图上产生，有利于检测大尺度物体，小的RoI从高分辨率的特征图上产生，有利于小物体

的检测。

## 2. RoI Align部分

Faster RCNN原始使用的RoI Pooling存在两次取整的操作，导致RoI选取出来的特征与最开始回归出的位置有一定的偏差，称之为不匹配问题（Missalignment），严重影响了检测或者分割的准确度。

Maks RCNN提出的RoI Align取消了取整操作，而是保留所有的浮点，然后通过双线性插值的方法获得多个采样点的值，再将多个采样点进行最大值的池化，即可得到该点最终的值。

由于使用了采样点与保留浮点的操作，RoI Align获得了更好的性能。这部分在前面已经详细介绍过，在此不再重复。

## 3. 损失任务设计

在得到感兴趣区域的特征后，Mask RCNN增加了Mask分支来进行图像分割，确定每一个像素具体属于哪一个类别。具体实现时，采用了FCN（Fully Convolutional Network）的网络结构，利用卷积与反卷积构建端到端的网络，最后对每一个像素分类，实现了较好的分割效果。由于属于分割领域、在此不对FCN细节做介绍，感兴趣的读者可以查看相关资料。

假设物体检测任务的类别是21类，则Mask分支对于每一个RoI，输出的维度为 $21 \times m \times m$ ，其中 $m \times m$ 表示mask的大小，每一个元素都是二值的，相当于得到了21个二值的mask。在训练时，如果当前RoI的标签类别是5，则只有第5个类别的mask参与计算，其他的类别并不参与计算。这种方法可以有效地避免类间竞争，将分类的任务交给更为专业的分类分支去处理。

增加了Mask分支后，损失函数变为了3部分，如式（4-8）所示。

$$L = L_{cls} + L_{box} + L_{mask} \quad (4-8)$$

公式中前两部分 $L_{cls}$ 、 $L_{box}$ 与Faster RCNN中相同，最后一部分 $L_{mask}$ 代表了分割的损失。对mask上的每一个像素应用Sigmoid函数，送到交叉熵损失中，最后取所有像素损失的平均值作为 $L_{mask}$ 。

Mask RCNN算法简洁明了，在物体检测与实例分割领域都能得到较高的精度，在实际应用中，尤其是涉及多任务时，可以采用Mask RCNN算法。

#### 4.7.4 全卷积网络：R-FCN

Faster RCNN在RoI Pooling后采用了全连接网络来得到分类与回归的预测，这部分全连接网络占据了整个网络结构的大部分参数，而目前越来越多的全卷积网络证明了不使用全连接网络效果会更好，以适应各种输入尺度的图片。

一个很自然的想法就是去掉RoI Pooling后的全连接，直接连接到分类与回归的网络中，但通过实验发现这种方法检测的效果很差，其中一个原因就是基础的卷积网络是针对分类设计的，具有平移不变性，对位置不敏感，而物体检测则对位置敏感。

针对上述“痛点”，微软亚洲研究院的代季峰团队提出了R-FCN（Region-based Fully Convolutional Networks）算法，利用一个精心设计的位置敏感得分图（position-sensitive score maps）实现了对位置的敏感，并且采用了全卷积网络，大大减少了网络的参数量。

图4.16所示为R-FCN的网络结构图，首先R-FCN采用了ResNet-101网络作为Backbone，并在原始的100个卷积层后增加了一个 $1\times 1$ 卷积，将通道数降低为1024。

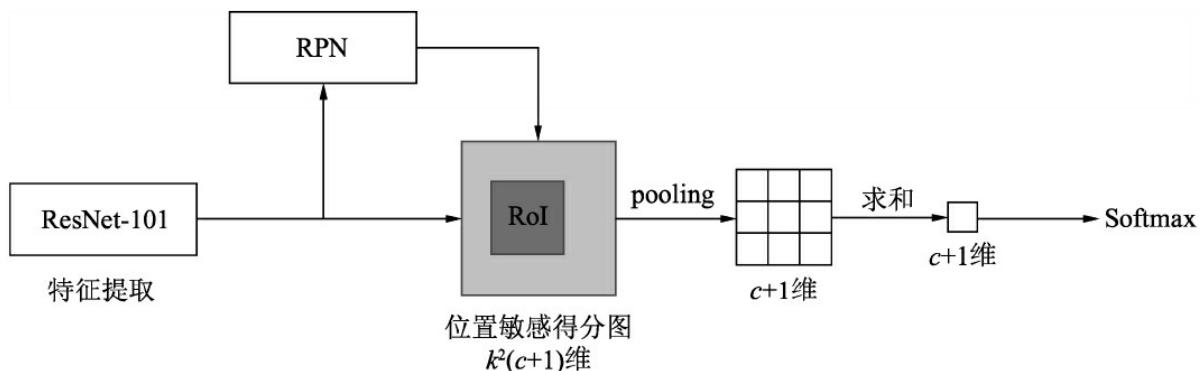


图4.16 R-FCN网络结构图

此外，为了增大后续特征图的尺寸，R-FCN将ResNet-101的下采样率从32降到了16。具体做法是，在第5个卷积组里将卷积的步长从2变为1，同时在这个阶段的卷积使用空洞数为2的空洞卷积以扩大感受野。降低步长增加空洞卷积是一种常用的方法，可以在保持特征图尺寸的同时，增大感受野。

在特征图上进行 $1 \times 1$ 卷积，可以得到位置敏感得分图，其通道数为 $k^2(c+1)$ 。这里的c代表物体类别，一般需要再加上背景这一类别。k的含义是将RoI划分为 $k^2$ 个区域，如图4.17分别展示了k为1、3、5的情况。例如当k=3时，可以将RoI分为左上、中上、右上等9个区域，每个区域对特征区域的信息敏感。因此，位置敏感得分图的通道包含了所有9个区域内所有类别的信息。

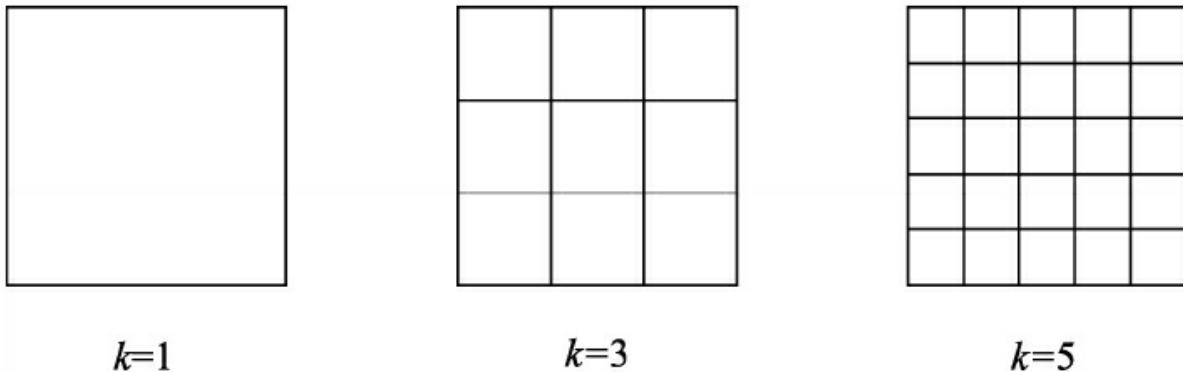


图4.17 ROI划分为 $k^2$ 个区域示意图

对于一个位置敏感得分图上的点，假设其坐标为 $m \times n$ ，通道在右上区域，类别为人，则该点表示当前位置属于人并且在人这个“物体”的右上区域的特征，因此这样就包含了位置信息。

在RPN提供了一个感兴趣区域后，对应到位置敏感得分图上，首先将RoI划分为 $k \times k$ 个网格，如图4.18所示，左侧为将9个不同区域展开后的RoI特征，9个区域分别对应着不同的位置，在Pooling时首先选取其所

在区域的对应位置的特征，例如左上区域只选取其左上角的特征，右下区域只选取右下角的特征，选取后对区域内求均值，最终可形成右侧的一个 $c+1$ 维的 $k \times k$ 特征图。

### 位置敏感得分图上的RoI

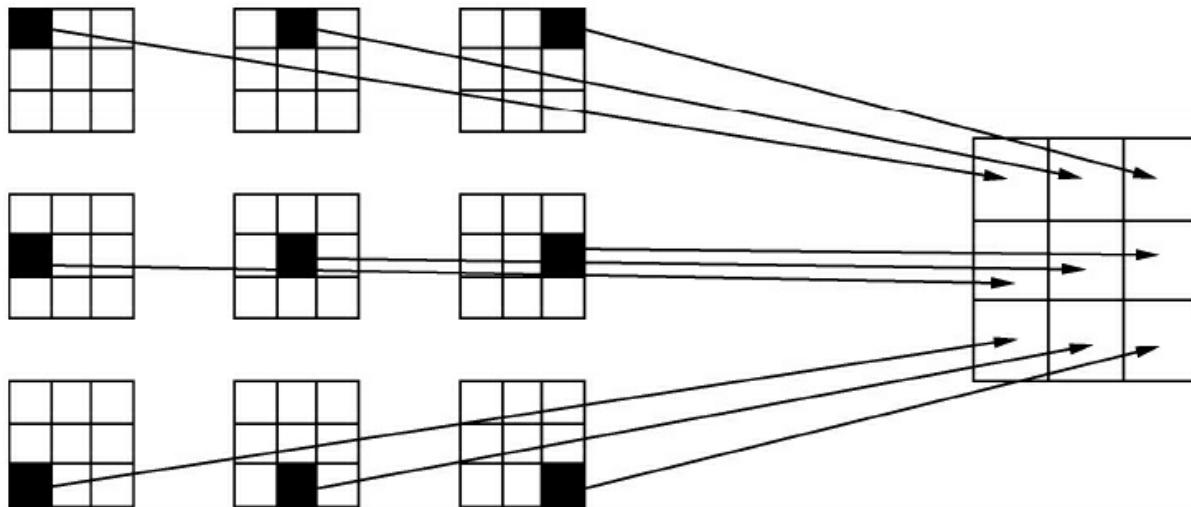


图4.18 位置敏感得分图的Pooling过程

接下来再对这个 $c+1$ 维的 $k \times k$ 特征进行逐通道求和，即可得到 $c+1$ 维的向量，最后进行Softmax即可完成这个RoI的分类预测。

至于RoI的位置回归，则与分类很相似，只不过位置敏感得分图的通道数为 $k^2(c+1)$ ，而回归的敏感回归图的通道数为 $k^2 \times 4$ ，按照相同的方法进行Pooling，可形成通道数为4的 $k \times k$ 特征，求和可得到 $1 \times 4$ 的向量，即为回归的预测。

由于R-FCN去掉了全连接层，并且整个网络都是共享计算的，因此速度很快。此外，由于位置敏感得分图的存在，引入了位置信息，因此R-FCN的检测效果也更好。

#### 4.7.5 级联网络：Cascade RCNN

在前面的讲解中可以得知，在得到一个RoI后，Faster RCNN通过RoI与标签的IoU值来判断该RoI是正样本还是负样本，默认的IoU阈值为0.5，这个阈值是一个超参数，对于检测的精度有较大影响。

如何选择合适的阈值是一个矛盾的问题。一方面，阈值越高，选出的RoI会更接近真实物体，检测器的定位会更加准确，但此时符合条件的RoI会变少，正、负样本会更加不均衡，容易导致训练过拟合；另一方面，阈值越低，正样本会更多，有利于模型训练，但这时误检也会增多，从而增大了分类的误差。

对于阈值的问题，通过实验可以发现两个现象：

- 一个检测器如果采用某个阈值界定正负样本时，那么当输入Proposal的IoU在这个阈值附近时，检测效果要比基于其他阈值时好，也就是很难让一个在指定阈值界定正、负样本的检测模型对所有IoU的输入Proposal检测效果都最佳。
- 经过回归之后的候选框与标签之间的IoU会有所提升。

基于以上结果，2018年CVPR上的Cascade RCNN算法通过级联多个检测器来不断优化结果，每个检测器都基于不同的IoU阈值来界定正负样本，前一个检测器的输出作为后一个检测器的输入，并且检测器越靠后，IoU的阈值越高。

级联检测器可以有多种形式，如图4.19所示为迭代式的边框回归模型示意图，图中的Pooling代表了RoI Pooling过程，H1表示RCNN部分网络，C与B分别表示分类与回归部分网络。从图中可以看出，这种方法

将前一个回归网络输出的边框作为下一个检测器的输入继续进行回归，连续迭代3次才得到结果。

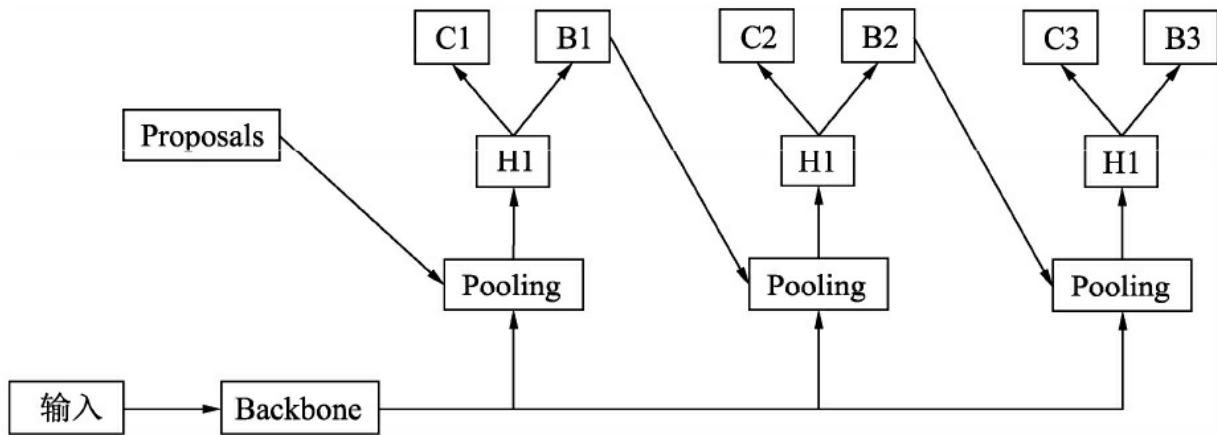


图4.19 迭代式边框回归模型示意图

从前面的实验可以得知，经过一个固定IoU阈值的检测器后，边框的IoU会提升，分布也发生了变化，即越来越靠近真实物体。如果下一个检测器仍然还是这个IoU阈值的话，显然不是一个最优的选择，这也是上述这种级联器的问题所在。

图4.20是另一种多个检测器的组合方式，称为Integral Loss。图中H1、H2与H3分别代表不同的IoU阈值界定正负样本的检测器，当阈值较高时，预测的边框会更为精准，但会损失一些正样本。这种方法中多个检测器相互独立，没有反馈优化的思想，仅仅是利用了多个IoU阈值的检测器。

图4.21的结构则是Cascade RCNN采用的方法，可以看到每一个检测器的边框输出作为下一个检测器的输入，并且检测器的IoU阈值是逐渐提升的，因此这种方法可以逐步过滤掉一些误检框，并且提升边框的定位精度。

总体来看，Cascade RCNN算法深入探讨了IoU阈值对检测器性能的

影响，并且在不增加任何tricks的前提下，在多个数据集上都有了明显的精度提升，是一个性能优越的高精度物体检测器。

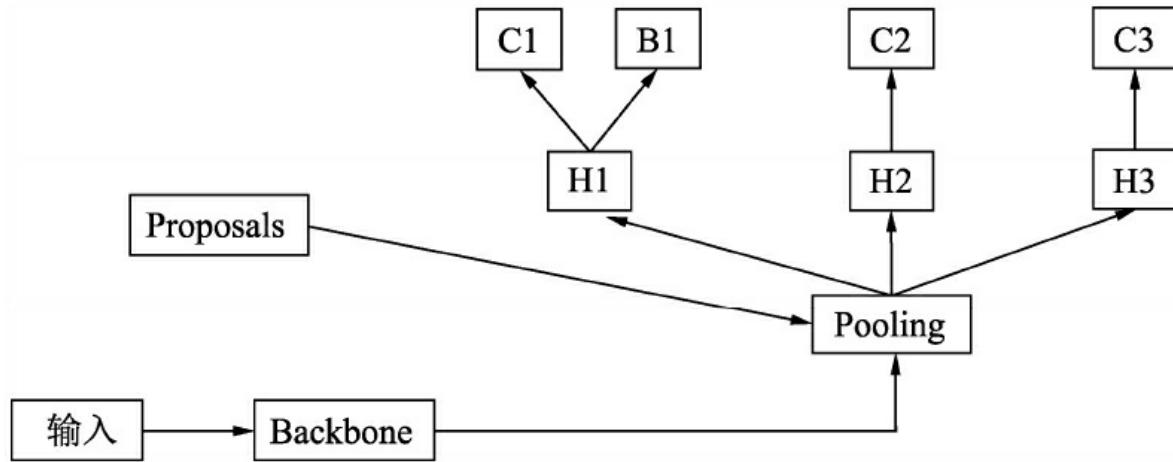


图4.20 Integral loss模型示意图

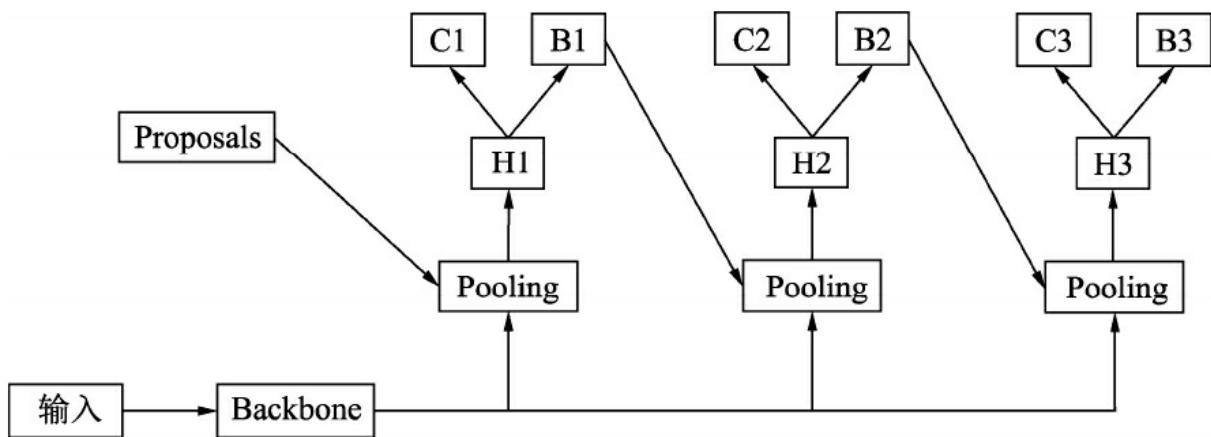


图4.21 Cascade RCNN级联检测器示意图

## 4.8 总结

本章从代码的角度详细介绍了Faster RCNN检测的原理与实现。Faster RCNN作为当前物体检测影响最为深远的检测框架，其设计的每一个细节、参数都值得我们好好推敲，也希望读者能够自己动手调整网络细节和超参数，相信你会有更深的理解。在Faster RCNN的原理基础上，本章进一步分析了其优缺点，并给出了Mask RCNN等多种改进方案。

Faster RCNN是极为典型的两阶检测算法，过程稍显复杂一些，下一章的“主角”SSD网络框架则以更简洁、优雅的方法实现物体检测。

## 第5章 单阶多层检测器：SSD

对于物体检测任务，第4章的Faster RCNN算法采用了两阶的检测架构，即首先利用RPN网络进行感兴趣区域生成，然后再对该区域进行类别的分类与位置的回归，这种方法虽然显著提升了精度，但也限制了检测速度。YOLO算法利用回归的思想，使用一阶网络直接完成了物体检测，速度很快，但是精度有了明显的下降。

在此背景下，SSD (Single Shot Multibox Detecor) 算法借鉴了 Faster RCNN 与 YOLO 的思想，在一阶网络的基础上使用了固定框进行区域生成，并利用了多层的特征信息，在速度与检测精度上都有了一定的提升。

本章将首先介绍SSD方法的主要思想，然后针对重要的结构模块，从代码层面一一解读其实现方法，最后将分析SSD的优缺点，并介绍一些经典的改进算法。

## 5.1 SSD总览

作为一阶网络，SSD算法从多个角度对物体检测做了创新，是一个既优雅又高效的检测网络，本节将从算法层面简要介绍SSD的总体过程。

### 5.1.1 SSD的算法流程

SSD算法的算法流程如图5.1所示，输入图像首先经过了VGGNet的基础网络，在此之上又增加了几个卷积层，然后利用 $3\times 3$ 的卷积核在6个大小与深浅不同的特征层上进行预测，得到预选框的分类与回归预测值，最后直接预测出结果，或者求得网络损失。

SSD的算法思想，主要可以分为4个方面：

·数据增强：SSD在数据部分做了充分的数据增强工作，包括光学变换与几何变换等，极大限度地扩充了数据集的丰富性，从而有效提升了模型的检测精度。

·网络骨架：SSD在原始VGGNet的基础上，进一步延伸了4个卷积模块，最深处的特征图大小为 $1\times 1$ ，这些特征图具有不同的尺度与感受野，可以负责检测不同尺度的物体。

·PriorBox与多层特征图：与Faster RCNN类似，SSD利用了固定大小与宽高的PriorBox作为区域生成，但与Faster RCNN不同的是，SSD不是只在一个特征图上设定预选框，而是在6个不同尺度上都设立预选框，并且在浅层特征图上设立较小的PriorBox来负责检测小物体，在深层特征图上设立较大的PriorBox来负责检测大物体。

·正、负样本的选取与损失计算：利用 $3\times 3$ 的卷积在6个特征图上进行特征的提取，并分为分类与回归两个分支，代表所有预选框的预测值，随后进行预选框与真实框的匹配，利用IoU筛选出正样本与负样本，最终计算出分类损失与回归损失。

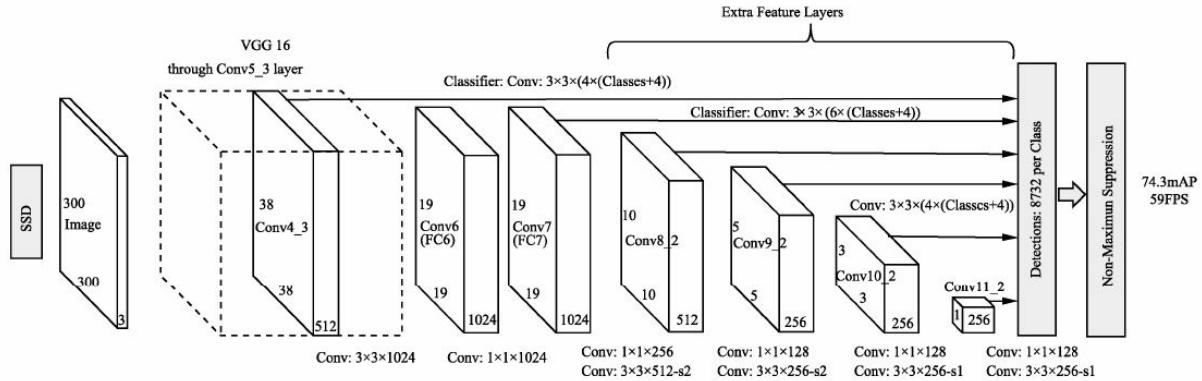


图5.1 SSD的算法流程图

由整个过程可以看出，SSD只进行了一次框的预测与损失计算，属于一阶网络。由于利用了多个特征图，SSD实现了较好的检测精度。接下来将会详细介绍SSD的以上4个方面。

## 5.1.2 代码准备工作

本章所使用的环境在第1章已有说明，如果还没配置好PyTorch环境的读者，请前往第1章按相应说明进行配置。本章使用的SSD源码来自于amdegroot的高质量复现，笔者迁移到了本书的代码中，使用方式如下：

(1) 首先从GitHub上下载本章所用的代码：

```
git clone git@github.com:dongdonghy/Detection-PyTorch-Notebook.git  
cd Detection-PyTorch-Notebook/chapter5/ssd-pytorch
```

(2) 下载PASCAL VOC 2012数据集。

(3) 从下面的网址下载预训练模型。

```
mkdir weights  
cd weights  
wget https://s3.amazonaws.com/amdegroot-models/vgg16_reducedfc.pth
```

(4) 进行网络训练，如下：

```
python3 train.py
```

## 5.2 数据预处理

PyTorch对于数据的加载与处理提供了一些标准类，可以很灵活地实现各种功能。SSD对于数据的预处理，尤其是数据增强做了丰富的处理，也实现了非常优越的性能，因此将这部分单独拿出来细讲。

### 5.2.1 加载PASCAL数据集

由2.5.2节可知，PyTorch加载数据集主要有3步：

(1) 继承Dataset类，重写 `len()` 和 `getitem()` 函数并实例化，这样就可以方便地进行数据集的迭代。

(2) 为了满足实际训练要求、增强数据的丰富性，还要选择合适的数据增强方法，这部分内容将在下一节细讲。

(3) 继承DataLoader类，添加batch和多线程等功能。

在训练脚本train.py中，加载数据集的示例代码如下：

---

```
# 利用VOCDetection重写了Dataset类，并传入了所需的数据变换transform
dataset = VOCDetection(root=args.dataset_root,
                       transform=SSDAugmentation(cfg['min_dim'], MEANS))
# 利用DataLoader实现了批量处理、打乱、多线程加载等功能
data_loader = data.DataLoader(dataset, args.batch_size,
                             num_workers=args.num_workers,
                             shuffle=True, collate_fn=detection_collate,
                             pin_memory=True)
```

---

## 5.2.2 数据增强

SSD做了丰富的数据增强策略，这部分为模型的mAP带来了8.8%的提升，尤其是对于小物体和遮挡物体等难点，数据增强起到了非常重要的作用。

SSD的数据增强整体流程如图5.2所示，总体上包括光学变换与几何变换两个过程。光学变换包括亮度和对比度等随机调整，可以调整图像像素值的大小，并不会改变图像尺寸；几何变换包括扩展、裁剪和镜像等操作，主要负责进行尺度上的变化，最后再进行去均值操作。大部分操作都是随机的过程，尽可能保证数据的丰富性。

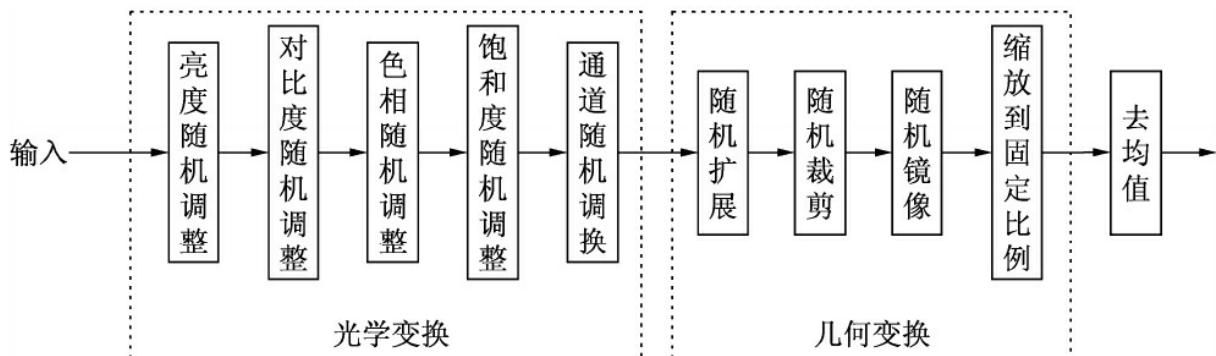


图5.2 SSD数据增强流程

数据增强的总体流程代码如下，源文件见utils/augmentations.py。

```
class SSDAugmentation(object):
    def __init__(self, size=300, mean=(104, 117, 123)):
        self.mean = mean
        self.size = size
        self.augment = Compose([
            # 首先将图像像素值从整型变成浮点型
            ConvertFromInts(),
            # 将标签中的边框从比例坐标变换为真实坐标
            ToAbsoluteCoords(),
```

```
# 进行亮度、对比度、色相与饱和度的随机调整，然后随机调换通道
PhotometricDistort(),
Expand(self.mean),           # 随机扩展图像大小，图像仅靠右下方
RandomSampleCrop(),          # 随机裁剪图像
RandomMirror(),              # 随机左右镜像
ToPercentCoords(),           # 从真实坐标变回比例坐标
Resize(self.size),            # 缩放到固定的300×300大小
SubtractMeans(self.mean)     # 最后进行去均值
])
```

---

将像素值从整型变到浮点型，边框从比例坐标变换到真实坐标，这两种变换较为简单，这里不展开叙述。接下来重点介绍光学变换与几何变换这两个重要的变换方法。

## 1. 光学变换

首先是进行亮度调整，具体方法是以0.5的概率为图像中的每一个点加一个实数，该实数随机选取于[-32,32)区间中。具体可见如下RandomBrightness类的实现。

```
class RandomBrightness(object):
    def __init__(self, delta=32):
        self.delta = delta
    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            # 随机选取一个位于[-32, 32)区间的数，相加到图像上
            delta = random.uniform(-self.delta, self.delta)
            image += delta
        return image, boxes, labels
```

---

接下来是对比度、色相与饱和度的随机调整。色相的随机调整与亮度很类似，都是随机地加一个数，而对比度与饱和度则是随机乘一个数。另外，对色相与饱和度的调整是在HSV色域空间进行的。由于此三者与亮度很相似，代码不再单独给出。

关于以上三者的调整顺序，SSD也给了一个随机处理，即有一半的

概率对比度在另外两者之前，另一半概率则是对比度在另外两者之后。

光学变换的最后一项是添加随机的光照噪声，具体做法是随机交换RGB三个通道的值，具体实现如RandomLightingNoise()类所示。

---

```
class RandomLightingNoise(object):
    def __init__(self):
        self.perms = ((0, 1, 2), (0, 2, 1),
                      (1, 0, 2), (1, 2, 0),
                      (2, 0, 1), (2, 1, 0))
    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            # 随机选取一个通道的交换顺序，交换图像三个通道的值
            swap = self.perms[random.randint(len(self.perms))]
            shuffle = SwapChannels(swap)                  # shuffle channels
            image = shuffle(image)
        return image, boxes, labels
```

---

## 2. 几何变换

在几何变换中，首先进行的是尺度的随机扩展。扩展的具体过程是随机选择一个在[1,4]区间的数作为扩展比例，将原图像放在扩展后图像的右下角，其他区域填入每个通道的均值，即[104,117,123]，如图5.3所示。

## 随机扩展区域

原图

图5.3 SSD的随机扩展方法

代码实现时，首先利用random函数保证有一半的概率进行扩展，然后随机选择一个位于[1,4)区间的比例值，新建一个图像，将均值与原图像素值依次赋予新图，最后再将对应的边框也进行平移，即完成了扩展流程。

```
class Expand(object):
    def __init__(self, mean):
        self.mean = mean
    def __call__(self, image, boxes, labels):
        if random.randint(2):
            return image, boxes, labels
        # 求取原图像在新图像中的左上角坐标值
        height, width, depth = image.shape
        ratio = random.uniform(1, 4)
        left = random.uniform(0, width*ratio - width)
        top = random.uniform(0, height*ratio - height)
        # 建立新的图像，并依次赋值
        expand_image = np.zeros(
```

```
(int(height*ratio), int(width*ratio), depth),
    dtype=image.dtype)
expand_image[:, :, :] = self.mean
expand_image[int(top):int(top + height),
            int(left):int(left + width)] = image
image = expand_image
# 对边框也进行相应变换
boxes = boxes.copy()
boxes[:, :2] += (int(left), int(top))
boxes[:, 2:] += (int(left), int(top))
return image, boxes, labels
```

---

随机扩展后，紧接着进行随机裁剪。SSD的裁剪策略是从图像中随机裁剪出一块，需要保证该图像块至少与一个物体边框有重叠，重叠的比例从{0.1、0.3、0.7、0.9}中随机选取，同时至少有一个物体的中心点落在该图像块中。

这种随机裁剪的好处在于，首先每一个图形块都要有物体，可以过滤掉不包含明显真实物体的图像；同时，不同的重叠比例也极大地丰富了训练集，尤其是针对物体遮挡的情况。由于处理步骤较多，在此不再针对裁剪进行代码说明。

裁剪后进行的是随机的图像镜像，通常是图像的左右翻转，这是一种简单又极为实用的图像增强手段，在多个物体检测算法中都会用到。

---

```
class RandomMirror(object):
    def __call__(self, image, boxes, classes):
        _, width, _ = image.shape
        if random.randint(2):
            # 这里的::代表反向，即将每一行的数据反向遍历，完成镜像
            image = image[:, ::-1]
            boxes = boxes.copy()
            boxes[:, 0::2] = width - boxes[:, 2::-2]
        return image, boxes, classes
```

---

在完成镜像后，最后一步几何变换是固定缩放，默认使用了

300×300的输入大小。这里300×300的固定输入大小是经过精心设计的，可以恰好满足后续特征图的检测尺度，例如最后一层的特征图大小为1×1，负责检测的尺度则为0.9。原论文作者也给出了另外一个更精确的500×500输入的版本。

经过上述光学与几何两个变换后，最后一步是常见的去均值处理，具体操作是减去每个通道的均值，具体代码如下：

---

```
class SubtractMeans(object):
    def __init__(self, mean):
        self.mean = np.array(mean, dtype=np.float32)
    def __call__(self, image, boxes=None, labels=None):
        image = image.astype(np.float32)
        image -= self.mean
        return image.astype(np.float32), boxes, labels
```

---

## 5.3 网络架构

SSD使用VGGNet作为基础Backbone，然后为了提取更高语义的特征，在VGGNet后又增加了多个卷积层，最后利用多个特征图进行边框的特征提取。得到深层网络后，SSD的计算过程如图5.4所示。

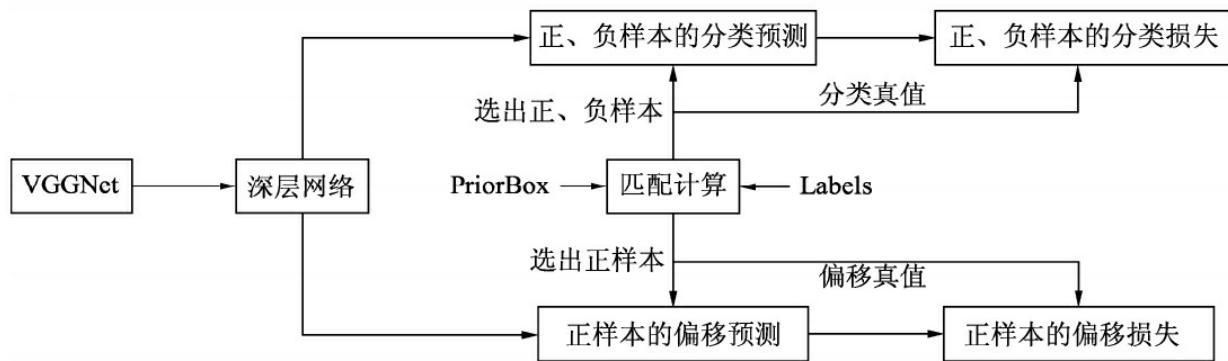


图5.4 SSD算法的计算过程

首先，利用人工设置的一系列PriorBox与标签里的边框进行匹配，并根据重叠程度筛选出正、负样本，得到分类与偏移的真值，这一步类似于Faster RCNN中的匹配过程。筛选出正、负样本后，从深层网络中拿出对应的样本的分类预测值与偏移预测值，与真值计算分类和偏移的损失。

本节将从代码层面详细介绍SSD算法的实现过程。

### 5.3.1 基础VGG结构

SSD采用了VGG 16作为基础网络，并在之上进行了一些改善，如图5.5所示。输入图像经过预处理后大小固定为 $300 \times 300$ ，首先经过VGG 16网络的前13个卷积层，然后利用两个卷积Conv 6与Conv 7取代了原来的全连接网络，进一步提取特征。

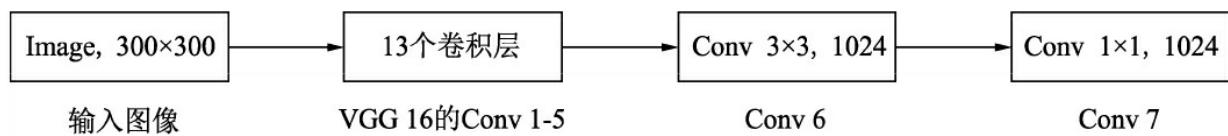


图5.5 SSD的基础VGG结构

针对SSD的基础网络，有以下两点需要注意：

- 原始的VGG 16的池化层统一大小为 $2 \times 2$ ，步长为2，而在SSD中，Conv 5后接的Maxpooling层池化大小为3，步长为1，这样做可以在增加感受野的同时，维持特征图的尺寸不变。

- Conv 6中使用了空洞数为6的空洞卷积，其padding也为6，这样做同样也是为了增加感受野的同时保持参数量与特征图尺寸的不变。

利用PyTorch构造该基础网络时，只需要在官方VGG 16的基础上进行一些修改即可。SSD的基础网络代码主要在ssd.py中，具体如下：

---

```
# 搭建VGG基础网络的函数
def vgg(cfg, i, batch_norm=False):
    layers = []
    in_channels = i
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        elif v == 'C':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2, ceil_mode=True)]
```

```
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
        pool5 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        conv6 = nn.Conv2d(512, 1024, kernel_size=3, padding=6, dilation=6)
        conv7 = nn.Conv2d(1024, 1024, kernel_size=1)
        layers += [pool5, conv6,
                   nn.ReLU(inplace=True), conv7, nn.ReLU(inplace=True)]
    return layers
# 这里的base为VGG 16前13个卷积层构造, M代表maxpooling, C代表ceil_mode为True
base = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'C', 512, 512, 512, 'M',
        512, 512, 512]
# 这里示例利用vgg(base, 3)即可实现基础网络的构造
vgg_base = vgg(base, 3)
```

---

### 5.3.2 深度卷积层

在VGG 16的基础上，SSD进一步增加了4个深度卷积层，用于更高语义信息的提取，如图5.6所示。可以看出，Conv 8的通道数为512，而Conv 9、Conv 10与Conv 11的通道数都为256。从Conv 7到Conv 11，这5个卷积后输出特征图的尺寸依次为 $19 \times 19$ 、 $10 \times 10$ 、 $5 \times 5$ 、 $3 \times 3$ 和 $1 \times 1$ 。

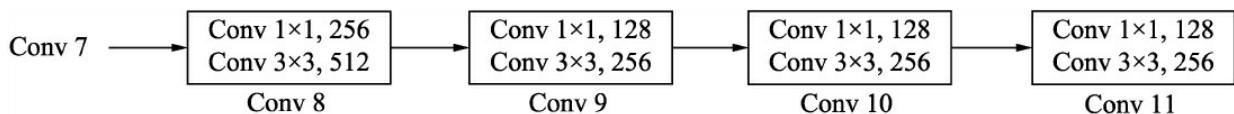


图5.6 SSD增加的深度卷积层

为了降低参数量，在此使用了 $1 \times 1$ 卷积先降低通道数为该层输出通道数的一半，再利用 $3 \times 3$ 卷积进行特征提取。

利用PyTorch可以很方便地实现该深度卷积层，源代码文件为ssd.py，具体如下：

---

```
# 额外的深度卷积层构造函数
def add_extras(cfg, i, batch_norm=False):
    # Extra layers added to VGG for feature scaling
    layers = []
    in_channels = i
    flag = False
    for k, v in enumerate(cfg):
        if in_channels != 'S':
            if v == 'S':
                layers += [nn.Conv2d(in_channels, cfg[k + 1],
                                   kernel_size=(1, 3)[flag], stride=2, padding=1)]
            else:
                layers += [nn.Conv2d(in_channels, v, kernel_size=(1, 3)[flag])]
            flag = not flag
            in_channels = v
    return layers
# 额外部分的卷积通道数，S代表了步长为2，其余卷积层默认步长为1
extras = [256, 'S', 512, 128, 'S', 256, 128, 256, 128, 256]
```

```
# 调用示例，直接调用add_extras函数即可完成该部分网络
conv_extras = add_extras(extras, 1024)
```

---

### 5.3.3 PriorBox与边框特征提取网络

与Faster RCNN的Anchor类似，SSD采用了PriorBox来进行区域生成。不同的是，Faster RCNN首先在第一个阶段对固定的Anchor进行了位置修正与筛选，得到感兴趣区域后，在第二个阶段再对该区域进行分类与回归，而SSD直接将固定大小宽高的PriorBox作为先验的兴趣区域，利用一个阶段完成了分类与回归。

PriorBox本质上是在原图上的一系列矩形框，如图5.7所示。某个特征图上的一个点根据下采样率可以得到在原图的坐标，SSD先验性地提供了以该坐标为中心的4个或6个不同大小的PriorBox，然后利用特征图的特征去预测这4个PriorBox的类别与位置偏移量。

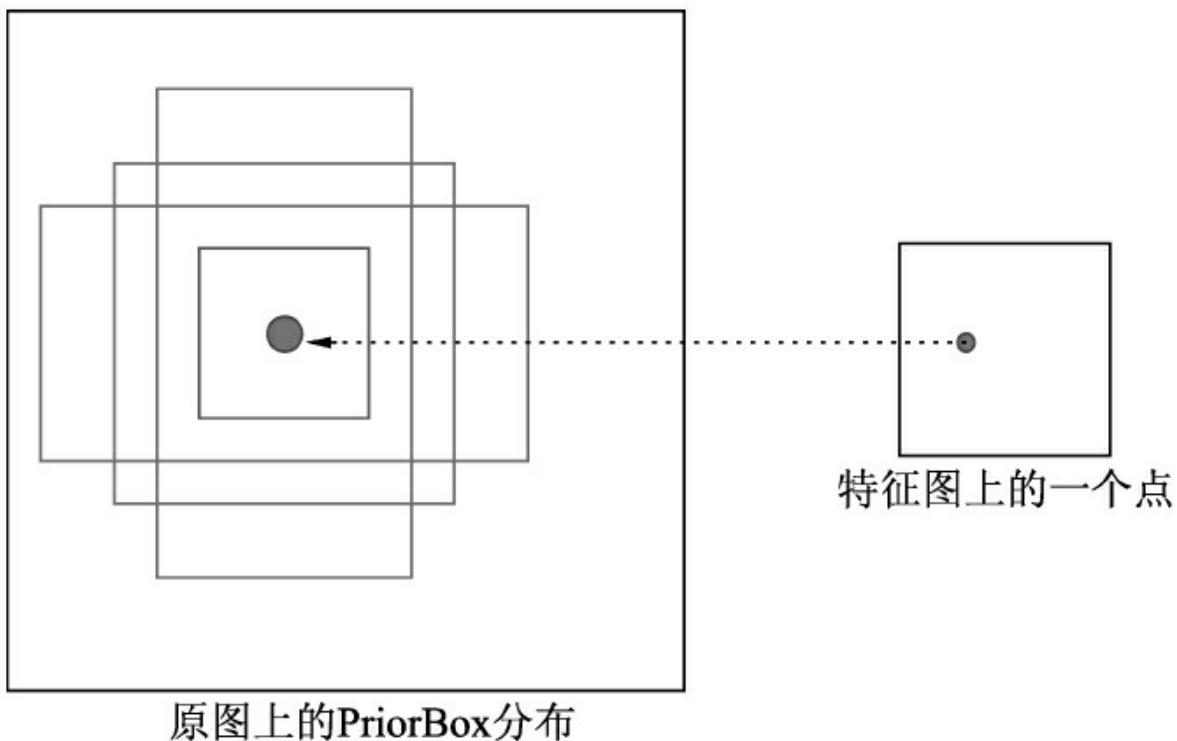


图5.7 SSD的PriorBox的分布示例

在Faster RCNN中，所有Anchors对应的特征都来源于同一个特征图，而该层特征的感受野相同，很难处理被检测物体的尺度变化较大的情况，多个大小宽高的Anchors能起到的作用也有限。

从前几章的讲解可以得出，在深度卷积网络中，浅层的特征图拥有较小的感受野，深层的特征图拥有较大的感受野，因此SSD充分利用了这个特性，使用了多层特征图来做物体检测，浅层的特征图检测小物体，深层的特征图检测大物体，如图5.8所示。

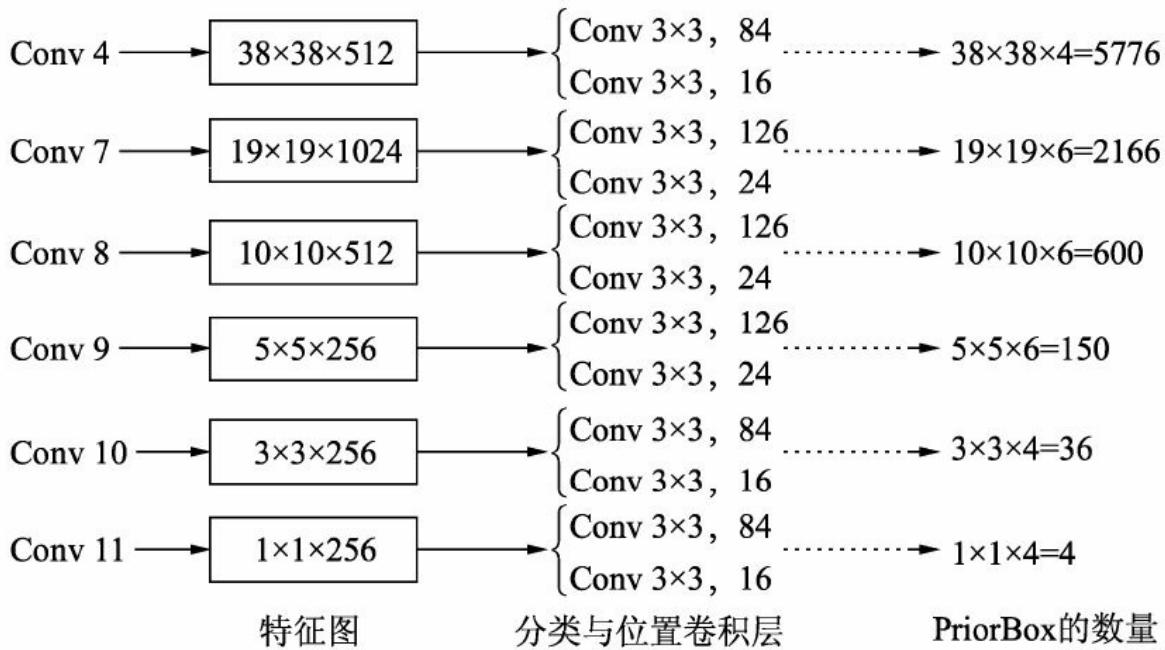


图5.8 SSD的分类与回归网络

从图5.8中可以看出，SSD使用了第4、7、8、9、10和11这6个卷积层得到的特征图，这6个特征图尺寸越来越小，而其对应的感受野越来越大。6个特征图上的每一个点分别对应4、6、6、6、4、4个PriorBox。接下来分别利用 $3 \times 3$ 的卷积，即可得到每一个PriorBox对应的类别与位置预测量。

举个例子，第8个卷积层得到的特征图大小为 $10 \times 10 \times 512$ ，每个点对

应6个PriorBox，一共有600个PriorBox。由于采用的PASCAL VOC数据集的物体类别为21类，因此 $3 \times 3$ 卷积后得到的类别特征维度为 $6 \times 21 = 126$ ，位置特征维度为 $6 \times 4 = 24$ 。

如何确定每一个特征图PriorBox的具体大小呢？由于越深的特征图拥有的感受野越大，因此其对应的PriorBox也应越来越大，SSD采用了公式（5-1）来计算每一个特征图对应的PriorBox的尺度。

$$S_k = S_{\min} + \frac{S_{\max} - S_{\min}}{5}(k-1) \quad k \in [1, 6] \quad (5-1)$$

公式中K的取值为1、2、3、4、5、6，分别对应着SSD中的第4、7、8、9、10、11个卷积层。 $S_k$ 代表这一层对应的尺度， $S_{\min}$ 为0.2， $S_{\max}$ 为0.9，分别表示最浅层与最深层对应的尺度与原图大小的比例，即第4个卷积层得到的特征图对应的尺度为0.2，第11个卷积层得到的特征图对应的尺度为0.9。

基于每一层的基础尺度 $S_k$ ，对于第1、5、6个特征图，每个点对应了4个PriorBox，因此其宽高分别为 $\{\frac{S_k}{\sqrt{2}}, \sqrt{2}S_k\}$ 、 $\{\sqrt{2}S_k, \frac{S_k}{\sqrt{2}}\}$ 与 $\{\sqrt{S_k}S_{k+1}, \sqrt{S_k}S_{k+1}\}$ ，而对于第2、3、4个特征图，每个点对应了6个PriorBox，则在上述4个宽高值上再增加 $\{\frac{S_k}{\sqrt{3}}, \sqrt{3}S_k\}$ 和 $\{\sqrt{3}S_k, \frac{S_k}{\sqrt{3}}\}$ 这两种比例的框。

下面利用代码详细介绍如何生成每一层所需的PriorBox，代码位于layers/functions/prior\_box.py中。

```

# 生成所有的 PriorBox，需要每一个特征图的信息
def forward(self):
    mean = []
    for k, f in enumerate(self.feature_maps):
        for i, j in product(range(f), repeat=2):
            # f_k 为每个特征图的尺寸
            f_k = self.image_size / self.steps[k]
            # 求取每个 box 的中心坐标
            cx = (j + 0.5) / f_k
            cy = (i + 0.5) / f_k
            # 对应{S_k, S_k}大小的 PriorBox
            s_k = self.min_sizes[k] / self.image_size
            mean += [cx, cy, s_k, s_k]
            # 对应{sqrt(S_k S_{k+1}), sqrt(S_k S_{k+1})}大小的 PriorBox
            s_k_prime = sqrt(s_k * (self.max_sizes[k] / self.image_size))
            mean += [cx, cy, s_k_prime, s_k_prime]
            # 剩余的比例为 2、1/2、3、1/3 的 PriorBox

            for ar in self.aspect_ratios[k]:
                mean += [cx, cy, s_k * sqrt(ar), s_k / sqrt(ar)]
                mean += [cx, cy, s_k / sqrt(ar), s_k * sqrt(ar)]
    output = torch.Tensor(mean).view(-1, 4)
    if self.clip:
        output.clamp_(max=1, min=0)
    return output

```

有了多层的特征图，利用 $3 \times 3$ 卷积即可完成如图5.7所示的提取过程。下面为所有边框特征提取的PyTorch实现。代码文件为ssd.py。

---

```

def multibox(vgg, extra_layers, cfg, num_classes):
    loc_layers = []
    conf_layers = []
    vgg_source = [21, -2]
    for k, v in enumerate(vgg_source):
        loc_layers += [nn.Conv2d(vgg[v].out_channels,
                               cfg[k] * 4, kernel_size=3, padding=1)]
        conf_layers += [nn.Conv2d(vgg[v].out_channels,
                               cfg[k] * num_classes, kernel_size=3, padding=1)]
    for k, v in enumerate(extra_layers[1::2], 2):
        loc_layers += [nn.Conv2d(v.out_channels, cfg[k]
                               * 4, kernel_size=3, padding=1)]
        conf_layers += [nn.Conv2d(v.out_channels, cfg[k]
                               * num_classes, kernel_size=3, padding=1)]
    return vgg, extra_layers, (loc_layers, conf_layers)
# 每个特征图上一个点对应的PriorBox数量
mbox = [4, 6, 6, 6, 4, 4]
# 利用上面的vgg_base与conv_extras网络，生成类别与位置预测网络head_

```

```
base_, extras_, head_ = multibox(vgg_base, conv_extras, mbox, num_classes)
```

---

综上所述，这一小节一方面生成了共计8732个PriorBox的位置信息，同时也利用卷积网络提取了这8732个PriorBox的特征。

### 5.3.4 总体网络计算过程

上一节讲解了SSD的PriorBox与特征提取网络。为了更好地梳理网络的前向过程，本节将从代码角度讲述SSD网络的整个前向过程，以便让读者理解起来更加清晰。代码文件为ssd.py。

---

```
def forward(self, x):
    # sources保存特征图, loc与conf保存所有PriorBox的位置与类别预测特征
    sources = list()
    loc = list()
    conf = list()
    # 对输入图像卷积到conv4_3, 将特征添加到sources中
    for k in range(23):
        x = self.vgg[k](x)
        s = self.L2Norm(x)
        sources.append(s)
    # 继续卷积到Conv 7, 将特征添加到sources中
    for k in range(23, len(self.vgg)):
        x = self.vgg[k](x)
        sources.append(x)
    # 继续利用额外的卷积层计算, 并将特征添加到sources中
    for k, v in enumerate(self.extras):
        x = F.relu(v(x), inplace=True)
        if k % 2 == 1:
            sources.append(x)
    # 对sources中的特征图利用类别与位置网络进行卷积计算, 并保存到loc与conf中
    for (x, l, c) in zip(sources, self.loc, self.conf):
        loc.append(l(x).permute(0, 2, 3, 1).contiguous())
        conf.append(c(x).permute(0, 2, 3, 1).contiguous())
    loc = torch.cat([o.view(o.size(0), -1) for o in loc], 1)
    conf = torch.cat([o.view(o.size(0), -1) for o in conf], 1)
    # 对于训练来说, output包括了loc与conf的预测值及PriorBox的信息
    output = (
        loc.view(loc.size(0), -1, 4),
        conf.view(conf.size(0), -1, self.num_classes),
        self.priors
    )
    return output
```

---

## 5.4 匹配与损失求解

上一节的卷积网络得到了所有PriorBox的预测值与边框位置，为了得到最终的结果，还需要进行边框的匹配及损失计算。

SSD的这部分网络后处理可以分为4步：首先按照一定的原则，对所有的PriorBox赋予正、负样本的标签，并确定对应的真实物体标签，以方便后续损失的计算；有了对应的真值后，即可计算框的定位损失，这部分只需要正样本即可；同时，为了克服正、负样本的不均衡，进行难样本挖掘，筛选出数量是正样本3倍的负样本；最后，计算筛选出的正、负样本的类别损失，完成整个网络向前计算的全过程。

### 5.4.1 预选框与真实框的匹配

在求得8732个PriorBox坐标及对应的类别、位置预测后，首先要做的是为每一个PriorBox贴标签，筛选出符合条件的正样本与负样本，以便进行后续的损失计算。判断依据与Faster RCNN相同，都是通过预测与真值的IoU值来判断。

SSD处理匹配过程时遵循以下4个原则：

·在判断正、负样本时，IoU阈值设置为0.5，即一个PriorBox与所有真实框的最大IoU小于0.5时，判断该框为负样本。

·判断对应关系时，将PriorBox与其拥有最大IoU的真实框作为其位置标签。

·与真实框有最大IoU的PriorBox，即使该IoU不是此PriorBox与所有真实框IoU中最大的IoU，也要将该Box对应到真实框上，这是为了保证真实框的Recall。

·在预测边框位置时，SSD与Faster RCNN相同，都是预测相对于预选框的偏移量，因此在求得匹配关系后还需要进行偏移量计算，具体公式参照4.4.2节。

具体的匹配过程在layers/box\_utils.py中，主要是下面的match函数。

---

```
# 输入包括IoU阈值、真实边框位置、预选框、方差、真实边框类别
# 输出为每一个预选框的类别，保存在conf_t中，对应的真实边框位置保存在loc_t中
def match(threshold, truths, priors, variances, labels, loc_t, conf_t, idx):
    # 注意，这里的truth是最大或最小值的形式，而prior是中心点与宽高形式
    # 求取真实框与预选框的IoU
    overlaps = jaccard(truths, point_form(priors))
    ....
```

```
# 将每一个真实框对应的最佳PriorBox的IoU设置为2，确保是最优的PriorBox
best_truth_overlap.index_fill_(0, best_prior_idx, 2)
# 对于每一个真实框，其拥有最大IoU的PriorBox要对应到该真实框上，即使在这个PriorBox
# 中该真实框不是最大的IoU，这是为了保证Recall
for j in range(best_prior_idx.size(0)):
    best_truth_idx[best_prior_idx[j]] = j
# 每一个PriorBox对应的真实框的位置
matches = truths[best_truth_idx]
# 每一个PriorBox对应的真实类别
conf = labels[best_truth_idx] + 1
# 如果一个PriorBox对应的最大IoU小于0.5，则视为负样本
conf[best_truth_overlap < threshold] = 0
# 进一步计算定位的偏移真值
loc = encode(matches, priors, variances)
loc_t[idx] = loc
conf_t[idx] = conf
```

---

## 5.4.2 定位损失的计算

在完成匹配后，由于有了正、负样本及每一个样本对应的真实框，因此可以进行定位的损失计算。与Faster RCNN相同，SSD使用了smoothL10函数作为定位损失函数，并且只对正样本计算。具体公式可参见4.4.5节。

---

```
# 计算所有正样本的定位损失，负样本不需要定位损失
pos = conf_t > 0
num_pos = pos.sum(dim=1, keepdim=True)
# 将pos_idx扩展为[32, 8732, 4]，正样本的索引
pos_idx = pos.unsqueeze(pos.dim()).expand_as(loc_data)
# 正样本的定位预测值
loc_p = loc_data[pos_idx].view(-1, 4)
# 正样本的定位真值
loc_t = loc_t[pos_idx].view(-1, 4)
# 所有正样本的定位损失
loss_l = F.smooth_l1_loss(loc_p, loc_t, size_average=False)
```

---

### 5.4.3 难样本挖掘

在完成正、负样本匹配后，由于一般情况下一张图片的物体数量不会超过100，因此会存在大量的负样本。如果这些负样本都考虑则在损失反传时，正样本能起到的作用就微乎其微了，因此需要进行难样本的挖掘。这里的难样本是针对负样本而言的。

Faster RCNN通过限制正负样本的数量来保持正、负样本均衡，而在SSD中，则是保证正、负样本的比例来实现样本均衡。具体做法是在计算出所有负样本的损失后进行排序，选取损失较大的那一部分进行计算，舍弃剩下的负样本，数量为正样本的3倍。

具体实现如代码所示。在计算完所有边框的类别交叉熵损失后，难样本挖掘过程主要分为5步：首先过滤掉正样本；然后将负样本的损失排序；接着计算正样本的数量；进而得到负样本的数量；最后根据损失大小得到留下的负样本索引。源代码文件见  
layers/modules/multibox\_loss.py。

---

```
# 对于类别损失，进行难样本挖掘，控制比例为1:3
# 所有PriorBox的类别预测量
batch_conf = conf_data.view(-1, self.num_classes)
# 利用交叉熵函数，计算所有PriorBox的类别损失
loss_c = log_sum_exp(batch_conf) - batch_conf.gather(1, conf_t.view(-1, 1))
# 接下来进行难样本挖掘，分为5步
loss_c = loss_c.view(pos.size()[0], pos.size()[1])
# 1: 首先过滤掉正样本
loss_c[pos] = 0 # filter out pos boxes for now
loss_c = loss_c.view(num, -1)
# 2: 将所有负样本的类别损失排序
_, loss_idx = loss_c.sort(1, descending=True)
# idx_rank为排序后每个PriorBox的排名
_, idx_rank = loss_idx.sort(1)
# 3: 计算正样本的数量
num_pos = pos.long().sum(1, keepdim=True)
```

```
# 4: 控制正、负样本的比例为1:3
num_neg = torch.clamp(self.negpos_ratio*num_pos, max=pos.size(1)-1)
# 5: 选择每个batch中负样本的索引
neg = idx_rank < num_neg.expand_as(idx_rank)
```

---

#### 5.4.4 类别损失计算

在得到筛选后的正、负样本后，即可进行类别的损失计算。SSD在此使用了交叉熵损失函数，并且正、负样本全部参与计算。交叉熵损失函数可参见4.4.5节。

---

```
# 计算正、负样本的类别损失
# 将正、负样本的索引扩展为[32, 8732, 21]格式
pos_idx = pos.unsqueeze(2).expand_as(conf_data)
neg_idx = neg.unsqueeze(2).expand_as(conf_data)
# 把类别的预测值从所有的预测中提取出来
conf_p = conf_data[(pos_idx+neg_idx).gt(0)].view(-1, self.num_classes)
# 把类别的真值从所有真值中提取出来
targets_weighted = conf_t[(pos+neg).gt(0)]
loss_c = F.cross_entropy(conf_p, targets_weighted, size_average=False)
```

---

## 5.5 SSD的改进算法

本章的前几节从代码角度详细讲解了SSD算法的整体流程。本节将对SSD算法进行总结，并介绍多个基于SSD的改进算法。

### 5.5.1 审视SSD

SSD实现了一个较为优雅、简洁的物体检测框架，使用了一阶网络即完成了物体检测任务，达到了同时期物体检测的较高水平。总体上，SSD主要有以下3个优点：

- 由于利用了多层的特征图进行预测，因此虽然是一阶的网络，但在某些场景与数据集下，检测精度依然可以与Faster RCNN媲美。
- 一阶网络的实现，使得其检测速度可以超过同时期的Faster RCNN及YOLO算法，对于速度要求较高的工程应用场景，SSD是一个很不错的选择。
- 网络优雅，实现简单，没有太多的工程技巧，这也为后续的改善工作提供了很大的空间。

与此同时，追求更高检测性能的脚步永不会停止，SSD算法也有以下3点限制：

- 对于小物体的检测效果一般，这是由于其虽然使用了分辨率大的浅层特征图来检测小物体，但浅层的语义信息不足，无法很好地完成分类与回归的预测。
- 每一层PriorBox的大小与宽高依赖于人工设置，无法自动学习，当检测任务更换时，调试过程较为烦琐。
- 由于是一阶的检测算法，分类与边框回归都只有一次，在一些追求高精度的场景下，SSD系列相较于Faster RNCN系列来讲，仍然处在下风。

针对SSD的这些问题，后续的学者从多个角度探讨了提升SSD性能的策略，在此介绍4个较为经典的改进算法，分别是DSSD、RSSD、RefineDet及RFBNet算法，如图5.9所示。

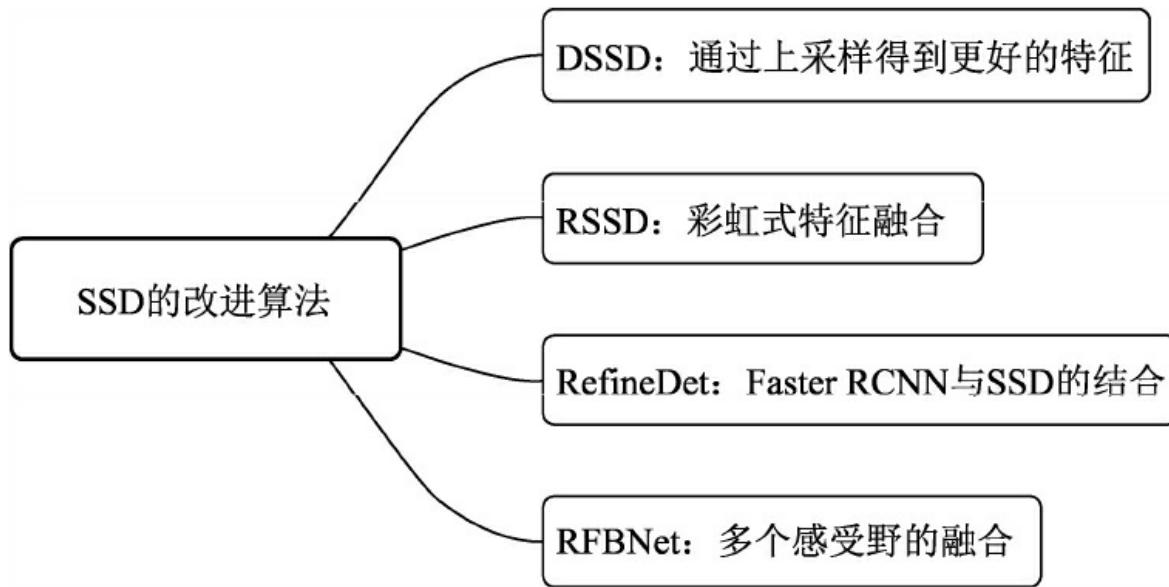


图5.9 SSD的多种改进算法

## 5.5.2 特征融合：DSSD

SSD采用多尺度的特征图预测物体，具有较大感受野的特征图检测大尺度物体，较小感受野特征图检测小尺度物体。这样会存在一个问题，即当小感受野特征图检测小物体时，由于语义信息不足，导致小物体检测效果差。

解决该语义信息不足的方法通常是深浅的特征图融合，DSSD正是从这一角度出发，提出了一套针对SSD多尺度预测的特征融合方法，改进了传统的上采样方法，并且可以适用于多种基础Backbone结构。

对于深浅层的特征融合，通常有3种计算方法：

- 按通道拼接（Concatenation）：将深层与浅层的特征按通道维度进行拼接。

- 逐元素相加（Eltw Sum）：将深层与浅层的每一个元素在对应位置进行相加，如FPN中的特征融合。

- 逐元素相乘（Eltw Product）：将深层与浅层的每一个元素在对应位置进行相乘，这也是DSSD采用的特征融合方式。

SSD利用了感受野与分辨率不同的6个特征图进行后续分类与回归网络的计算，DSSD保留了这6个特征图，但对这6个特征图进一步进行了融合处理，然后将融合后的结果送入后续分类与回归网络，如图5.10所示。

具体做法是，将最深层的特征图直接用作分类与回归，接着，该特征经过一个反卷积模块，并与更浅一层的特征进行逐元素相乘，将输出的特征用于分类与回归计算。类似地，继续将该特征与浅层特征进行反

卷积与融合，共计输出6个融合后的特征图，形成一个沙漏式的结构，最后给分类与回归网络做预测。

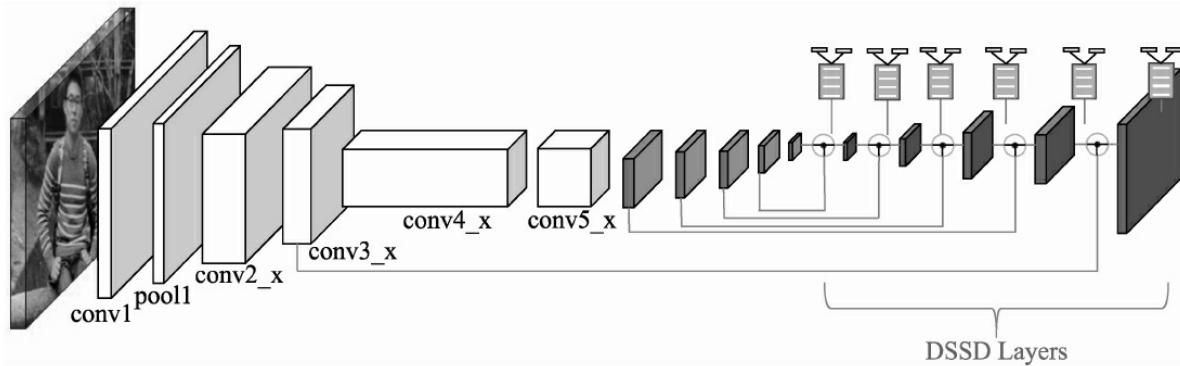


图5.10 DSSD算法的网络结构

具体的反卷积模块如图5.11所示。这里深特征图的大小是 $H \times W \times 512$ ，浅特征图的大小为 $2H \times 2W \times D$ 。深特征图经过反卷积后尺寸与浅特征图相同，再经过一些卷积、ReLU与BN操作后，两者进行逐元素的相乘，最后经过一个ReLU模块，得到最终需要的特征图。

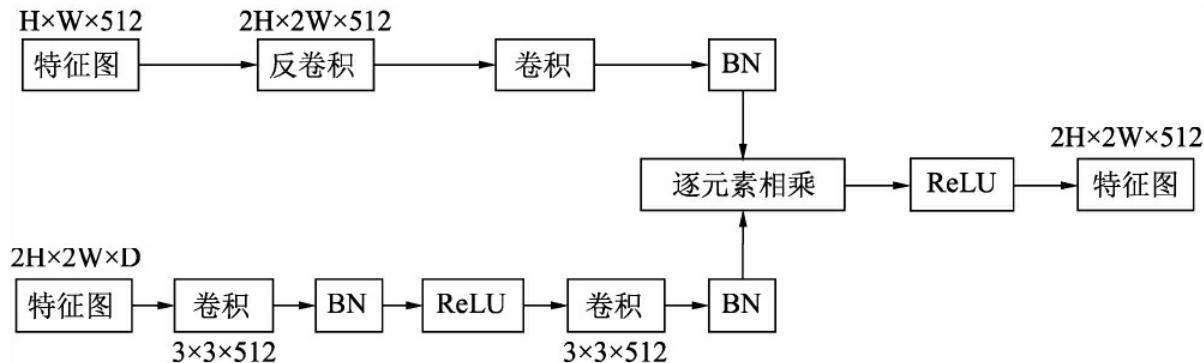


图5.11 DSSD采用的反卷积模块

这里有以下3点需要说明：

- BN层在卷积与激活函数之间，由于不同层之间的特征尺度与感受野不同，直接融合在一起训练难度较大，利用BN层可以进行均一化处理，加速训练。

·在实验时，DSSD采用逐元素相乘比逐元素相加提升了0.2%个mAP，速度上相乘操作比相加稍慢一点。

·对于上采样的实现，之前的一些方法一般利用双线性插值操作，而DSSD这里使用了反卷积，即包含了可学习的参数。

在得到特征图后，DSSD也改进了分类与回归的预测模块。SSD的预测模块是直接使用 $3\times 3$ 卷积，而DSSD则对比了多种方法，最终选择了如图5.12所示的计算方式，包含了一个残差单元，主路和旁路进行逐元素相加，然后再接到分类与回归的预测模块中。

DSSD的算法将深层的特征融合到了浅层特征图中，提升了浅层特征的语义性，也因此提升了模型的性能，尤其是对于小物体的检测。

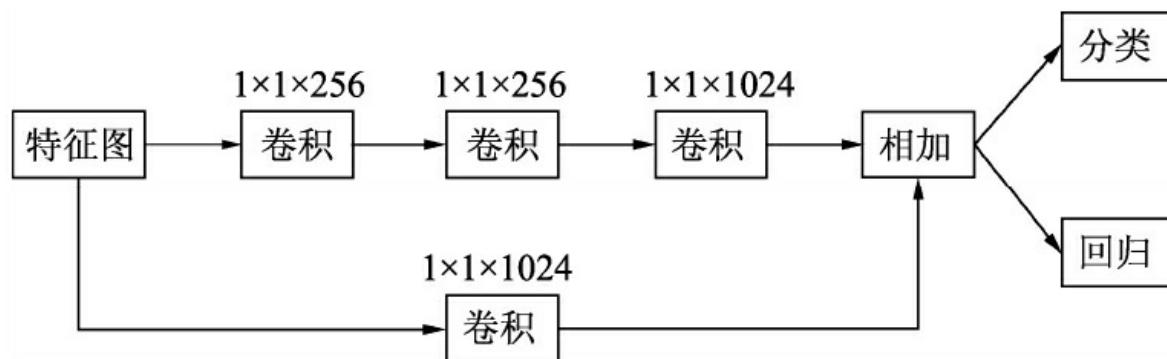


图5.12 DSSD的预测网络结构

### 5.5.3 彩虹网络：RSSD

原始的SSD算法使用了多层大小与感受野不同的特征图进行检测，因此对于物体的尺度变化有了一定的鲁棒性，但仍存在以下两个问题：

- 各个特征图只考虑当前层的检测尺度，没有考虑特征图之间的关联性，容易出现多个特征图上的预选框与一个真实框（GroundTruth）相匹配，即使有NMS后续操作，但也不能完全避免这种误检的情况。

- 小物体检测效果差：SSD主要利用浅层特征图检测小物体，但由于浅层的特征图语义信息太少，影响了对小物体的检测。

针对上述问题，RSSD（Rainbow SSD，R-SSD）算法一方面利用分类网络增加了不同特征图之间的联系，减少了重复框的出现，另一方面提出了一种全新的深浅特征融合的方法，增加了特征图的通道数，大幅度提升了检测的效果。

当前，深、浅层特征融合已经成为了一种流行的提升检测效果的方法。RSSD尝试了3种不同的融合方案。这里的融合只针对需要进行后续分类与回归的特征图，卷积网络中的其他特征图不参与融合。

#### 1. 池化融合

对浅层的特征图进行池化，然后与下一个特征图进行通道拼接（Concatenation），作为下一层特征图的最终特征。类似地，依次向下进行通道拼接，这样特征图的通道数逐渐增加，并且融合了越来越多层的特征，如图5.13所示。

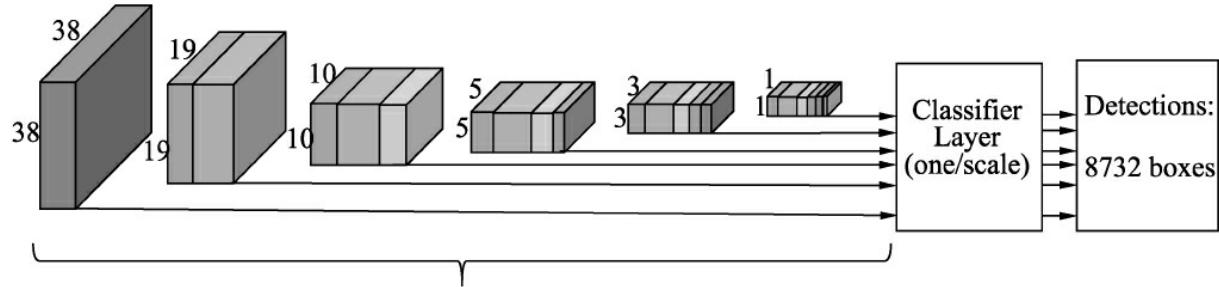


图5.13 池化融合方法示意图

这样的连接方式可以使得深层特征图既具有高语义信息，又拥有浅层特征图传来的细节信息，因此可以提升检测的效果。

## 2. 反卷积融合

与池化融合方法相反，反卷积融合是对深层的特征图进行反卷积，扩充尺寸，然后与浅层的特征图进行通道拼接，并逐渐传递到最浅层，如图5.14所示。

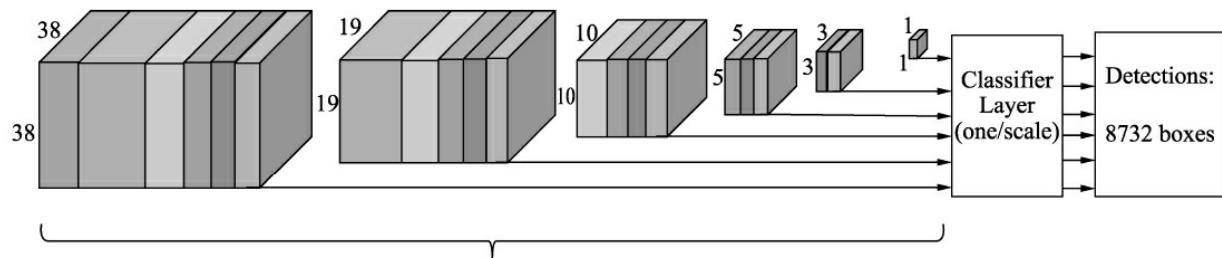


图5.14 反卷积融合方法示意图

这种方法类似于FPN结构，弥补了浅层的语义信息。

## 3. 彩虹（Rainbow）式融合

最后一种是彩虹式融合方法，结合了前两种策略，也是Rainbow SSD最终采用的方式。网络结构如图5.15所示，浅层特征通过池化层融合到深层中，深层特征通过反卷积融合到浅层中，这样就使每一个特征

图的通道数是相同的。由于每一层都融合了多个特征图的特征，并且可以用图5.15的颜色进行区分，因此被称为Rainbow SSD。

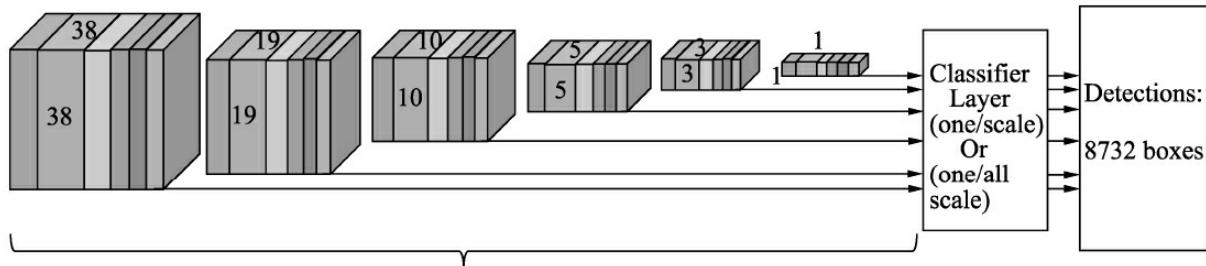


图5.15 彩虹融合方法示意图

值得注意的是，由于每一层的特征图尺度不同，并且感受野也不同，因此在不同层融合之前，需要做一次BN操作，以统一融合时的每个层的尺度，从而达到均一化的效果。

RSSD除了提出彩虹式的特征提取网络之外，还共享了分类网络分支的卷积权重。从图5.15中可以看出，每一个特征图的特征通道相同，因此可以采用同一个分类网络来提取预选框的类别预测信息，共用一套分类网络权重参数，这样做好处是将不同尺度的问题归一化处理，模型更加稳定，也更容易收敛。

总体来看，RSSD在不明显增加计算量的基础上，融合了双向的特征信息，并且考虑了各个分类分支的关联性，使得模型的表达能力更强，在一定程度上提升了小物体的检测与物体的定位精度。

## 5.5.4 基于SSD的两阶：RefineDet

SSD作为一阶网络，通过直接对固定预选框进行预测分类的方法，实现了简单有效的检测，但相比两阶网络，其检测精度较低。另外，两阶网络如Faster RCNN，检测精度高，同时也限制了其检测速度。

在上述前提下，2018年，CVPR的RefineDet结合了一阶网络与两阶网络的优点，在保持高效的前提下实现了精度更高的检测。

RefineDet在SSD的基础上，一方面引入了Faster RCNN两阶网络中边框由粗到细两步调整的思想，即先通过一个网络粗调固定框（在Faster RCNN中是Anchor，在SSD中是PriorBox），然后再通过一个网络细调；另一方面采用了类似于FPN的特征融合方法，可以有效提高对小物体的检测效果。

RefineDet的网络结构主要由ARM（Anchor Refinement Module）、TCB（Transfer Connection Block）与ODM（Object Detection Module）这3个模块组成，如图5.16所示。下面详细介绍这3部分的结构与作用。

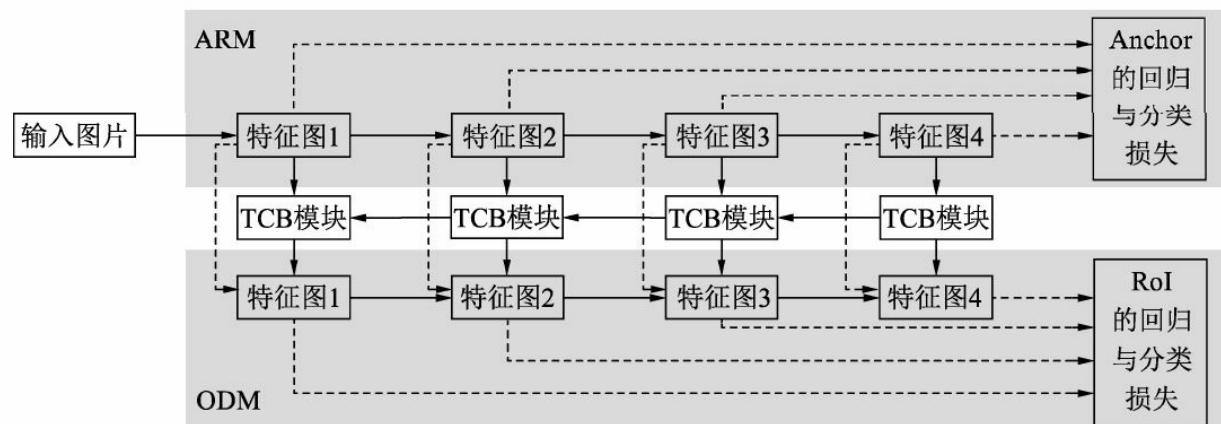


图5.16 RefineDet的网络结构

## 1. ARM部分

ARM部分首先经过一个VGG 16或者ResNet-101的基础网络，然后在多个特征图上对应不同大小宽高的Anchors，并用卷积网络来提取这些Anchors的特征，进一步可以求得每一个Anchor的分类与回归损失。这一步起到了类似于RPN的作用，网络结构有点类似于SSD。

具体网络以ResNet-101为例，假设输入图像大小为 $320 \times 320$ ，则图5.16中ARM部分的4个特征图大小分别为 $40 \times 40$ 、 $20 \times 20$ 、 $10 \times 10$ 及 $5 \times 5$ ，前3个是ResNet中的网络层，最后一个是由额外添加的一个模块。

总体来看，ARM有如下两点作用：

- 过滤掉一些简单的负样本，减少后续模型的搜索空间，也缓解了正、负样本不均衡的问题。
- 通过类似于RPN一样的网络，可以得到Anchor的预测值，因此可以粗略地修正Anchor的位置，为下一个模块提供比固定框更精准的兴趣区域。相比一阶网络，这样做往往可以得到更高的精度。

下面以VGNet为例，使用PyTorch实现ARM模块的搭建。

---

```
def arm_multibox(vgg, extra_layers, cfg):
    arm_loc_layers = []
    arm_conf_layers = []
    vgg_source = [21, 28, -2] # VGGNet 中的层索引
    for k, v in enumerate(vgg_source):
        arm_loc_layers += [nn.Conv2d(vgg[v].out_channels,
                                    cfg[k]*4, kernel_size=3, padding=1)]
        arm_conf_layers += [nn.Conv2d(vgg[v].out_channels,
                                    cfg[k]*2, kernel_size=3, padding=1)]
    # 将深层的几个特征图也添加到分类与回归预测中
    for k, v in enumerate(extra_layers[1::2], 3):
        arm_loc_layers += [nn.Conv2d(v.out_channels, cfg[k]
                                    *4, kernel_size=3, padding=1)]
        arm_conf_layers += [nn.Conv2d(v.out_channels, cfg[k])]
```

```
        *2, kernel_size=3, padding=1)]  
    return (arm_loc_layers, arm_conf_layers)
```

---

## 2. TCB部分

TCB模块主要用于完成特征的转换操作。从图5.16中可以看出，TCB模块首先对ARM中的每一个特征图进行转换，然后将深层的特征图融合到浅层的特征图中，这一步非常像FPN中自上而下与横向连接操作的操作。

TCB模块的具体结构如图5.17所示，使用了两个 $3 \times 3$ 大小的卷积进一步处理ARM部分的特征图，与此同时，利用反卷积处理深层的特征图，得到与浅层尺寸相同的特征，然后逐元素相加，最后再经过一个 $3 \times 3$ 卷积，将结果输出到ODM模块中。

从网络结构来看，TCB模块有些类似于FPN，这种深浅融合的网络可以使特征图拥有更加丰富的信息，对于小物体等物体预测会更加准确。

下面使用PyTorch来搭建一个TCB模块。

---

```
def add_tcb(cfg):  
    feature_scale_layers = []  
    feature_upsample_layers = []  
    feature_pred_layers = []  
    # 这里的cfg为[512, 512, 1024, 512]，即ARM模块传过来的每个特征图的通道数  
    for k, v in enumerate(cfg):  
        # 构建图5.15中上部的卷积层  
        feature_scale_layers += [nn.Conv2d(cfg[k], 256, 3, padding=1),  
                                nn.ReLU(inplace=True),  
                                nn.Conv2d(256, 256, 3, padding=1)]  
        # 构建图5.15中下部的卷积层  
        feature_pred_layers += [nn.ReLU(inplace=True),  
                               nn.Conv2d(256, 256, 3, padding=1),  
                               nn.ReLU(inplace=True)]
```

```

    ]
    # 除了最后一个不需要上采样，其余的都需要进行上采样处理
    if k != len(cfg) - 1:
        feature_upsample_layers += [nn.ConvTranspose2d(256, 256, 4, 2)]
    return (feature_scale_layers, feature_upsample_layers, feature_pred_layers)

```

---

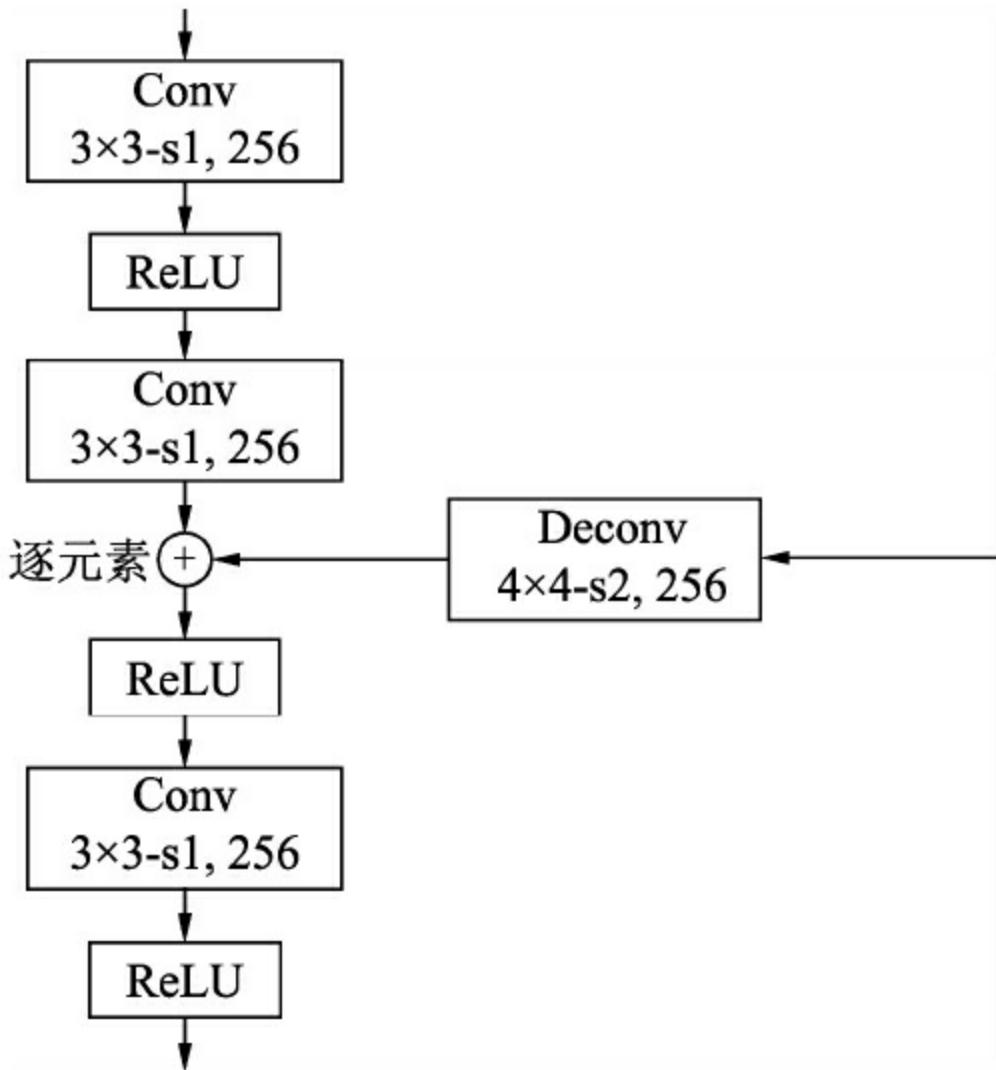


图5.17 TCB模块网络结构

### 3. ODM部分

ODM是RefineDet的最后一个模块，基本采取了SSD多层特征图的网络结构，但是相比SSD算法，ODM模块的输入有下面两个优势。

·更好的特征图：相比SSD从浅到深的特征，ODM接收的是TCB模块深浅融合的特征，其信息更加丰富，质量更高。

·更好的预选框：SSD使用了固定的预选框PriorBox，而ODM接收的则是经过ARM模块优化后的Anchor，位置更加精准，并且正、负样本更加均衡。

ODM后续的匹配与损失计算也基本沿用了SSD的算法，采用0.5作为正、负样本的阈值，正、负样本的比例为1：3。

RefineDet的损失包含了ARM与ODM两部分。ARM模块包含了交叉熵分类与smoothL1损失函数；ODM部分则是Softmax损失与smooth<sub>L1</sub>损失两部分。此外，ARM与ODM损失在训练时可以一起反向传播，因此可以实现端到端训练。

在具体的代码实现时，为了计算方便，在ARM处并没有直接抑制掉得分很低的Anchor，而是在随后的ODM中将两部分的得分综合考虑，完成Anchor的分类。这种操作也保证了全网络只有一次预选框筛选，从这个角度来看，RefineDet是一阶的。

RefineDet借鉴了两阶网络的优点，对预选框优化（Refine）了两次，这也是其名称RefineDet的由来。由于结合了Faster RCNN、SSD及FPN等算法的优势，在RefineDet算法诞生时，其在VOC 2007数据集上实现了单模型的最佳效果。

## 5.5.5 多感受野融合：RFBNet

在物体检测领域中，精度与速度始终是难以兼得的两个指标。对于两阶网络，如Faster RCNN，第一阶RPN网络用于提取没有类别信息的兴趣区域，然后利用第二阶对该区域进行分类与细回归，这样虽然获得了较高的精度，但牺牲了速度。对于性能强悍的主干网络，如ResNet-101的残差结构、FPN自上到下的特征融合，也都是在牺牲了前向速度的前提下，实现了较高的精度。

一阶网络如SSD、YOLO等利用一次预测就完成了物体的分类与回归任务，基本可以达到实时的速度，但精度始终不如同时期的两阶网络。基于一阶网络的改进算法，如DSSD、RefineDet等虽然提升了一阶网络的精度，但仍牺牲了网络的计算速度。

针对以上问题，RFBNet（Receptive Field Block Net）受人类视觉感知的启发，在原有的检测框架中增加了一些精心设计的模块，使得网络拥有更为强悍的表征能力，而不是简单地增加网络层数、融合等，因此获得了较好的检测速度与精度。

神经学发现，群体感受野的尺度会随着视网膜图的偏心而变化，偏心度越大的地方，感受野的尺度也越大。而在卷积网络中，卷积核的不同大小可以实现感受野的不同尺度，空洞卷积的空洞数也可以实现不同的偏心度。

传统的卷积模块，如Inception结构，虽然利用了多个分支来实现多个尺度的感受野，但由于空洞数都为1，离心率相同，因此在融合时仅仅是不同尺度的堆叠。

相比之下，RFBNet同时将感受野的尺度与偏心度纳入到卷积模块

中，设计了RFB模块，如图5.18所示。与Inception结构类似，RFB模块拥有3个不同的分支，并且使用了 $1 \times 1$ 、 $3 \times 3$ 与 $5 \times 5$ 不同大小的卷积核来模拟不同的感受野尺度，使用了空洞数为1、3、5的空洞卷积来实现不同的偏心度。

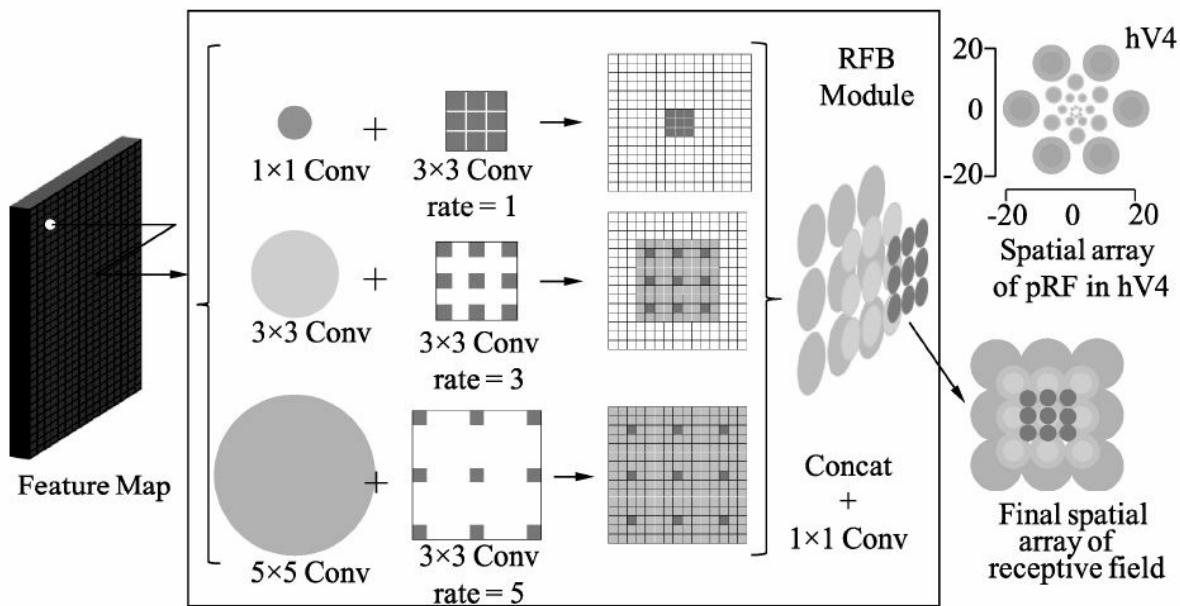


图5.18 RFBNet特征提取模块

完成3个分支后，再利用通道拼接的方法进行融合，利用 $1 \times 1$ 的卷积降低特征的通道数。可以看出，RFB模块的特征融合与人类的感知特点十分类似。

RFBNet将此RFB模块集成到了SSD中，在没有使用ResNet-101的前提下，可以达到两阶网络的检测精度。同时，RFBNet还尝试使用了更轻量化的网络，如MobileNet，检测效果仍然很好，证明了该算法具有较强的泛化能力。

总体来讲，RFBNet同时实现了较好的检测性能与较快的速度，并且实现简单，是一个非常优雅的物体检测算法。

## 5.6 总结

SSD利用多个特征图，只用了一次的预测，就实现了整个物体检测。与Faster RCNN相比，虽然其精度稍逊色一些，但在追求速度的场景，尤其是实时检测的要求中，SSD则更为适用。此外，SSD的单阶检测之所以如此有效，很大程度上得益于其充分的数据增强手段。当我们在实际应用时，如果缺乏数据或者容易发生过拟合现象，可以借鉴SSD的数据增强过程。

在下一章中，我们将学习另一个大名鼎鼎的一阶检测算法——YOLO，其速度更快，在工程中应用更为广泛。

## 第6章 单阶经典检测器：YOLO

第4章的Faster RCNN算法利用了两阶结构，先实现感兴趣区域的生成，再进行精细的分类与回归，虽出色地完成了物体检测任务，但也限制了其速度，在更追求速度的实际应用场景下，应用起来仍存在差距。

在此背景下，YOLO v1算法利用回归的思想，使用一阶网络直接完成了分类与位置定位两个任务，速度极快。随后出现的YOLO v2与v3在检测精度与速度上有了进一步的提升，加速了物体检测在工业界的應用，开辟了物体检测算法的另一片天地。

基于此背景，本章将分别介绍YOLO三个版本的思想与整体结构。由于YOLO v2的改进与提升十分显著，本章将从代码角度对此版本进行详细讲述。

## 6.1 无锚框预测：YOLO v1

相比起Faster RCNN的两阶结构，2015年诞生的YOLO v1创造性地使用一阶结构完成了物体检测任务，直接预测物体的类别与位置，没有RPN网络，也没有类似于Anchor的预选框，因此速度很快。本节将详细介绍YOLO v1的网络结构与算法思想。

### 6.1.1 网络结构

与其他物体检测算法一样，YOLO v1首先利用卷积神经网络进行了特征提取，具体结构如图6.1所示，该结构与GoogLeNet模型有些类似。在该结构中，输出图像的尺寸固定为 $448 \times 448$ ，经过24个卷积层与两个全连接层后，最后输出的特征图大小为 $7 \times 7 \times 30$ 。

关于YOLO v1的网络结构，有以下3个细节：

·在 $3 \times 3$ 的卷积后通常会接一个通道数更低的 $1 \times 1$ 卷积，这种方式既降低了计算量，同时也提升了模型的非线性能力。

·除了最后一层使用了线性激活函数外，其余层的激活函数为Leaky ReLU。

·在训练中使用了Dropout与数据增强的方法来防止过拟合。

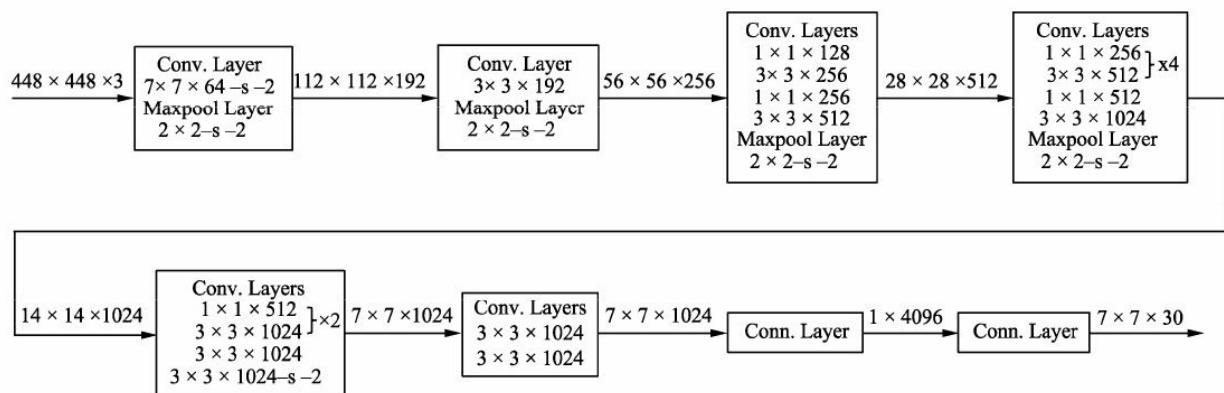


图6.1 YOLO v1网络结构

## 6.1.2 特征图的意义

YOLO v1的网络结构并无太多创新之处，其精髓主要在最后 $7 \times 7 \times 30$ 大小的特征图中。如图6.2所示，YOLO v1将输入图像划分成 $7 \times 7$ 的区域，每一个区域对应于最后特征图上的一个点，该点的通道数为30，代表了预测的30个特征。

YOLO v1在每一个区域内预测两个边框，如图6.2中的预测框A与B，这样整个图上一共预测 $7 \times 7 \times 2 = 98$ 个框，这些边框大小与位置各不相同，基本可以覆盖整个图上可能出现的物体。

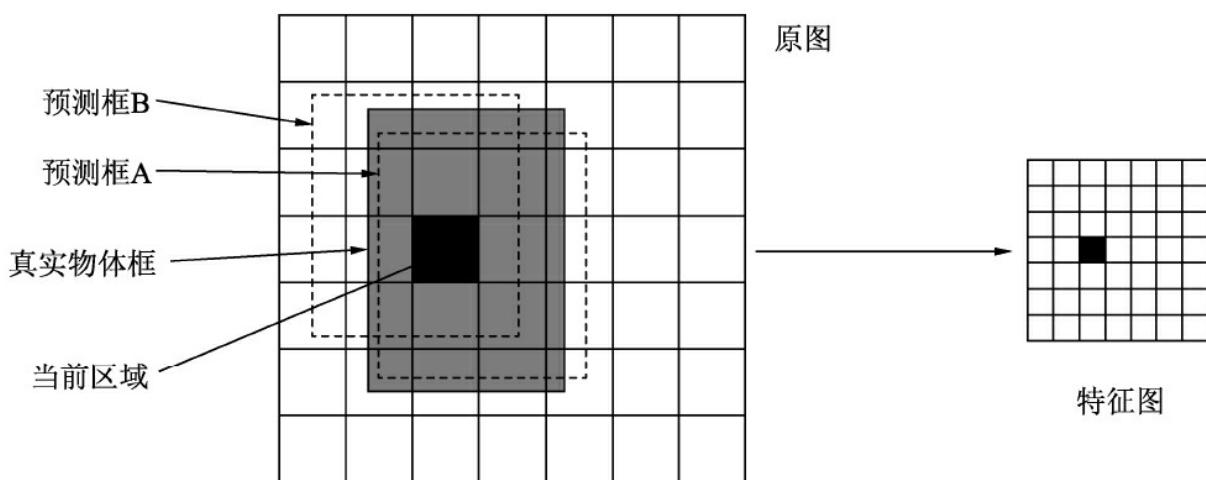


图6.2 YOLO v1检测原理图

如果一个物体的中心点落在了某个区域内，则该区域就负责检测该物体。图6.2中真实物体框的中心点在当前区域内，该区域就负责检测该物体，具体是将该区域的两个框与真实物体框进行匹配，IoU更大的框负责回归该真实物体框，在此A框更接近真实物体。

最终的预测特征由类别概率、边框的置信度及边框的位置组成，如图6.3所示。这三者的含义如下：

·类别概率: 由于PASCAL VOC数据集一共有20个物体类别，因此这里预测的是边框属于哪一个类别。

·置信度: 表示该区域内是否包含物体的概率，类似于Faster RCNN中是前景还是背景。由于有两个边框，因此会存在两个置信度预测值。

·边框位置: 对每一个边框需要预测其中心坐标及宽、高这4个量，两个边框共计8个预测值。

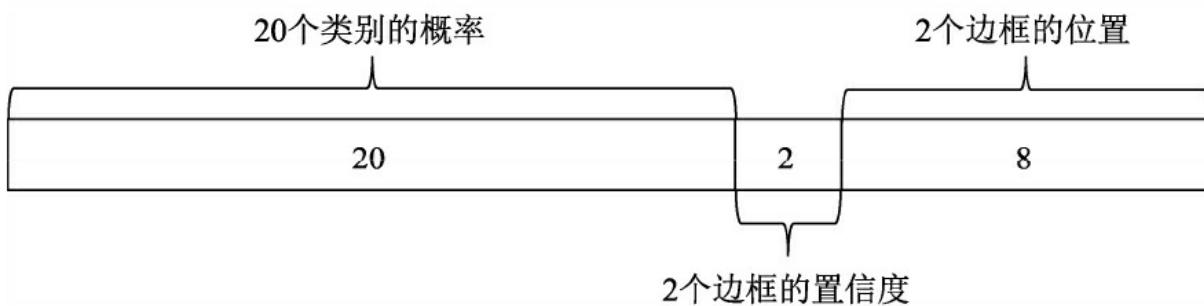


图6.3 预测特征的30维通道含义

这里有以下3点值得注意的细节：

·YOLO v1并没有先验框，而是直接在每个区域预测框的大小与位置，是一个回归问题。这样做能够成功检测的原因在于，区域本身就包含了一定的位置信息，另外被检测物体的尺度在一个可以回归的范围内。

·从图6.3中可以看出，一个区域内的两个边框共用一个类别预测，在训练时会选取与物体IoU更大的一个边框，在测试时会选取置信度更高的一个边框，另一个会被舍弃，因此整张图最多检测出49个物体。

·YOLO v1采用了物体类别与置信度分开的预测方法，这点与Faster RCNN不同。Faster RCNN将背景也当做了-一个类别，共计21种，在类别预测中包含了置信度的预测。

### 6.1.3 损失计算

通过卷积网络得到每个边框的预测值后，为了进一步计算网络训练的损失，还需要确定每一个边框是对应着真实物体还是背景框，即区分开正、负样本。YOLO v1在确定正负样本时，有以下两个原则：

- 当一个真实物体的中心点落在了某个区域内时，该区域就负责检测该物体。具体做法是将与该真实物体有最大IoU的边框设为正样本，这个区域的类别真值为该真实物体的类别，该边框的置信度真值为1。
- 除了上述被赋予正样本的边框，其余边框都为负样本。负样本没有类别损失与边框位置损失，只有置信度损失，其真值为0。

YOLO v1的损失一共由5部分组成，均使用了均方差损失，如式(6-1)所示。

$$\begin{aligned} Loss = & \lambda_{coord} \sum_{i=0}^{s^2} \sum_{j=0}^B 1_{ij}^{obj} \left( x_i - \hat{x}_i \right)^2 + \left( y_i - \hat{y}_i \right)^2 \\ & + \lambda_{coord} \sum_{i=0}^{s^2} \sum_{j=0}^B 1_{ij}^{obj} \left( \sqrt{\omega_i} - \sqrt{\hat{\omega}_i} \right)^2 + \left( \sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \\ & + \sum_{i=0}^{s^2} \sum_{j=0}^B 1_{ij}^{obj} \left( C_i - \hat{C}_i \right)^2 + \lambda_{noobj} \sum_{i=0}^{s^2} \sum_{j=0}^B 1_{ij}^{noobj} \left( C_i - \hat{C}_i \right)^2 \\ & + \sum_{i=0}^{s^2} 1_{ij}^{obj} \sum_{c \in classes} \left( p_i(c) - \hat{p}_i(c) \right)^2 \end{aligned} \quad (6-1)$$

公式中*i*代表第几个区域，一共有S<sup>2</sup>个区域，在此为49； *j*代表某个区域的第几个预测边框，一共有B个预测框，在此为2； *obj*代表该框对应了真实物体； *noobj*代表该框没有对应真实物体。这5项损失的意义如

下：

·第一项为正样本中心点坐标的损失。  $\lambda_{\text{coord}}$ 的目的是为了调节位置损失的权重，YOLO v1设置 $\lambda_{\text{coord}}$ 为5，调高了位置损失的权重。

·第二项为正样本宽高的损失。由于宽高差值受物体尺度的影响，因此这里先对宽高进行了平方根处理，在一定程度上降低对尺度的敏感，强化了小物体的损失权重。

·第三、四项分别为正样本与负样本的置信度损失，正样本置信度真值为1，负样本置信度为0。 $\lambda_{\text{noobj}}$ 默认为0.5，目的是调低负样本置信度损失的权重。

·最后一项为正样本的类别损失。

总体上，YOLO v1利用了回归的思想，使用轻量化的一阶网络同时完成了物体的定位与分类，处理速度极快，可以达到45 FPS，当使用更轻量的网络时甚至可以达到155 FPS。得益于其出色的处理速度，YOLO v1被广泛应用在实际的工业场景中，尤其是追求实时处理的场景。当然，YOLO v1也有一些不足之处，主要有如下3点：

·由于每一个区域默认只有两个边框做预测，并且只有一个类别，因此YOLO v1有着天然的检测限制。这种限制会导致模型对于小物体，以及靠得特别近的物体检测效果不好。

·由于没有类似于Anchor的先验框，模型对于新的或者不常见宽高比例的物体检测效果不好。另外，由于下采样率较大，边框的检测精度不高。

·在损失函数中，大物体的位置损失权重与小物体的位置损失权重是一样的，这会导致同等比例的位置误差，大物体的损失会比小物体

大，小物体的损失在总损失中占比较小，会带来物体定位的不准确。

## 6.2 依赖锚框：YOLO v2

针对YOLO v1的不足，2016年诞生了YOLO v2。相比起第一个版本，YOLO v2预测更加精准（Better）、速度更快（Faster）、识别的物体类别也更多（Stronger），在VOC 2007数据集上可以得到mAP 10%以上的提升效果。

YOLO v2从很多方面对YOLO做出了改进，大体可以分为网络结构的改善、先验框的设计及训练技巧3个方面，下面主要就这三部分进行讲解。本节中使用的YOLO v2源码来自于longcw的高质量实现，笔者将其迁移到了本书的代码中。

## 6.2.1 网络结构的改善

首先，YOLO v2对于基础网络结构进行了多种优化，提出了一个全新的网络结构，称之为DarkNet。原始的DarkNet拥有19个卷积层与5个池化层，在增加了一个Passthrough层后一共拥有22个卷积层，精度与VGGNet相当，但浮点运算量只有VGGNet的1/5左右，因此速度极快，具体结构如图6.4所示。

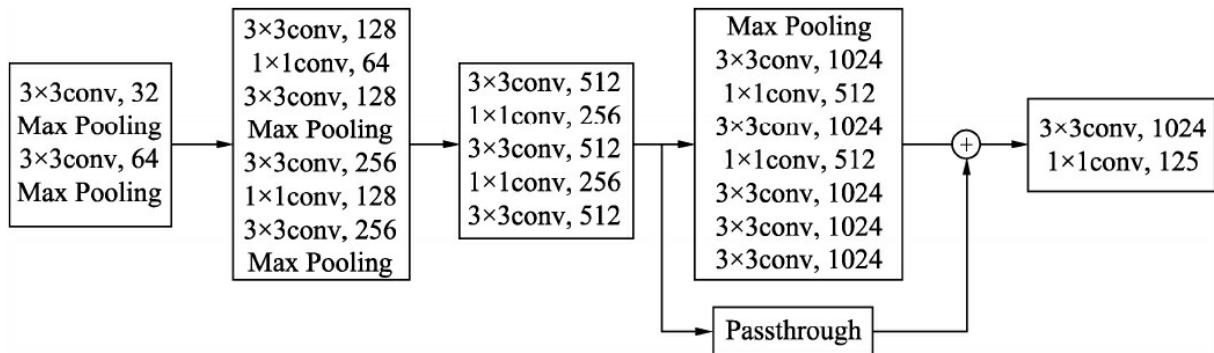


图6.4 DarkNet网络结构图

相比起v1版本的基础网络，DarkNet进行了以下几点改进：

·BN层：DarkNet使用了BN层，这一点带来了2%以上的性能提升。BN层有助于解决反向传播中的梯度消失与爆炸问题，可以加速模型的收敛，同时起到一定的正则化作用。BN层的具体位置是在每一个卷积之后，激活函数LeakyReLU之前。

·用连续 $3 \times 3$ 卷积替代了v1版本中的 $7 \times 7$ 卷积，这样既减少了计算量，又增加了网络深度。此外，DarkNet去掉了全连接层与Dropout层。

·Passthrough层：DarkNet还进行了深浅层特征的融合，具体方法是将浅层 $26 \times 26 \times 512$ 的特征变换为 $13 \times 13 \times 2048$ ，这样就可以直接与深层

13×13×1024的特征进行通道拼接。这种特征融合有利于小物体的检测，也为模型带来了1%的性能提升。

·由于YOLO v2在每一个区域预测5个边框，每个边框有25个预测值，因此最后输出的特征图通道数为125。其中，一个边框的25个预测值分别是20个类别预测、4个位置预测及1个置信度预测值。这里与v1有很大区别，v1是一个区域内的边框共享类别预测，而这里则是相互独立的类别预测值。

下面从代码角度介绍如何搭建DarkNet网络，源代码文件见darknet.py。和第3章常见的基础结构一样，在实现时，可以定义一个`_make_layers`函数，通过输入不同的配置，可以快速搭建整个基础网络。

---

```
def _make_layers(in_channels, net_cfg):
    layers = []
    # 如果输入是一个list，继续调用_make_layers，拆解到每一个网络基本单元
    if len(net_cfg) > 0 and isinstance(net_cfg[0], list):
        for sub_cfg in net_cfg:
            layer, in_channels = _make_layers(in_channels, sub_cfg)
            layers.append(layer)
    else:
        for item in net_cfg:
            # 定义每一个Max Pooling层
            if item == 'M':
                layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
            # 定义每一个卷积层，这里定义的卷积层已经包含了BN与ReLU层
            else:
                out_channels, ksize = item
                layers.append(net_utils.Conv2d_BatchNorm(in_channels,
                                                        out_channels,
                                                        ksize,
                                                        same_padding=True))
                in_channels = out_channels
    return nn.Sequential(*layers), in_channels
```

---

为了完成整个网络的搭建、前向与自动反向传播，在此定义了一个

DarkNet19类，继承自nn.Module，通过网络配置信息快速构建出每一个模块。

---

```
class Darknet19(nn.Module):
    def __init__(self):
        super(Darknet19, self).__init__()
        # 这里的cfgs与图6.4中的网络结构是一致的
        net_cfgs = [
            [(32, 3)],
            ['M', (64, 3)],
            ['M', (128, 3), (64, 1), (128, 3)],
            ['M', (256, 3), (128, 1), (256, 3)],
            ['M', (512, 3), (256, 1), (512, 3), (256, 1), (512, 3)],
            ['M', (1024, 3), (512, 1), (1024, 3), (512, 1), (1024, 3)],
            [(1024, 3), (1024, 3)],
            [(1024, 3)]
        ]
        # 构建前几个卷积模块
        self.conv1s, c1 = _make_layers(3, net_cfgs[0:5])
        self.conv2, c2 = _make_layers(c1, net_cfgs[5])
        self.conv3, c3 = _make_layers(c2, net_cfgs[6])
        # 为了实现Passthrough层，定义了一个ReorgLayer类
        stride = 2
        self.reorg = ReorgLayer(stride=2)
        # 这里conv4的输入通道数是conv1s的4倍再加conv3的和
        self.conv4, c4 = _make_layers((c1*(stride*stride) + c3), net_cfgs[7])
        # 最终输出通道数为预选框数量×25，即5×25=125
        out_channels = cfg.num_anchors * (cfg.num_classes + 5)
        self.conv5 = net_utils.Conv2d(c4, out_channels, 1, 1, relu=False)
        self.global_average_pool = nn.AvgPool2d((1, 1))
```

---

## 6.2.2 先验框的设计

YOLO v2吸收了Faster RCNN的优点，设置了一定量的预选框，使得模型不需要直接预测物体尺度与坐标，只需要预测先验框到真实物体的偏移，降低了预测难度。

关于先验框，YOLO v2首先使用了聚类的算法来确定先验框的尺度，并且优化了后续的偏移计算方法，下面详细介绍这两部分。

先验框的设计为YOLO v2带来了7%的召回率提升。

### 1. 聚类提取先验框尺度

Faster RCNN中预选框（即Anchor）的大小与宽高是由人手工设计的，因此很难确定设计出的一组预选框是最贴合数据集的，也就有可能为模型性能带来负面影响。

针对此问题，YOLO v2通过在训练集上聚类来获得预选框，只需要设定预选框的数量k，就可以利用聚类算法得到最适合的k个框。在聚类时，两个边框之间的距离使用式（6-2）的计算方法，即IoU越大，边框距离越近。

$$d(box, centroid) = 1 - IoU(box, centroid) \quad (6-2)$$

在衡量一组预选框的好坏时，使用真实物体与这一组预选框的平均IoU作为标准。值得一提的是，这一判断标准在手工设计预选框的方法中也可以使用。

至于预选框的数量选取，显然数量k越多，平均IoU会越大，效果会

更好，但相应的也会带来计算量的提升，YOLO v2在速度与精度的权衡中选择了预选框数量为5。

## 2. 优化偏移公式

有了先验框后，YOLO v2不再直接预测边框的位置坐标，而是预测先验框与真实物体的偏移量。在Faster RCNN中，中心坐标的偏移公式如式（6-3）所示。

$$\begin{cases} x = (t_x \times w_a) + x_a \\ y = (t_y \times h_a) + y_a \end{cases} \quad (6-3)$$

公式中 $w_a$ 、 $h_a$ 、 $x_a$ 及 $y_a$ 代表Anchor的宽高与中心坐标， $t_x$ 与 $t_y$ 是模型预测的Anchor相对于真实物体的偏移量，经过计算后得到预测的物体中心坐标x和y。

YOLO v2认为这种预测方式没有对预测偏移进行限制，导致预测的边框中心可以出现在图像的任何位置，尤其是在训练初始阶段，模型参数还相对不稳定。例如 $t_x$ 是1与-1时，预测的物体中心点会有两个宽度的差距。

因此，YOLO v2提出了式（6-4）所示的预测公式：

$$\begin{cases} b_x = \sigma(t_x) + c_x \\ b_y = \sigma(t_y) + c_y \\ b_w = p_w e^{t_w} \\ b_h = p_h e^{t_h} \\ p_r(\text{object}) \times IoU(b, \text{object}) = \sigma(t_0) \end{cases} \quad (6-4)$$

公式中参数的意义可以与图6.5结合进行理解，图中实线框代表预测框，虚线框代表先验框：

· $c_x$ 与 $c_y$ 代表中心点所处区域左上角的坐标， $p_w$ 与 $p_h$ 代表了当前先验框的宽高，如图6.5中的虚线框所示。

· $\sigma(t_x)$ 与 $\sigma(t_y)$ 代表预测框中心点与中心点所处区域左上角坐标的距离，加上 $c_x$ 与 $c_y$ 即得到预测框的中心坐标。

· $t_w$ 与 $t_h$ 为预测的宽高偏移量。先验框的宽高乘上指数化后的宽高偏移量，即得到预测框的宽高。

·公式中的 $\sigma$ 代表Sigmoid函数，作用是将坐标偏移量化到(0,1)区间，这样得到的预测边框的中心坐标 $b_x$ 、 $b_y$ 会限制在当前区域内，保证一个区域只预测中心点在该区域内的物体，有利于模型收敛。

·YOLO v1将预测值 $t_0$ 作为边框的置信度，而YOLO v2则是将做Sigmoid变换后的 $\sigma(t_0)$ 作为真正的置信度预测值。

下面从代码角度介绍一下DarkNet的前向过程，模型的输出包含每一个预选框的类别预测、位置预测及置信度预测，然后再进一步处理为可以方便计算损失的形式。

---

```
def forward(self, im_data, gt_boxes=None, gt_classes=None, dontcare=None,
           size_index=0):
    # 前几个卷积模块的前向过程
    conv1s = self.conv1s(im_data)
    conv2 = self.conv2(conv1s)
    conv3 = self.conv3(conv2)
    conv1s_reorg = self.reorg(conv1s)
    # 将主线与Passthrough层的特征进行Concatenation操作
    cat_1_3 = torch.cat([conv1s_reorg, conv3], 1)
    conv4 = self.conv4(cat_1_3)
    conv5 = self.conv5(conv4)  # batch_size, out_channels, h, w
```

```

# 从特征图中提取3类预测信息
bsize, _, h, w = global_average_pool.size()
global_average_pool_reshaped = \
    global_average_pool.permute(0, 2, 3, 1).contiguous().view(bsize,
        -1, cfg.num_anchors, cfg.num_classes + 5)
# 从特征图中提取位置预测、置信度预测及类别预测信息
# 对tx、ty、tw、th、t0进行后处理，分别得到σ(tx)、σ(ty)、ew、eh与σ(t0)
xy_pred = F.sigmoid(global_average_pool_reshaped[:, :, :, 0:2])
wh_pred = torch.exp(global_average_pool_reshaped[:, :, :, 2:4])
bbox_pred = torch.cat([xy_pred, wh_pred], 3)
iou_pred = F.sigmoid(global_average_pool_reshaped[:, :, :, 4:5])
# 对类别得分进行Softmax处理，可以得到类别概率
score_pred = global_average_pool_reshaped[:, :, :, 5:].contiguous()
prob_pred = F.softmax(score_pred.view(-1,
    score_pred.size()[-1])).view_as(score_pred) # noqa

```

---

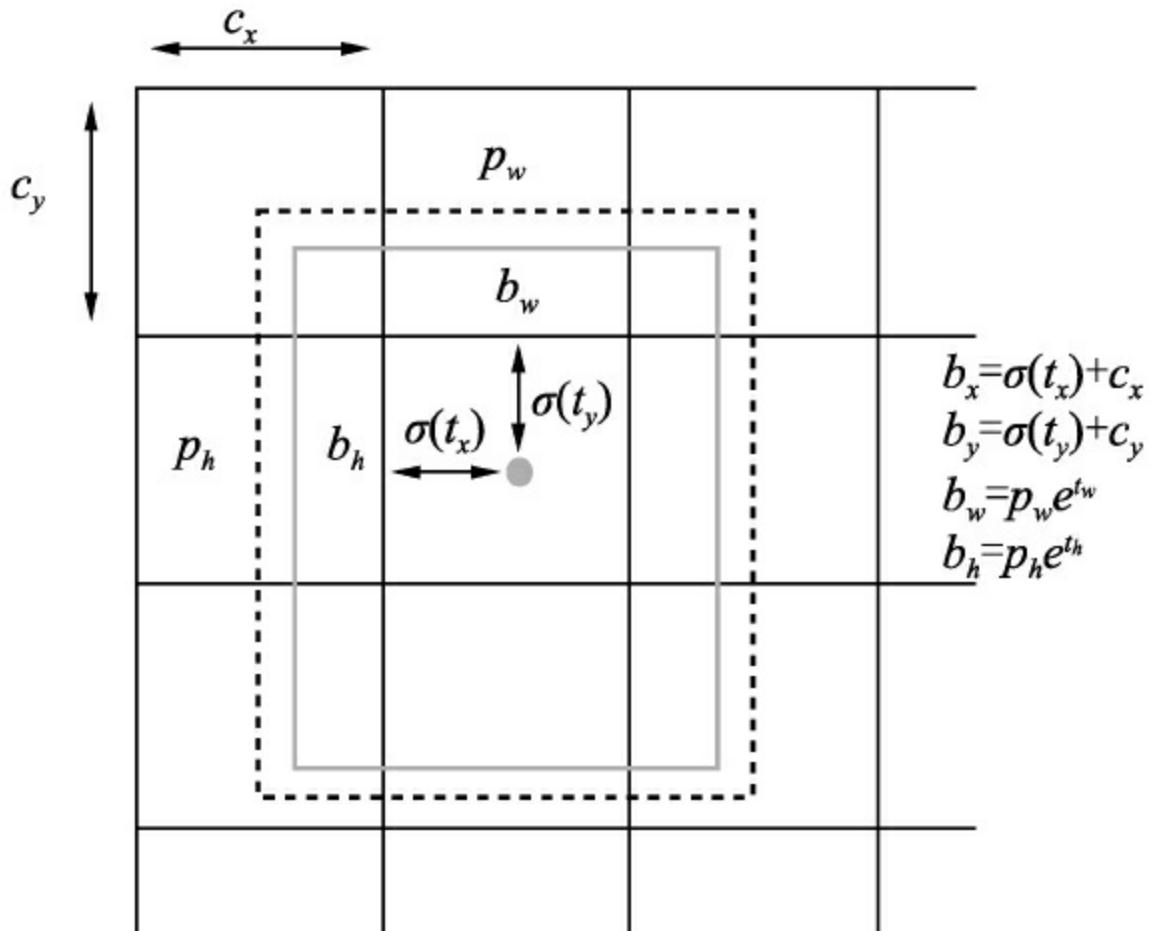


图6.5 YOLO v2预测值的含义

### 6.2.3 正、负样本与损失函数

关于正、负样本的选取，YOLO v2基本保持了之前的方法，其基本流程如下：

首先利用式（6-4），将预测的位置偏移量作用到先验框上，得到预测框的真实位置。

如果一个预测框与所有真实物体的最大IoU小于一定阈值（默认为0.6）时，该预测框视为负样本。

每一个真实物体的中心点落在了某个区域内，该区域就负责检测该物体。具体做法是将与该物体有最大IoU的预测框视为正样本。

确定了正样本与负样本后，最后是网络损失的计算。由于利用了先验框，YOLO v2的损失函数也相应的进行了改变，公式如式（6-5）所示。

$$\begin{aligned} Loss_t = & \sum_{i=0}^W \sum_{j=0}^H \sum_{k=0}^A (1_{\max IoU < Thresh} \times \lambda_{noobj} \times (-b_{ijk}^o)^2 \\ & + 1_{t<12800} \times \lambda_{prior} \times \sum_{r \in (x,y,w,h)} (prior_k^r - b_{ijk}^r)^2 \\ & + 1_k^{truth} \times \lambda_{coord} \times \sum_{r \in (x,y,w,h)} (truth^r - b_{ijk}^r)^2 \quad (6-5) \\ & + \lambda_{obj} \times (IoU_{truth}^k - b_{ijk}^o)^2 \\ & + \lambda_{class} \times \sum_{c=1}^C (truth^c - b_{ijk}^c)^2) \end{aligned}$$

损失一共有5项组成，意义分别如下：

·第一项为负样本的置信度损失，公式中 $1_{\max_{IoU < \text{Thresh}}}$ 表示最大IoU小于阈值，即负样本的边框， $\lambda_{noobi}$ 是负样本损失的权重， $b^o_{ijk}$ 为置信度 $\sigma(t_o)$ 。

·第二项为先验框与预测框的损失，只存在于前12800次迭代中，目的是使预测框先收敛于先验框，模型更稳定。

·第三项为正样本的位置损失，表示筛选出的正样本，为权重。

·后两项分别为正样本的置信度损失与类别损失，为置信度的真值。

在计算正、负样本的过程中，虽然有些预测框的最大IoU可能小于0.6，即被赋予了负样本，但如果后续是某一个真实物体对应的最大IoU的框时，该预测框会被最终赋予成正样本，以保证recall。

有些预测框的最大IoU大于0.6，但是在在一个区域内又不是与真实物体有最大IoU，这种预测框会被舍弃掉不参与损失计算，既不是正样本也不是负样本。

## 6.2.4 正、负样本选取代码示例

YOLO v2的正、负样本选取及损失计算方法还是较为复杂的，因此接下来从代码层讲解计算过程。

在卷积网络部分已经得出了损失计算的预测值，在此只需要计算真值及确定正、负样本。假设当前特征图尺寸为 $10 \times 10$ ，一个区域内有5个先验框，为了方便损失的计算，在此定义了3个真值量与3个标志量  
(mask)：

- boxes: 偏移量真值，大小为 $100 \times 5 \times 4$ ，其中，负样本的值为0，正样本为真实物体相对于先验框的偏移量，即 $b_x - c_x$ 、 $b_y - c_y$ 、 $b_w/p_w$ 及 $b_h/p_h$ 。
- box mask: 大小为 $100 \times 5 \times 1$ ，标志量，正样本为1，负样本为0。
- classes: 类别真值，大小为 $100 \times 5 \times 20$ ，正样本对应真实物体的类别为1，其余全为0。
- class mask: 类别标志量，大小为 $100 \times 5 \times 1$ ，正样本为1，负样本为0。
- ious: 置信度真值，大小为 $100 \times 5 \times 1$ ，负样本为0，正样本为真实的IoU值。
- iou mask: 置信度标志量，大小为 $100 \times 5 \times 1$ ，负样本的值是其最大IoU的负数。需要注意的是，正样本的mask是 $5 \times (1 - IoU)$ ，可以看到，IoU越小，mask越大，这也为不同的预测框提供了不同大小的权重。

下面的代码详细展示了真值及标志量的计算过程。

---

```

# 分别定义上述6个变量
_classes = np.zeros([hw, num_anchors, cfg.num_classes], dtype=np.float)
_class_mask = np.zeros([hw, num_anchors, 1], dtype=np.float)
_iou = np.zeros([hw, num_anchors, 1], dtype=np.float)
_iou_mask = np.zeros([hw, num_anchors, 1], dtype=np.float)
_boxes = np.zeros([hw, num_anchors, 4], dtype=np.float)
_boxes[:, :, 0:2] = 0.5
_boxes[:, :, 2:4] = 1.0
_box_mask = np.zeros([hw, num_anchors, 1], dtype=np.float) + 0.01
# 获取先验框Anchor的大小
anchors = np.ascontiguousarray(cfg.anchors, dtype=np.float)
# 将预测偏移作用到Anchor上，得到真正的预测框
bbox_pred_np = np.expand_dims(bbox_pred_np, 0)
bbox_np = yolo_to_bbox(
    np.ascontiguousarray(bbox_pred_np, dtype=np.float),
    anchors,
    H, W)
# 得到预测边框的在原图尺度下的坐标
bbox_np = bbox_np[0]
bbox_np[:, :, 0::2] *= float(inp_size[0])
bbox_np[:, :, 1::2] *= float(inp_size[1])
gt_boxes_b = np.asarray(gt_boxes, dtype=np.float)
# 计算所有预测框与所有真实物体gt之间的IoU
bbox_np_b = np.reshape(bbox_np, [-1, 4])
ious = bbox_iou(
    np.ascontiguousarray(bbox_np_b, dtype=np.float),
    np.ascontiguousarray(gt_boxes_b, dtype=np.float))
)
# 如果一个预测框的最大IoU小于0.6，则其为负样本，标记_iou_mask为-IoU
best_ious = np.max(ious, axis=1).reshape(_iou_mask.shape)
iou_penalty = 0 - iou_pred_np[best_ious < cfg.iou_thresh]
_iou_mask[best_ious <= cfg.iou_thresh] = cfg.noobject_scale * iou_penalty
# 计算每一个gt属于哪一个区域（cell）
cell_w = float(inp_size[0]) / W
cell_h = float(inp_size[1]) / H
cx = (gt_boxes_b[:, 0] + gt_boxes_b[:, 2]) * 0.5 / cell_w
cy = (gt_boxes_b[:, 1] + gt_boxes_b[:, 3]) * 0.5 / cell_h
cell_inds = np.floor(cy) * W + np.floor(cx)
cell_inds = cell_inds.astype(np.int)
# 计算所有gt的位置偏移真值
target_boxes = np.empty(gt_boxes_b.shape, dtype=np.float)
target_boxes[:, 0] = cx - np.floor(cx)
target_boxes[:, 1] = cy - np.floor(cy)
target_boxes[:, 2] = \
    (gt_boxes_b[:, 2] - gt_boxes_b[:, 0]) / inp_size[0] * out_size[0]
target_boxes[:, 3] = \
    (gt_boxes_b[:, 3] - gt_boxes_b[:, 1]) / inp_size[1] * out_size[1]

```

```

# 对于每一个gt，选择与其有最大IoU的预测框作为正样本
gt_boxes_resize = np.copy(gt_boxes_b)
gt_boxes_resize[:, 0::2] *= (out_size[0] / float(inp_size[0]))
gt_boxes_resize[:, 1::2] *= (out_size[1] / float(inp_size[1]))
anchor_iou = anchor_intersections(
    anchors,
    np.asarray(gt_boxes_resize, dtype=np.float)
)
anchor_inds = np.argmax(anchor_iou, axis=0)
ious_reshaped = np.reshape(iou, [hw, num_anchors, len(cell_inds)])
# 接下来是遍历所有的gt，计算对应正样本的3个真值
for i, cell_ind in enumerate(cell_inds):
    if cell_ind >= hw or cell_ind < 0:
        print('cell inds size {}'.format(len(cell_inds)))
        print('cell over {} hw {}'.format(cell_ind, hw))
        continue
    a = anchor_inds[i]
    # 为每一个正样本计算对应的置信度iou的权重，这里权重为5(1-IoU)
    iou_pred_cell_anchor = iou_pred_np[cell_ind, a, :]
    _iou_mask[cell_ind, a, :] = cfg.object_scale * (1 - iou_pred_cell_anchor)
    # 赋予正样本iou真值
    _ious[cell_ind, a, :] = ious_reshaped[cell_ind, a, i]
    # 赋予正样本的边框偏移真值
    _box_mask[cell_ind, a, :] = cfg.coord_scale
    target_boxes[i, 2:4] /= anchors[a]
    _boxes[cell_ind, a, :] = target_boxes[i]
    # 计算正样本的类别真值与相应的权重
    _class_mask[cell_ind, a, :] = cfg.class_scale
    _classes[cell_ind, a, gt_classes[i]] = 1.
return _boxes, _ious, _classes, _box_mask, _iou_mask, _class_mask

```

---

在求得这6个量后，最终的损失也就变得十分简单了，三项损失均采用了均方差函数，可以直接调用PyTorch的nn.MSELoss()函数。

---

```

self.bbox_loss = nn.MSELoss(size_average=False)(bbox_pred *
                                              box_mask, _boxes * box_mask) / num_boxes
self.iou_loss = nn.MSELoss(size_average=False)(iou_pred *
                                             iou_mask, _ious * iou_mask) / num_boxes
self.cls_loss = nn.MSELoss(size_average=False)(prob_pred *
                                              class_mask, _classes * class_mask) / num_boxes

```

---

## 6.2.5 工程技巧

除了模型上的改进，YOLO v2也是一个充满工程技巧的检测模型，下面从两个方面介绍其工程上的特点。

### 1. 多尺度训练

由于移除了全连接层，因此YOLO v2可以接受任意尺寸的输入图片。在训练阶段，为了使模型对于不同尺度的物体鲁棒，YOLO v2采取了多种尺度的图片作为训练的输入。

由于下采样率为32，为了满足整除的需求，YOLO v2选取的输入尺度集合为{320,352,384,...,608}，这样训练出的模型可以预测多个尺度的物体。并且，输入图片的尺度越大则精度越高，尺度越低则速度越快，因此YOLO v2多尺度训练出的模型可以适应多种不同的场景要求。

### 2. 多阶段训练

由于物体检测数据标注成本较高，因此大多数物体检测模型都是先利用分类数据集来训练卷积层，然后再在物体检测数据集上训练。例如，YOLO v1先利用ImageNet上 $224 \times 224$ 大小的图像预训练，然后在 $448 \times 448$ 的尺度上进行物体检测的训练。这种转变使得模型要适应突变的图像尺度，增加了训练难度。

YOLO v2针对以上问题，优化了训练过程，采用如图6.6所示的训练方式，具体过程如下：

(1) 利用DarkNet网络在ImageNet上预训练分类任务，图像尺度为 $224 \times 224$ 。

(2) 将ImageNet图片放大到 $448 \times 448$ , 继续训练分类任务, 让模型首先适应变化的尺度。

(3) 去掉分类卷积层, 在DarkNet上增加Passthrough层及3个卷积层, 利用尺度为 $448 \times 448$ 的输入图像完成物体检测的训练。



图6.6 YOLO v2的多阶段训练

总体上来看, YOLO v2相较于之前的版本有了质的飞跃, 主要体现~~在吸收了其他算法的优点, 使用了先验框、特征融合等方法, 同时利用了多种训练技巧, 使得模型在保持极快速度的同时大幅度提升了检测的精度。YOLO v2已经达到了较高的检测水平, 但如果要分析其不足的话,~~大体有以下3点:

~~·单层特征图:~~ 虽然采用了Passthrough层来融合浅层的特征, 增强多尺度检测性能, 但仅仅采用一层特征图做预测, 纹理度仍然不够, 对~~小物体等检测提升有限, 并且没有使用残差这种较为简单、有效的结构。~~

~~·受限于其整体结构, 依然没有很好地解决小物体的检测问题。~~

~~·太工程化:~~ YOLO v2的整体实现有较多工程化调参的过程, 尤其是后续损失计算有些复杂, ~~不是特别“优雅”~~, 导致后续改进与扩展空间不足。

## 6.3 多尺度与特征融合：YOLO v3

针对YOLO v2的缺陷，2018年Joseph“大神”又推出了YOLO v3版本，将当今一些较好的检测思想融入到了YOLO中，在保持速度优势的前提下，进一步提升了检测精度，尤其是对小物体的检测能力。

具体来讲，YOLO v3主要改进了网络结构、网络特征及后续计算三个部分，下面对这三个部分进行详细讲解。

### 6.3.1 新网络结构DarkNet-53

YOLO v3继续吸收了当前优秀的检测框架的思想，如残差网络和特征融合等，提出了如图6.7所示的网络结构，称之为DarkNet-53。这里默认采用 $416 \times 416 \times 3$ 的输入，图中的各模块意义如下：

·DBL：代表卷积、BN及Leaky ReLU三层的结合，在YOLO v3中，卷积层都是以这样的组件出现的，构成了DarkNet的基本单元。DBL后面的数字代表有几个DBL模块。

·Res：图6.7中的Res代表残差模块，具体结构请参考第3章。Res后面的数字代表有几个串联的残差模块。

·上采样：上采样使用的方式为上池化，即元素复制扩充的方法使得特征尺寸扩大，没有学习参数。

·Concat：上采样后将深层与浅层的特征图进行Concat操作，即通道的拼接，类似于FPN，但FPN中使用的是逐元素相加。

从图6.7中可以看出新的DarkNet-53结构的一些新特性：

·残差思想：DarkNet-53借鉴了ResNet的残差思想，在基础网络中大量使用了残差连接，因此网络结构可以设计得很深，并且缓解了训练中梯度消失的问题，使得模型更容易收敛。

·多层特征图：通过上采样与Concat操作，融合了深、浅层的特征，最终输出了3种尺寸的特征图，用于后续预测。从前面的章节学习中可以知道，多层特征图对于多尺度物体及小物体检测是有利的。

·无池化层：之前的YOLO网络有5个最大池化层，用来缩小特征图

的尺寸，下采样率为32，而DarkNet-53并没有采用池化的做法，而是通过步长为2的卷积核来达到缩小尺寸的效果，下采样次数同样是5次，总体下采样率为32。

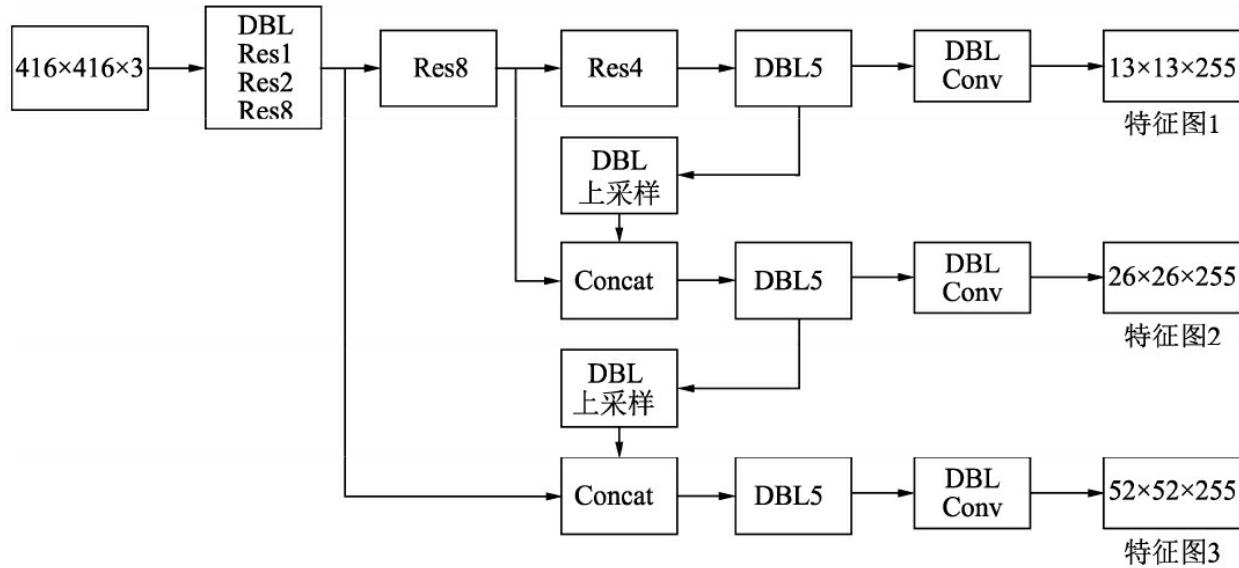


图6.7 YOLO v3网络结构

值得注意的是，YOLO v3的速度并没有之前的版本快，而是在保证实时性的前提下追求检测的精度。如果追求速度，YOLO v3提供了一个更轻量化的网络tiny-DarkNet，在模型大小与速度上，实现了SOTA（State of the Art）的效果。

### 6.3.2 多尺度预测

从图6.7中可以看到，YOLO v3输出了3个大小不同的特征图，从上到下分别对应深层、中层与浅层的特征。深层的特征图尺寸小，感受野大，有利于检测大尺度物体，而浅层的特征图则与之相反，更便于检测小尺度物体，这一点类似于FPN结构。

YOLO v3依然沿用了预选框Anchor，由于特征图数量不再是一个，因此匹配方法也要相应地进行改变。具体方法是，依然使用聚类的算法得到了9种不同大小宽高的先验框，然后按照表6.1所示的方法进行先验框的分配，这样，每一个特征图上的一个点只需要预测3个先验框，而不是YOLO v2中的5个。

表6.1 先验框在特征图上的分布

特征图大小	感 受 野	先 验 框
13×13	大	(116×90) (156×198) (373×326)
26×26	中	(30×61) (62×45) (59×119)
52×52	小	(10×13) (16×30) (33×23)

YOLO v3使用的方法有别于SSD，虽然都利用了多个特征图的信息，但SSD的特征是从浅到深地分别预测，没有深浅的融合，而YOLO v3的基础网络更像是SSD与FPN的结合。

YOLO v3默认使用了COCO数据集，一共有80个物体类别，因此一个先验框需要80维的类别预测值、4个位置预测及1个置信度预测，3个预测框一共需要 $3 \times (80 + 5) = 255$ 维，也就是每一个特征图的预测通道数。

### 6.3.3 Softmax改为Logistic

YOLO v3的另一个改进是使用了Logistic函数代替Softmax函数，以处理类别的预测得分。原因在于，Softmax函数输出的多个类别预测之间会相互抑制，只能预测出一个类别，而Logistic分类器相互独立，可以实现多类别的预测。

实验证明，Softmax可以被多个独立的Logistic分类器取代，并且准确率不会下降，这样的设计可以实现物体的多标签分类，例如一个物体如果是Women时，同时也属于Person这个类别。

值得注意的是，Logistic类别预测方法在Mask RCNN中也被采用，可以实现类别间的解耦。预测之后使用Binary的交叉熵函数可以进一步求得类别损失。

到写本书时，YOLO算法已经历经了3个版本的演变，在物体检测领域做出了卓越的成绩，以最先进的YOLO v3为例，其优缺点主要如下：

优点：速度快是YOLO系列最重要的特质，同时YOLO系列的通用性很强，由于其正样本生成过程较为严格，因此背景的误检率也较低。

缺点：位置的准确性较差，召回率也不高，尤其是对于遮挡与拥挤这种较难处理的情况，难以做到高精度。

## 6.4 总结

总体上，YOLO因为其较快的速度，在工业界应用极为广泛，尤其是在不追求预测框高精度的场景下。在与其他检测算法精度相同时，可以达到3到4倍的前向速度，是一个十分适合实际应用的检测框架。有趣的是，YOLO v1不依赖先验框Anchor而直接预测边框的思想在近一两年又得到了复兴，涌现了一大批Anchor-Free但检测精度优良的算法。

至此，笔者依次介绍完了当前最为流行的3大物体检测框架：Faster RCNN、SSD及YOLO。当然，要想实现一个优越的检测器，绝非是拿自己的数据训练这几个框架的模型这么简单，还有很多细节需要推敲。接下来的第7章就站在网络速度的角度上，介绍一些更轻量化的检测网络。

## 第3篇 物体检测的难点与发展

·第7章 模型加速之轻量化网络

·第8章 物体检测细节处理

·第9章 物体检测难点

·第10章 物体检测的未来发展

## 第7章 模型加速之轻量化网络

当前的经典物体检测结构大都依赖使用卷积网络进行特征提取，即Backbone，在前面的章节中我们也详细介绍过如VGGNet、ResNet等优秀的基础网络。但很遗憾，这些网络往往计算量巨大，当前依靠这些基础网络的检测算法很难达到实时运行的要求，尤其是在ARM、FPGA及ASIC等计算力有限的移动端硬件平台。因此如何将物体检测算法加速到满足工业应用的要求，是一个亟待解决的问题。

当前，实现模型加速的方法很多，如轻量化设计、BN层合并、剪枝与量化、张量分解、蒸馏与迁移学习等，这些方法相互之间并不独立，可以灵活地结合使用，如图7.1所示。

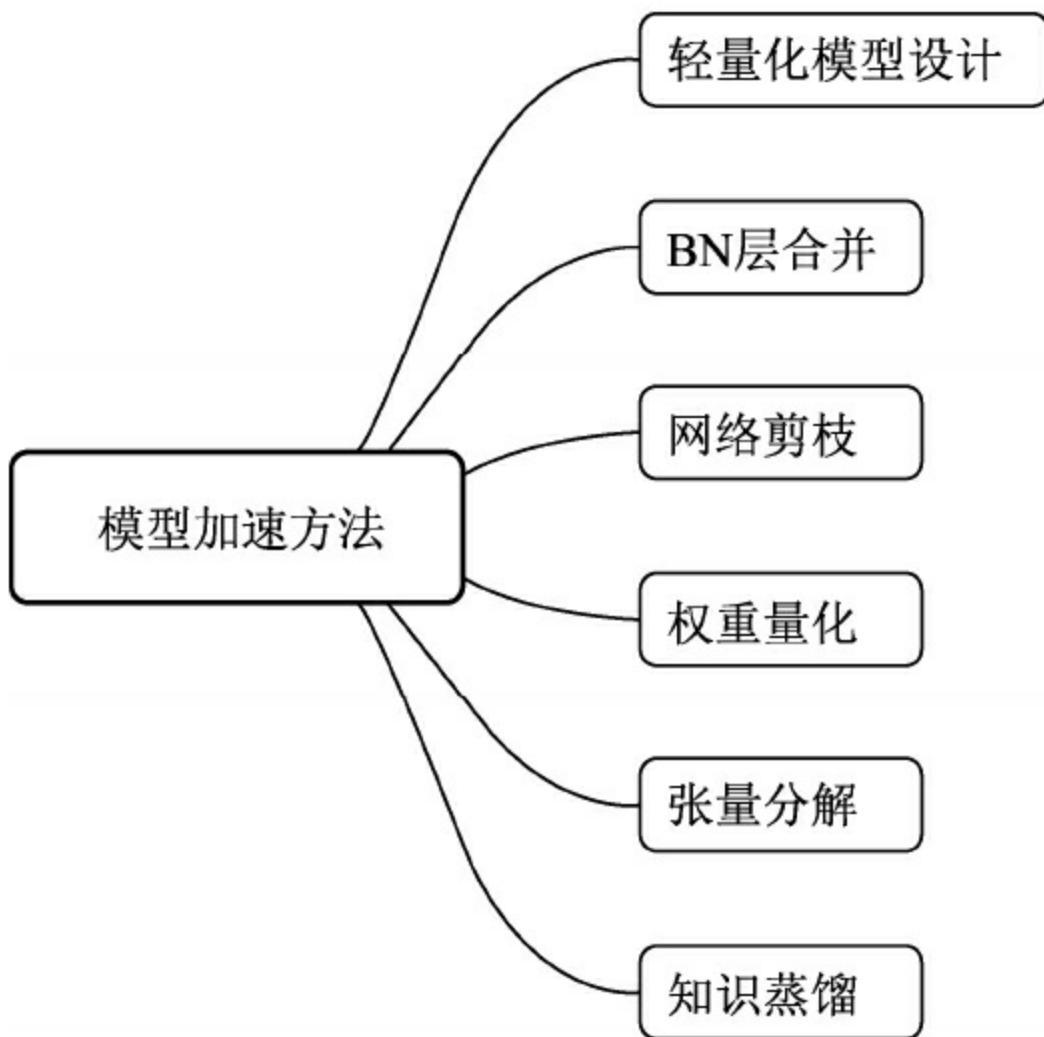


图7.1 模型加速的多种方法

·轻量化设计：从模型设计时就采用一些轻量化的思想，例如采用深度可分离卷积、分组卷积等轻量卷积方式，减少卷积过程的计算量。此外，利用全局池化来取代全连接层，利用 $1\times 1$ 卷积实现特征的通道降维，也可以降低模型的计算量，这两点在众多网络中已经得到了应用。

·BN层合并：在训练检测模型时，BN层可以有效加速收敛，并在一定程度上防止模型的过拟合，但在前向测试时，BN层的存在也增加了多余的计算量。由于测试时BN层的参数已经固定，因此可以在测试时将BN层的计算合并到卷积层，从而减少计算量，实现模型加速。

·**网络剪枝**：在卷积网络成千上万的权重中，存在着大量接近于0的参数，这些属于冗余参数，去掉后模型也可以基本达到相同的表达能力，因此有众多学者以此为出发点，搜索网络中的冗余卷积核，将网络稀疏化，称之为网络剪枝。具体来讲，网络剪枝有训练中稀疏与训练后剪枝两种方法。

·**权重量化**：是指将网络中高精度的参数量化为低精度的参数，从而加速计算的方法。高精度的模型参数拥有更大的动态变化范围，能够表达更丰富的参数空间，因此在训练中通常使用32位浮点数（单精度）作为网络参数的模型。训练完成后为了减小模型大小，通常可以将32位浮点数量化为16位浮点数的半精度，甚至是int8的整型、0与1的二值类型。典型方法如Deep Compression。

·**张量分解**：由于原始网络参数中存在大量的冗余，除了剪枝的方法以外，我们还可以利用SVD分解和PQ分解等方法，将原始张量分解为低秩的若干张量，以减少卷积的计算量，提升前向速度。

·**知识蒸馏**：通常来讲，大的模型拥有更强的拟合与泛化能力，而小模型的拟合能力较弱，并且容易出现过拟合。因此，我们可以使用大的模型指导小模型的训练，保留大模型的有效信息，实现知识的蒸馏。

对于轻量化的网络设计，目前较为流行的有SqueezeNet、MobileNet及ShuffleNet等结构，如图7.2所示。其中，SqueezeNet采用了精心设计的压缩再扩展的结构，MobileNet使用了效率更高的深度可分离卷积，而ShuffleNet提出了通道混洗的操作，进一步降低了模型的计算量。

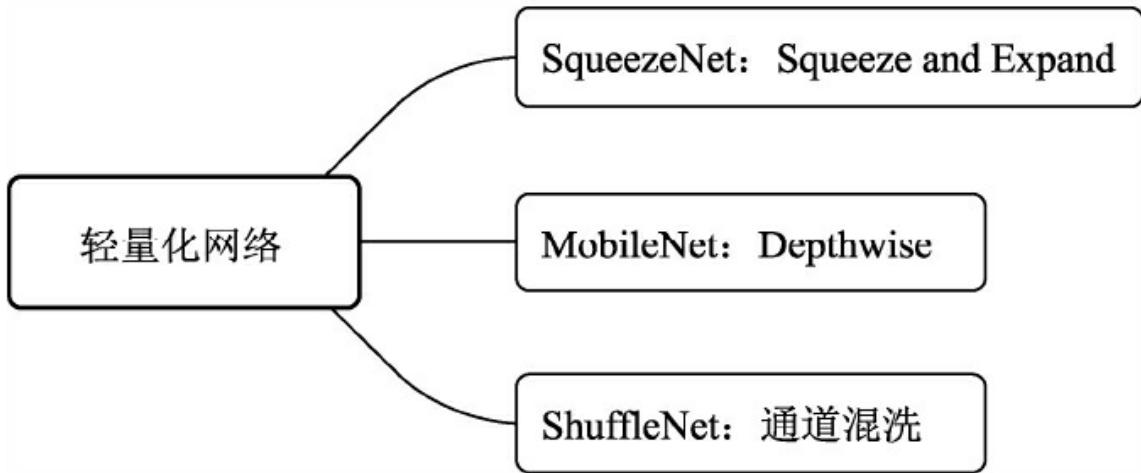


图7.2 多种轻量化卷积网络

本章主要从网络的轻量化设计出发，依次介绍SqueezeNet、MobileNet及ShuffleNet等网络结构的思想与实现方法，为物体检测技术移动端的部署与应用提供更强有力的方法。

## 7.1 压缩再扩展：SqueezeNet

当物体检测应用到实际工业场景时，模型的参数量是一个十分重要的指标，较小的模型可以高效地进行分布式训练，减小模型更新开销，降低平台体积功耗存储和计算能力的限制，方便在FPGA等边缘平台上部署。

基于以上几点，Han等人提出了轻量化模型SqueezeNet，其性能与AlexNet相近，而模型参数仅有AlexNet的1/50。在本节将首先介绍SqueezeNet的模型结构，然后对该模型进行总结与分析。

### 7.1.1 SqueezeNet网络结构

随着网络结构的逐渐加深，模型的性能有了大幅度提升，但这也增加了网络参数与前向计算的时间。SqueezeNet从网络结构优化的角度出发，使用了如下3点策略来减少网络参数，提升网络性能：

·使用 $1\times 1$ 卷积来替代部分的 $3\times 3$ 卷积，这也是之前介绍过的常用的策略，可以将参数减少为原来的 $1/9$ 。

·减少输入通道的数量，这一点也是通过 $1\times 1$ 卷积来实现，通道数量的减少可以使后续卷积核的数量也相应地减少。

·在减少通道数之后，使用多个尺寸的卷积核进行计算，以保留更多的信息，提升分类的准确率。

基于以上3点，SqueezeNet提出了如图7.3所示的基础模块，称之为Fire Module。图中输入特征尺寸为 $H\times W$ ，通道数为M，依次经过一个Squeeze层与Expand层，然后进行融合处理。

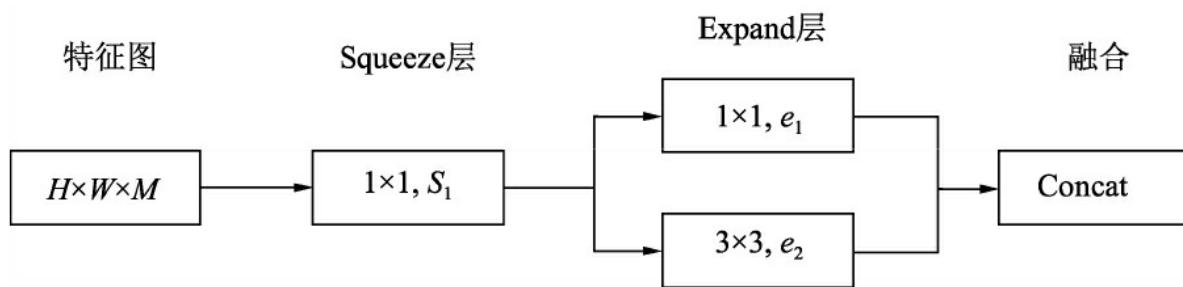


图7.3 Fire Module结构示意图

每一个模块的具体意义如下：

·SqueezeNet层：首先使用 $1\times 1$ 卷积进行降维，特征图的尺寸不变，

这里的S1小于M，达到了压缩的目的。

·Expand层：并行地使用 $1\times 1$ 卷积与 $3\times 3$ 卷积获得不同感受野的特征图，有点类似Inception模块，达到扩展的目的。

·Concat：对得到的两个特征图进行通道拼接，作为最终输出。

·模块中的 $S_1$ 、 $e_1$ 与 $e_2$ 都是可调的超参，Fire Module默认  
 $e_1=e_2=4\times S_1$ 。激活函数使用了ReLU函数。

下面使用PyTorch来搭建一个单独的SqueezeNet的Fire模块，新建一个squeezenet\_fire.py文件，代码如下：

---

```
import torch
from torch import nn
class Fire(nn.Module):
    def __init__(self, inplanes, squeeze_planes, expand_planes):
        super(Fire, self).__init__()
        # 这里的squeeze_planes为S1
        self.conv1 = nn.Conv2d(inplanes, squeeze_planes, kernel_size=1, stride=1)
        self.bn1 = nn.BatchNorm2d(squeeze_planes)
        self.relu1 = nn.ReLU(inplace=True)
        # expand_planes为e1和e2
        self.conv2 = nn.Conv2d(squeeze_planes, expand_planes, kernel_size=1,
                             stride=1)
        self.bn2 = nn.BatchNorm2d(expand_planes)
        self.conv3 = nn.Conv2d(squeeze_planes, expand_planes, kernel_size=3,
                             stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(expand_planes)
        self.relu2 = nn.ReLU(inplace=True)
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        out1 = self.conv2(x)
        out1 = self.bn2(out1)
        out2 = self.conv3(x)
        out2 = self.bn3(out2)
        # 对两个分支的输出进行Concat处理
        out = torch.cat([out1, out2], 1)
```

```
    out = self.relu2(out)
    return out
```

---

在终端中进入上述squeezeenet\_fire.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用该模块。

---

```
>>> import torch
>>> from squeezeenet_fire import Fire # 引入Fire模块
# 实例化Fire模块，输入通道为512，扩展通道为512
>>> fire_block = Fire(512, 128, 512).cuda()
>>> fire_block
Fire(
  (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1))
  (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
  (relu1): ReLU(inplace)
  (conv2): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1))
  (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
  (conv3): Conv2d(128, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
  (relu2): ReLU(inplace)
)
>>> input = torch.randn(1, 512, 28, 28).cuda()
>>> output = fire_block(input)
# 输出通道数为512×2=1024
>>> output.shape
torch.Size([1, 1024, 28, 28])
```

---

基于Fire Module，SqueezeNet的网络结构如图7.4所示。输入图像首先送入Conv 1，得到通道数为96的特征图，然后依次使用8个Fire Module，通道数也逐渐增加。图7.4中横线上的值代表了通道数。最后一个卷积为Conv 10，输入通道数为N的特征图，N代表需要物体的类别数。

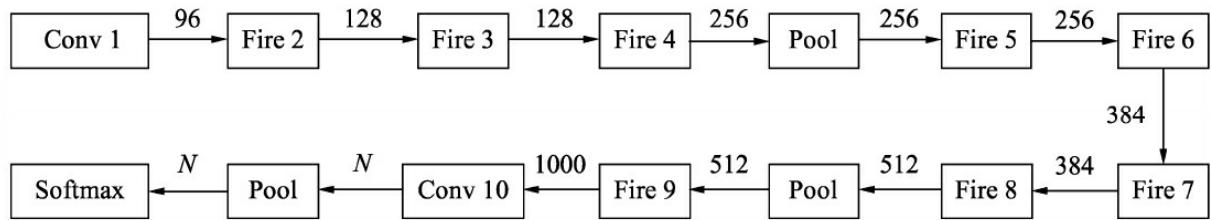


图7.4 SqueezeNet网络整体结构

SqueezeNet一共使用了3个Pool层，前两个是Max Pooling层，步长为2，最后一个为全局平均池化，利用该层可以取代全连接层，减少了计算量。

## 7.1.2 SqueezeNet总结

SqueezeNet是一个精心设计的轻量化网络，使用常见的模型压缩技术，如SVD、剪枝和量化等，可以进一步压缩该模型的大小。例如，使用Deep Compresion技术对其进行压缩时，在几乎不损失性能的前提下，模型大小可以压缩到0.5MB。

基于其轻量化的特性，SqueezeNet可以广泛地应用到移动端，促进了物体检测技术在移动端的部署与应用。

## 7.2 深度可分离：MobileNet

SqueezeNet虽在一定程度上减少了卷积计算量，但仍然使用传统的卷积计算方式，而在其后的MobileNet利用了更为高效的深度可分离卷积的方式，进一步加速了卷积网络在移动端的应用。

为了更好地理解深度可分离卷积，在本节首先回顾标准的卷积计算过程，然后详细讲解深度可分离卷积过程，以及基于此结构的两个网络结构MobileNet v1与MobileNet v2。

### 7.2.1 标准卷积

假设当前特征图大小为 $C_i \times H \times W$ , 需要输出的特征图大小为 $C_o \times H \times W$ , 卷积核大小为 $3 \times 3$ , Padding为1, 则标准卷积的计算过程如图7.5所示。

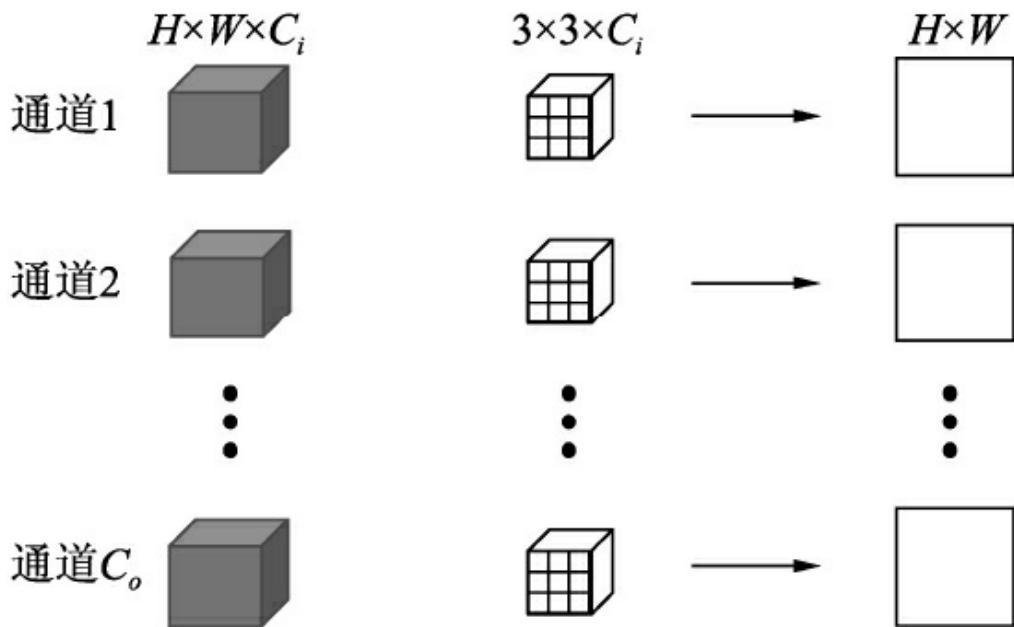


图7.5 标准卷积的计算过程

标准卷积的过程如下:

·对于输入特征图的左上 $C_i \times 3 \times 3$ 特征, 利用 $C_i \times 3 \times 3$ 大小的卷积核进行点乘并求和, 得到输出特征图中一个通道上的左上点, 这一步操作的计算量为 $C_i \times 3 \times 3$ 。

·在输入特征图上进行滑窗, 重复第一步操作, 最终得到输出特征图中一个通道的 $H \times W$ 大小的输出, 总计算量为 $C_i \times 3 \times 3 \times H \times W$ 。这一步完

成了图7.5中一个通道的过程。

·利用 $C_o$ 个上述大小的卷积核，重复第一步过程，最终得到 $C_o \times H \times W$ 大小的特征图。

在整个标准卷积计算过程中，所需的卷积核参数量为 $C_i \times 3 \times 3 \times C_o$ ，总的计算量如式（7-1）所示。

$$\underline{F_s = C_i \times 3 \times 3 \times H \times W \times C_o} \quad (7-1)$$

需要注意，这里的计算量仅仅是指乘法操作，而没有将加法计算在内。

## 7.2.2 深度可分离卷积

标准卷积在卷积时，同时考虑了图像的区域与通道信息，那么为什么不能分开考虑区域与通道呢？基于此想法，诞生了深度可分离卷积（Depthwise Separable Convolution），将卷积的过程分为逐通道卷积与逐点 $1\times 1$ 卷积两步。虽然深度可分离卷积将一步卷积过程扩展为两步，但减少了冗余计算，因此总体上计算量有了大幅度降低。MobileNet也大量采用了深度可分离卷积作为基础单元。

逐通道卷积的计算过程如图7.6所示，对于一个通道的输入特征 $H\times W$ ，利用一个 $3\times 3$ 卷积核进行点乘求和，得到一个通道的输出 $H\times W$ 。然后，对于所有的输入通道 $C_i$ ，使用 $C_i$ 个 $3\times 3$ 卷积核即可得到 $C_i\times H\times W$ 大小的输出。

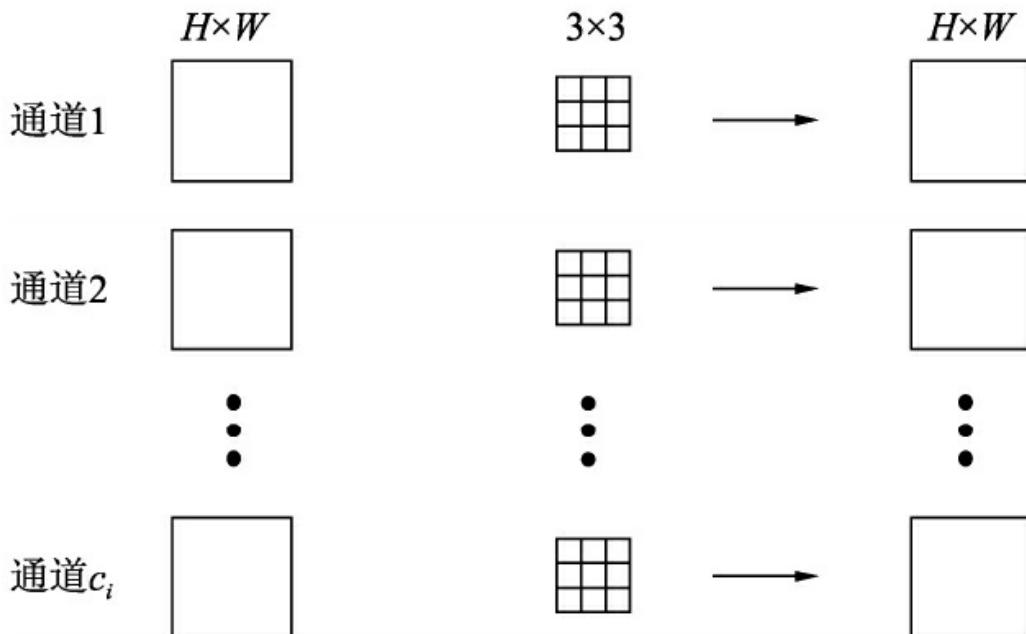


图7.6 逐通道卷积的计算过程

综合上述计算过程，逐通道卷积有如下几个特点：

·卷积核参数量为 $C_i \times 3 \times 3$ ，远少于标准卷积 $C_i \times 3 \times 3 \times C_o$ 的数量。

·通道之间相互独立，没有各通道间的特征融合，这也是逐通道卷积的核心思想，例如图7.6中输出特征的每一个点只对应输入特征一个通道上的 $3 \times 3$ 大小的特征，而不是标准卷积中 $C_i \times 3 \times 3$ 大小。

·由于只在通道间进行卷积，导致输入与输出特征图的通道数相同，无法改变通道数。

逐通道卷积的总计算量如式（7-2）所示。

$$F_d = C_i \times 3 \times 3 \times H \times W \quad (7-2)$$

由于逐通道卷积通道间缺少特征的融合，并且通道数无法改变，因此后续还需要继续连接一个逐点的 $1 \times 1$ 的卷积，一方面可以融合不同通道间的特征，同时也可以改变特征图的通道数。由于这里 $1 \times 1$ 卷积的输入特征图大小为 $C_i \times H \times W$ ，输出特征图大小为 $C_o \times H \times W$ ，因此这一步的总计算量如式（7-3）所示。

$$F_i = C_i \times 1 \times 1 \times H \times W \times C_o \quad (7-3)$$

综合这两步，可以得到深度可分离卷积与标准卷积的计算量之比，如式（7-4）所示。

$$r = \frac{F_d + F_i}{F_s} = \frac{C_i \times 3 \times 3 \times H \times W + C_i \times 1 \times 1 \times H \times W \times C_o}{C_i \times 3 \times 3 \times H \times W \times C_o} = \frac{1}{C_o} + \frac{1}{9} \approx \frac{1}{9} \quad (7-4)$$

可以看到，虽然深度可分离卷积将卷积过程分为了两步，但凭借其轻量的卷积方式，总体计算量约等于标准卷积的 $1/9$ ，极大地减少了卷

积过程的计算量。

MobileNet v1使用的深度可分离模块的具体结构如图7.7所示。其中使用了BN层及ReLU的激活函数。值得注意的是，在此使用了ReLU6来替代原始的ReLU激活函数，将ReLU的最大输出限制在6以下。



图7.7 深度可分离卷积模块结构图

使用ReLU6的原因主要是为了满足移动端部署的需求。移动端通常使用Float16或者Int8等较低精度的模型，如果不对激活函数的输出进行限制的话，激活值的分布范围会很大，而低精度的模型很难精确地覆盖如此大范围的输出，这样会带来精度的损失。

### 7.2.3 MobileNet v1结构

MobileNet v1整体的网络是由上述深度可分离卷积基本单元组成的，具体结构如图7.8所示。与VGGNet类似，也是一个逐层堆叠式网络。图中的Dw代表一个深度分解卷积，其后需要跟一个 $1\times 1$ 卷积，s2代表步长为2的卷积，可以缩小特征图尺寸，起到与Pooling层一样的作用。

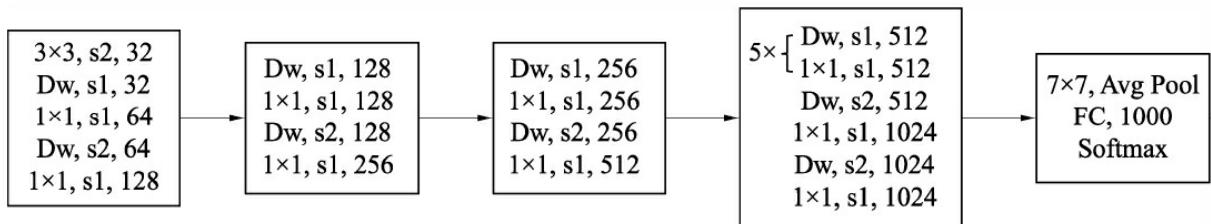


图7.8 MobileNet整体结构图

这里列出的是MobileNet v1用于物体分类的网络，可以看到网络最后利用一个全局平均池化层，送入到全连接与Softmax进行分类预测。如果用于物体检测，只需要在之前的特征图上进行特征提取即可。

在基本的结构之外，MobileNet v1还设置了两个超参数，用来控制模型的大小与计算量，具体如下：

·宽度乘子：用于控制特征图的通道数，记做 $\alpha$ ，当 $\alpha < 1$ 时，模型会变得更薄，可以将计算量减少为原来的 $\alpha^2$ 。

·分辨率乘子：用于控制特征图的尺寸，记做 $\rho$ ，在相应的特征图上应用该乘子，也可以有效降低每一层的计算量。

在实现层面上，PyTorch中卷积层提供了一个组卷积参数，可以方便地实现深度可分离卷积。下面利用PyTorch搭建一个MobileNet v1网

络，新建一个mobilenet\_v1.py，代码如下：

---

```
from torch import nn
class MobileNet(nn.Module):
    def __init__(self):
        super(MobileNet, self).__init__()
        # 标准卷积
        def conv_bn(dim_in, dim_out, stride):
            return nn.Sequential(
                nn.Conv2d(dim_in, dim_out, 3, stride, 1, bias=False),
                nn.BatchNorm2d(dim_out),
                nn.ReLU(inplace=True)
            )
        # 深度分解卷积
        def conv_dw(dim_in, dim_out, stride):
            return nn.Sequential(
                nn.Conv2d(dim_in, dim_in, 3, stride, 1, groups= dim_in, bias=False),
                nn.BatchNorm2d(dim_in),
                nn.ReLU(inplace=True),
                nn.Conv2d(dim_in, dim_out, 1, 1, 0, bias=False),
                nn.BatchNorm2d(dim_out),
                nn.ReLU(inplace=True),
            )
        # MobileNet每一个卷积组的结构参数
        self.model = nn.Sequential(
            conv_bn( 3, 32, 2),
            conv_dw( 32, 64, 1),
            conv_dw( 64, 128, 2),
            conv_dw(128, 128, 1),
            conv_dw(128, 256, 2),
            conv_dw(256, 256, 1),
            conv_dw(256, 512, 2),
            conv_dw(512, 512, 1),
            conv_dw(512, 512, 1),
            conv_dw(512, 512, 1),
            conv_dw(512, 512, 1),
            conv_dw(512, 1024, 2),
            conv_dw(1024, 1024, 1),
            nn.AvgPool2d(7),
        )
        self.fc = nn.Linear(1024, 1000)
    def forward(self, x):
        x = self.model(x)
        x = x.view(-1, 1024)                                # 前向过程中在全连接前将特征图平展成1维
        x = self.fc(x)
```

```
    return x
```

---

在终端中进入上述mobilenet\_v1.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用该模块。

---

```
>>> import torch
>>> from mobilenet_v1 import MobileNet
>>> mobilenet = MobileNet().cuda()          # 构建mobilenet的实例
# 打印mobilenet_v1网络结构，包含14个卷积模块，1个全局池化及最后的全连接
>>> mobilenet
MobileNet(
  (model1): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
      bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (2): ReLU(inplace)
    )
    (1): Sequential(
      (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      groups=32, bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (2): ReLU(inplace)
      (3): Conv2d(32, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (5): ReLU(inplace)
    )
    (2): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
      groups=64, bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (2): ReLU(inplace)
      (3): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
      (5): ReLU(inplace)
    )
    (3): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
      groups=128, bias=False)
```

```
(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(4): Sequential(
(0): Conv2d(128,128,kernel_size=(3,3),stride=(2,2),padding=(1,1),
groups=128,bias=False)
(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(5): Sequential(
(0): Conv2d(256,256,kernel_size=(3,3),stride=(1,1),padding=(1,1),
groups=256, bias=False)
(1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(6): Sequential(
(0): Conv2d(256,256,kernel_size=(3,3),stride=(2,2),padding=(1,1),
groups=256, bias=False)
(1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(7): Sequential(
(0): Conv2d(512,512,kernel_size=(3,3),stride=(1,1),padding=(1,1),
groups=512, bias=False)
(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
(4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(8): Sequential(
(0): Conv2d(512, 512, kernel_size=(3,3), stride=(1,1), padding=(1,1),
groups=512, bias=False)
(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(9): Sequential(
(0): Conv2d(512, 512, kernel_size=(3,3), stride=(1,1), padding=(1,1),
groups=512, bias=False)
(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(10): Sequential(
(0): Conv2d(512, 512, kernel_size=(3,3), stride=(1,1), padding=(1,1),
groups=512, bias=False)
(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
(11): Sequential(
(0): Conv2d(512, 512, kernel_size=(3,3), stride=(1,1), padding=(1,1),
groups=512, bias=False)
(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU(inplace)
(3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
(4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU(inplace)
)
```

```
(12): Sequential(
    (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
    groups=512, bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
    running_stats=True)
    (2): ReLU(inplace)
    (3): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_
    running_stats=True)
    (5): ReLU(inplace)
)
(13): Sequential(
    (0): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
    groups=1024, bias=False)
    (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_
    running_stats=True)
    (2): ReLU(inplace)
    (3): Conv2d(1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_
    running_stats=True)
    (5): ReLU(inplace)
)
(14): AvgPool2d(kernel_size=7, stride=7, padding=0)
)
(fc): Linear(in_features=1024, out_features=1000, bias=True)
)
# 输入Tensor，并转移到CUDA上
>>> input = torch.randn(1, 3, 224, 224).cuda()
>>> output = mobilenet(input)
>>> output.shape                                     # 最终输出维度为1000的特征
torch.Size([1, 1000])
```

---

#### 7.2.4 MobileNet v1总结

总体上，MobileNet v1利用深度可分离的结构牺牲了较小的精度，带来了计算量与网络层参数的大幅降低，从而也减小了模型的大小，方便应用于移动端。

但MobileNet v1也有其自身结构带来的缺陷，主要有以下两点：

- 模型结构较为复古，采用了与VGGNet类似的卷积简单堆叠，没有采用残差、特征融合等先进的结构。
- 深度分解卷积中各通道相互独立，卷积核维度较小，输出特征中只有较少的输入特征，再加上ReLU激活函数，使得输出很容易变为0，难以恢复正常训练，因此在训练时部分卷积核容易被训练废掉。

## 7.2.5 MobileNet v2

针对以上MobileNet v1的缺点，2018年诞生的MobileNet v2吸收了残差网络的思想，主要从两个方面改善了网络结构，提升了MobileNet的检测精度。

首先，MobileNet v2利用残差结构取代了原始的卷积堆叠方式，提出了一个Inverted Residual Block结构，如图7.9所示。依据卷积的步长，该结构可分为两种情形，在步长为1时使用了残差连接，融合的方式为逐元素相加。

相比于MobileNet v1与原始的残差结构，这里有两点不同：

### 1. Inverted Residual Block结构

由于 $3 \times 3$ 卷积处的计算量较大，因此传统的残差网络通常先使用 $1 \times 1$ 卷积进行特征降维，减少通道数，再送入 $3 \times 3$ 卷积，最后再利用 $1 \times 1$ 卷积升维。这种结构从形状上看中间窄两边宽，类似于沙漏形状。

然而在MobileNet v2中，由于使用了深度可分离卷积来逐通道计算，本身计算量就比较少，因此在此可以使用 $1 \times 1$ 卷积来升维，在计算量增加不大的基础上获取更好的效果，最后再用 $1 \times 1$ 卷积降维。这种结构中间宽两边窄，类似于柳叶，该结构也因此被称为Inverted Residual Block。

### 2. 去掉ReLU6

深度可分离卷积得到的特征对应于低维空间，特征较少，如果后续接线性映射则能够保留大部分特征，而如果接非线性映射如ReLU，则会破坏特征，造成特征的损耗，从而使得模型效果变差。

针对此问题，MobileNet v2直接去掉了每一个Block中最后的ReLU6层，减少了特征的损耗，获得了更好的检测效果。

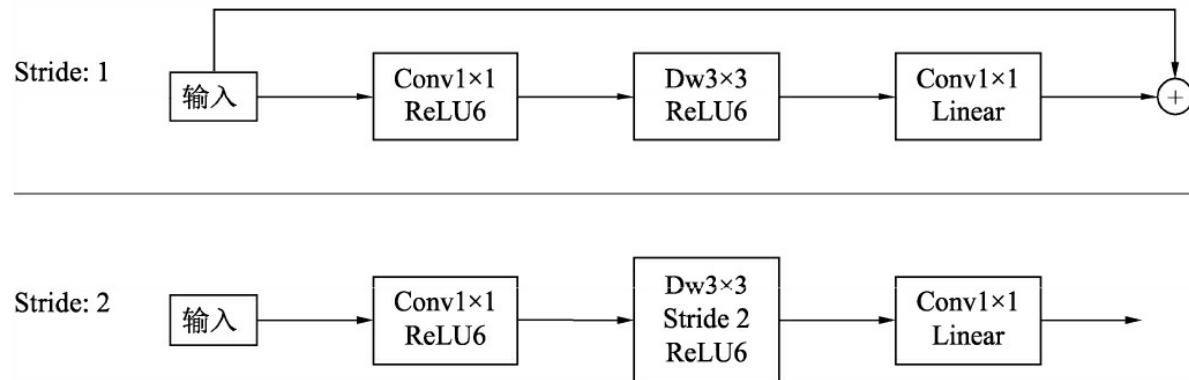


图7.9 MobileNet v2的残差结构

下面利用PyTorch构建MobileNet v2的残差模块，即图7.9中的stride为1的结构，新建一个mobilenet\_v2\_block.py文件。代码如下：

```
from torch import nn
class InvertedResidual(nn.Module):
    def __init__(self, inp, oup, stride, expand_ratio):
        super(InvertedResidual, self).__init__()
        self.stride = stride
        # 中间扩展层的通道数
        hidden_dim = round(inp * expand_ratio)
        self.conv = nn.Sequential(
            # 1x1的逐点卷积进行升维
            nn.Conv2d(inp, hidden_dim, 1, 1, 0, bias=False),
            nn.BatchNorm2d(hidden_dim),
            nn.ReLU6(inplace=True),
            # 深度可分离模块
            nn.Conv2d(hidden_dim, hidden_dim, 3, stride, 1, groups=hidden_dim, bias=False),
            nn.BatchNorm2d(hidden_dim),
            nn.ReLU6(inplace=True),
            # 1x1的逐点卷积进行降维
            nn.Conv2d(hidden_dim, oup, 1, 1, 0, bias=False),
            nn.BatchNorm2d(oup),
        )
    def forward(self, x):
        return x + self.conv(x)
```

---

在终端中进入上述mobilenet\_v2\_block.py文件的同级目录，输入python3进入交互式环境，利用下面的代码调用该模块。

---

```
>>> import torch
>>> from mobilenet_v2_block import InvertedResidual
# 实例化模块，并传入响应参数
>>> block = InvertedResidual(24, 24, 1, 6).cuda()
# 打印出该block，包含了3个卷积层
>>> block
InvertedResidual(
(conv): Sequential(
(0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
(1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(2): ReLU6(inplace)
(3): Conv2d(144, 144, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=144, bias=False)
(4): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
(5): ReLU6(inplace)
(6): Conv2d(144, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
(7): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
)
)
>>> input = torch.randn(1, 24, 56, 56).cuda()
>>> output = block(input)
>>> output.shape # 输出特征图的大小与输入完全相同
torch.Size([1, 24, 56, 56])
```

---

总体上，MobileNet v2在原结构的基础上进行了简单的修改，通过较少的计算量即可获得较高的精度，非常适合于移动端的部署。

## 7.3 通道混洗：ShuffleNet

为了降低计算量，当前先进的卷积网络通常在 $3\times 3$ 卷积之前增加一个 $1\times 1$ 卷积，用于通道间的信息流通与降维。然而在ResNeXt、MobileNet等高性能的网络中， $1\times 1$ 卷积却占用了大量的计算资源。

2017年的ShuffleNet v1从优化网络结构的角度出发，利用组卷积与通道混洗（Channel Shuffle）的操作有效降低了 $1\times 1$ 逐点卷积的计算量，是一个极为高效的轻量化网络。而2018年的ShuffleNet v2则在ShuffleNet v1版本的基础上实现了更为优越的性能，本节将详细介绍这两个ShuffleNet网络的思想与结构。

### 7.3.1 通道混洗

当前先进的轻量化网络大都使用深度可分离卷积或者组卷积，以降低网络的计算量，但这两种操作都无法改变特征的通道数，因此还需要使用 $1\times 1$ 卷积。总体来讲，逐点的 $1\times 1$ 卷积有如下两点特性：

- 可以促进通道之间信息的融合，改变通道至指定维度。
- 轻量化网络中 $1\times 1$ 卷积占据了大量的计算，并且致使通道之间充满约束，一定程度上降低了模型的精度。

为了进一步降低计算量，ShuffleNet提出了通道混洗的操作，通过通道混洗也可以完成通道之间信息的融合，具体结构如图7.10所示。

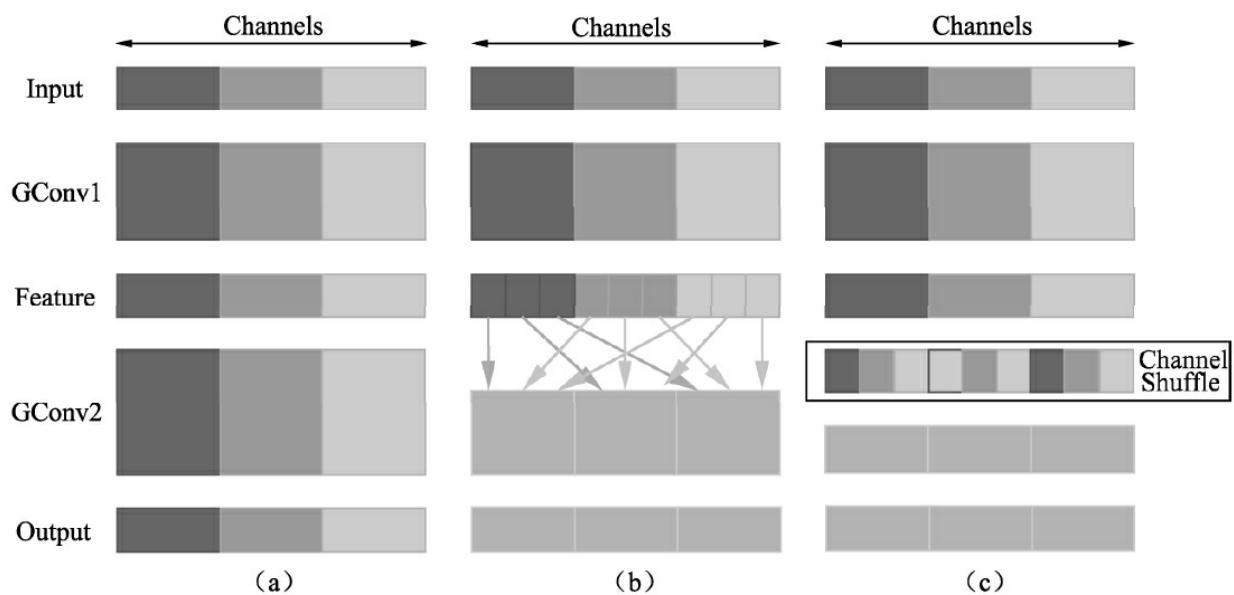


图7.10 通道混洗操作示意图

图7.10中a图代表了常规的两个组卷积操作，可以看到，如果没有逐点的 $1\times 1$ 卷积或者通道混洗，最终输出的特征仅由一部分输入通道的特

征计算得出，这种操作阻碍了信息的流通，进而降低了特征的表达能力。

因此，我们希望在一个组卷积之后，能够将特征图之间的通道信息进行融合，类似于图7.10中b的操作，将每一个组的特征分散到不同的组之后，再进行下一个组卷积，这样输出的特征就能够包含每一个组的特征，而通道混洗恰好可以实现这个过程，如图7.10的c图所示。

通道混洗可以通过几个常规的张量操作巧妙地实现，如图7.11所示。为了更好地讲解实现过程，这里对输入通道做了1~12的编号，一共包含3个组，每个组包含4个通道。

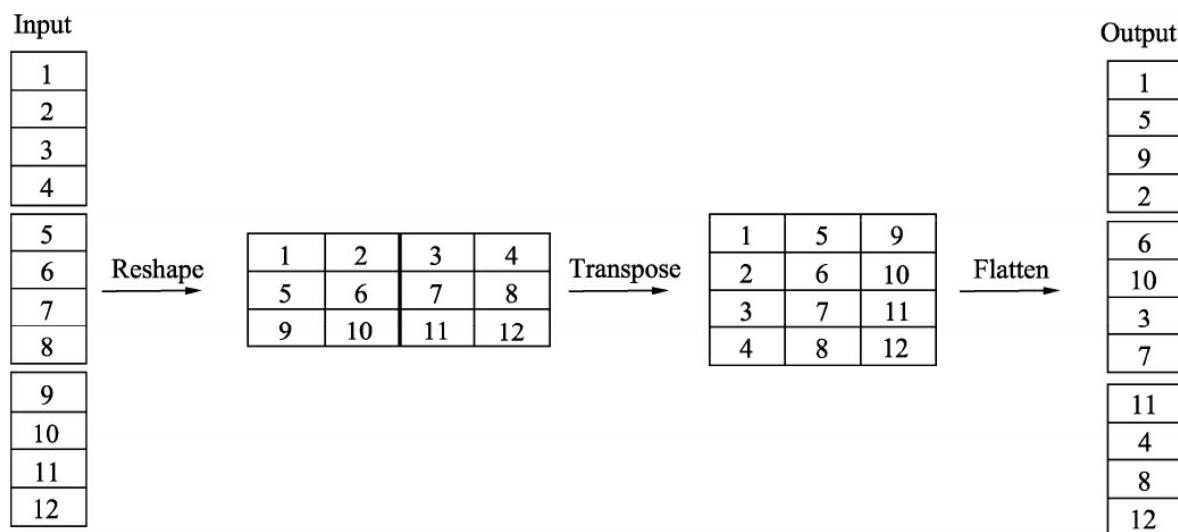


图7.11 通道混洗的具体实现过程

下面详细介绍混洗过程中使用到的3个操作：

·**Reshape**：首先将输入通道一个维度Reshape成两个维度，一个是卷积组数，一个是每个卷积组包含的通道数。

·**Transpose**：将扩展出的两维进行置换。

·**Flatten**: 将置换后的通道**Flatten**平展后即可完成最后的通道混洗。

下面从代码角度讲解一下通道混洗的实现过程。

---

```
def channel_shuffle(x, groups):
    batchsize, num_channels, height, width = x.data.size()
    channels_per_group = num_channels // groups
    # Reshape操作, 将通道扩展为两维
    x = x.view(batchsize, groups, channels_per_group, height, width)
    # Transpose操作, 将组卷积两个维度进行置换
    x = torch.transpose(x, 1, 2).contiguous()
    # Flatten操作, 两个维度平展成一个维度
    x = x.view(batchsize, -1, height, width)
    return x
```

---

### 7.3.2 网络结构

利用上述通道混洗操作，ShuffleNet构建出了如图7.12所示的网络基本模块。

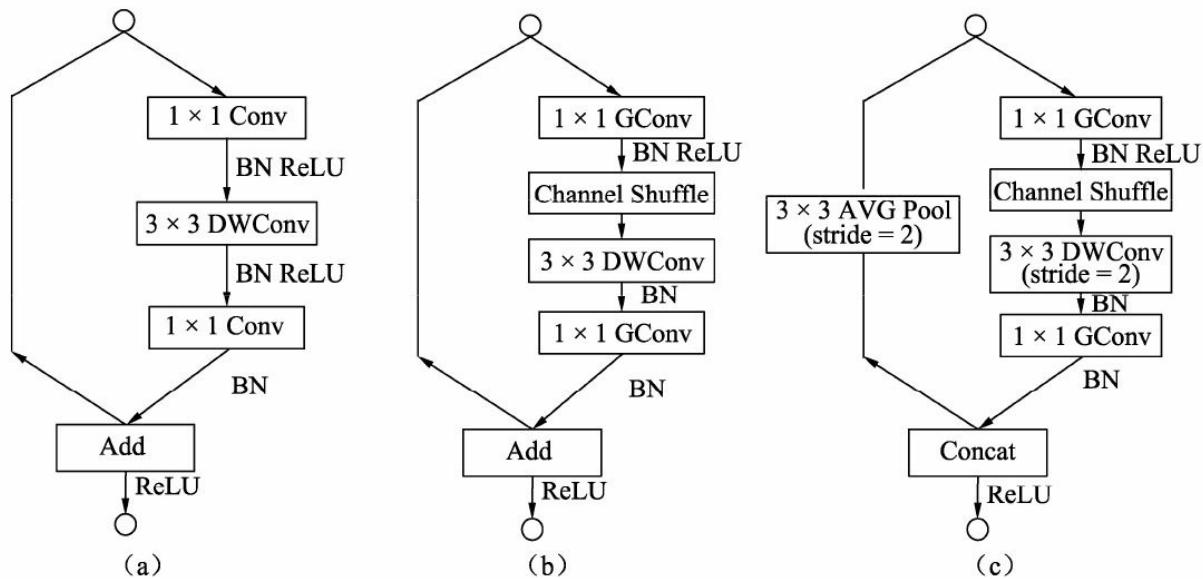


图7.12 ShuffleNet基本结构单元

图7.12的a图是一个带有深度可分离卷积的残差模块，这里的 $1\times 1$ 是逐点的卷积。相比深度可分离卷积， $1\times 1$ 计算量较大。

图7.12的b图则是基本的ShuffleNet基本单元，可以看到 $1\times 1$ 卷积采用的是组卷积，然后进行通道的混洗，这两步可以取代 $1\times 1$ 的逐点卷积，并且大大降低了计算量。 $3\times 3$ 卷积仍然采用深度可分离的方式。

图7.12的c图是带有降采样的ShuffleNet单元，在旁路中使用了步长为2的 $3\times 3$ 平均池化进行降采样，在主路中 $3\times 3$ 卷积步长为2实现降采样。另外，由于降采样时通常要伴有通道数的增加，ShuffleNet直接将两分支拼接在一起实现了通道数的增加，而不是常规的逐点相加。

得益于组卷积与通道混洗，ShuffleNet的基本单元可以很高效地进行计算。在该基本单元的基础上，ShuffleNet的整体网络结构如表7.1所示。

表7.1 ShuffleNet v1网络整体结构图

Layer	Output	Ksize	Repeat	Output Channels (g groups)				
				g=1	g=2	g=3	g=4	g=8
Image	224×224			3	3	3	3	3
Convl	112×112	3×3	2	24	24	24	24	24
Maxpool	56×56	3×3	2					
Stage2	28×28		2	144	200	240	272	384
	28×28		1	144	200	240	272	384
Stage3	14×14		2	288	400	480	544	768

(续)

Layer	Output	Ksize	Repeat	Output Channels (g groups)				
				g=1	g=2	g=3	g=4	g=8
Stage3	14×14		1	288	400	480	544	768
Stage4	7×7		2	576	800	960	1088	1536
	7×7		1	576	800	960	1088	1536
GlobalPool	1×1	7×7						
FC				1000	1000	1000	1000	1000
Complexity				143M	140M	137M	133M	137M

关于ShuffleNet的整体结构，有以下3点需要注意：

- g代表组卷积的组数，以控制卷积连接的稀疏性。组数越多，计算量越少，因此在相同的计算资源，可以使用更多的卷积核以获取更多的通道数。

- ShuffleNet在3个阶段内使用了其特殊的基本单元，这3个阶段的第一个Block的步长为2以完成降采样，下一个阶段的通道数是上一个的两倍。

·深度可分离卷积虽然可以有效降低计算量，但其存储访问效率较差，因此第一个卷积并没有使用ShuffleNet基本单元，而是只在后续3个阶段使用。

下面以g=3为例，从代码层面讲解ShuffleNet网络的构建。

---

```
from torch import nn
import torch.nn.functional as F
class ShuffleNet(nn.Module):
    def __init__(self, groups=3, in_channels=3, num_classes=1000):
        super(ShuffleNet, self).__init__()
        self.groups = groups
        # 3个阶段的Block数量
        self.stage_repeats = [3, 7, 3]
        self.in_channels = in_channels
        self.num_classes = num_classes
        # g为3时，每一个阶段输出特征的通道数
        self.stage_out_channels = [-1, 24, 240, 480, 960]
        self.conv1 = conv3x3(self.in_channels,
                            self.stage_out_channels[1],           # stage 1
                            stride=2)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        # 单独构建3个阶段
        self.stage2 = self._make_stage(2)
        self.stage3 = self._make_stage(3)
        self.stage4 = self._make_stage(4)
        # 全连接层，当前模型用于物体分类
        num_inputs = self.stage_out_channels[-1]
        self.fc = nn.Linear(num_inputs, self.num_classes)
    def _make_stage(self, stage):
        modules = OrderedDict()
        stage_name = "ShuffleUnit_Stage{}".format(stage)
        # 在第二个阶段之后开始使用组卷积
        grouped_conv = stage > 2
        # 每个阶段的第一个模块使用的是Concat，因此需要单独构建
        first_module = ShuffleUnit(
            self.stage_out_channels[stage-1],
            self.stage_out_channels[stage],
            groups=self.groups,
            grouped_conv=grouped_conv,
            combine='concat'
        )
        modules[stage_name+"_0"] = first_module
        # 重复构建每个阶段剩余的Block模块
```

```
for i in range(self.stage_repeats[stage-2]):
    name = stage_name + "_{}".format(i+1)
    module = ShuffleUnit(
        self.stage_out_channels[stage],
        self.stage_out_channels[stage],
        groups=self.groups,
        grouped_conv=True,
        combine='add'
    )
    modules[name] = module
return nn.Sequential(modules)

def forward(self, x):
    x = self.conv1(x)
    x = self.maxpool(x)
    x = self.stage2(x)
    x = self.stage3(x)
    x = self.stage4(x)
    # 全局的平均池化层
    x = F.avg_pool2d(x, x.data.size()[-2:])
    # 将特征平展，并送入全连接层与Softmax，得到最终预测概率
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return F.log_softmax(x, dim=1)
```

---

总体上，ShuffleNet提出了一个巧妙的通道混洗模块，在几乎不影响准确率的前提下，进一步地降低了计算量，性能优于ResNet和Xception等网络，因此也更适合部署在移动设备上。

### 7.3.3 ShuffleNet v2

在2018年，旷视的团队进一步升级了ShuffleNet，提出了新的版本ShuffleNet v2。相比于ShuffleNet v1，ShuffleNet v2进一步分析了影响模型速度的因素，提出了新的规则，并基于此规则，改善了原版本的不足。

原有的一些轻量化方法在衡量模型性能时，通常使用浮点运算量FLOPs（Floating Point Operations）作为主要指标。FLOPs是指模型在进行一次前向传播时所需的浮点计算次数，其单位为FLOP，可以用来衡量模型的复杂度。

然而，通过一系列实验发现ShuffleNet v2仅仅依赖FLOPs是有问题的，FLOPs近似的网络会存在不同的速度，还有另外两个重要的指标：内存访问时间（Memory Access Cost，MAC）与网络的并行度。

以此作为出发点，ShuffleNet v2做了大量的实验，分析影响网络运行速度的原因，提出了建立高性能网络的4个基本规则：

（1）卷积层的输入特征与输出特征通道数相等时，MAC最小，此时模型速度最快。

（2）过多的组卷积会增加MAC，导致模型的速度变慢。

（3）网络的碎片化会降低可并行度，这表明模型中分支数量越少，模型速度会越快。

（4）逐元素（Element Wise）操作虽然FLOPs值较低，但其MAC较高，因此也应当尽可能减少逐元素操作。

以这4个规则为基础，可以看出ShuffleNet v1有3点违反了此规则：

- 在Bottleneck中使用了 $1 \times 1$ 组卷积与 $1 \times 1$ 的逐点卷积，导致输入输出通道数不同，违背了规则1与规则2。
- 整体网络中使用了大量的组卷积，造成了太多的分组，违背了规则3。
- 网络中存在大量的逐点相加操作，违背了规则4。

针对v1的问题，ShuffleNet v2提出了一种新的网络基本单元，具体如图7.13所示。图7.13a与图7.13b为v1版本的基础结构，图7.13c与图7.13d是v2提出的新结构。

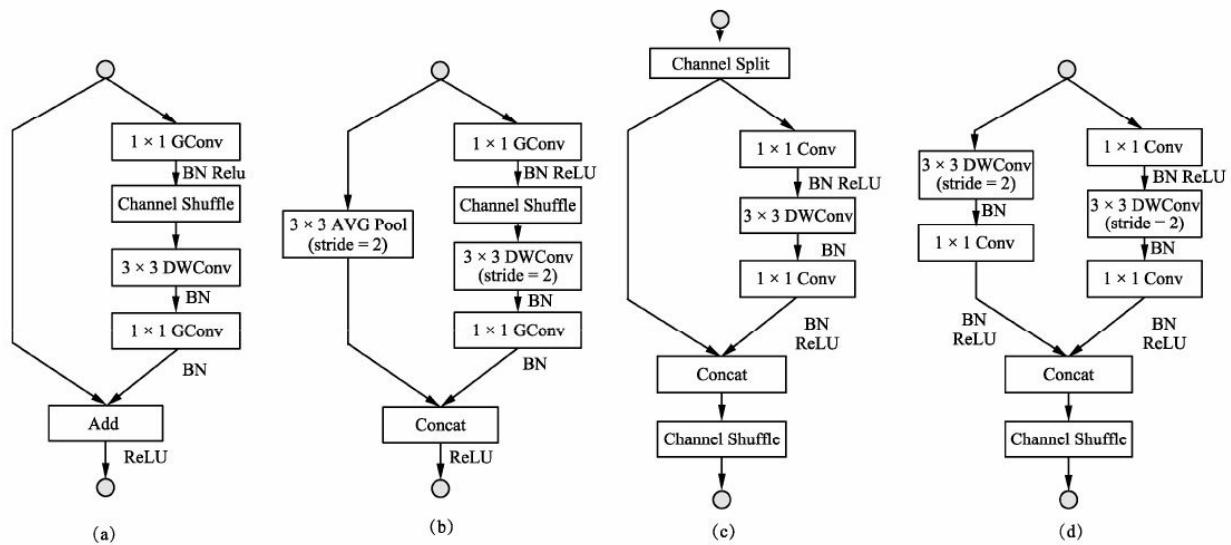


图7.13 ShuffleNet v2的基本结构单元

ShuffleNet v2的基本单元有如下3点新特性：

- 提出了一种新的Channel Split操作，如图7.13c所示，将输入特征分成两部分，一部分进行真正的深度可分离计算，将计算结果与另一部分进行通道Concat，最后进行通道的混洗操作，完成信息的互通。

·整个过程没有使用到 $1\times 1$ 组卷积，也避免了逐点相加的操作。

·在需要降采样与通道翻倍时，ShuffleNet v2去掉了Channel Split操作，这样最后Concat时通道数会翻倍，如图7.13d所示。

基于此基本单元，ShuffleNet v2的整体结构如表7.2所示。与ShuffleNet v1相比，ShuffleNet v2在全局平均池化之前增加了一个 $1\times 1$ 卷积来融合特征。依据输出特征的通道数多少，ShuffleNet v2也给出了多种配置，相应的模型精度、大小也会有所区别。

表7.2 ShuffleNet v2的网络整体结构

Layer	Outputsize	Ksize	Stride	Repeat	Output Channels			
					0.5x	1x	1.5x	2x
Image	$224\times 224$				3	3	3	3
Convl	$112\times 112$	$3\times 3$	2	1	24	24	24	24
Maxpool	$56\times 56$	$3\times 3$	2					
Stage2	$28\times 28$		2	1	48	116	176	244
	$28\times 28$		1	3				
Stage3	$14\times 14$		2	1	96	232	352	488
	$14\times 14$		1	7				
Stage4	$7\times 7$		2	1	192	464	704	976
	$7\times 7$		1	3				
Conv5	$7\times 7$	$1\times 1$	1	1	1024	1024	1024	2048
GlobalPool	$1\times 1$	$7\times 7$						
FC					1000	1000	1000	1000
FLOPS					41M	146M	299M	591M
# of Weights					1.4M	2.3M	3.5M	7.4M

ShuffleNet v2做了广泛的实验来验证该模型的优越性能，在此就不详细展开了。总体上，该模型从更直接影响模型运行速度的角度出发，在速度与精度的平衡上达到了当前的最佳水平，非常适合于移动端模型的应用。

## 7.4 总结

本章依次介绍了3种出色的轻量化网络：SqueezeNet、MobileNet及ShuffleNet。其中，SqueezeNet精心设计了一个压缩再扩展的模块，有效降低了卷积计算量；MobileNet v1及MobileNet v2则发挥了深度可分离卷积的优势，提升了卷积计算的效率。ShuffleNet更进一步，在分组卷积的思想上提出了通道混洗操作，避免了大量 $1\times 1$ 卷积的操作，可谓经典。通常情况下，将这几种轻量化网络应用到检测框架中，在速度上均可以得到不同程度的提升。

在前面几章的物体检测框架中，介绍了很多检测算法的细节处理，为了使读者能更充分地进行横向对比，在下一章中将会更加全面地介绍一些检测细节问题。

## 第8章 物体检测细节处理

在前面的章节中，我们对物体检测模型的思想、结构及实现有了一定的了解，但是要想获得较好的检测性能，检测算法的细节处理也极为重要。

在众多的细节处理中，非极大值抑制、样本的不均衡及模型的过拟合这3个问题尤为重要，对模型的性能影响极大。本章将依次对这3个问题进行详细的分析，并给出一些经典的解决方法。

## 8.1 非极大值抑制：NMS

当前的物体检测算法为了保证召回率，对于同一个真实物体往往会有多个候选框输出。由于多余的候选框会影响检测精度，因此需要利用NMS过滤掉重叠的候选框，得到最佳的预测输出。

NMS方法简单有效，并且对检测结果至关重要，在物体检测算法中有着广泛的应用。当前有几种较为常见的NMS方法，如图8.1所示，首先是**最为基本的NMS方法**，利用得分高的边框抑制得分低且重叠程度高的边框。然而基本的NMS存在一些缺陷，简单地过滤掉得分低且重叠度高的边框可能会导致漏检等问题。针对此问题陆续产生了一系列改进的方法，如Soft NMS、Softer NMS及IoU-Net等。

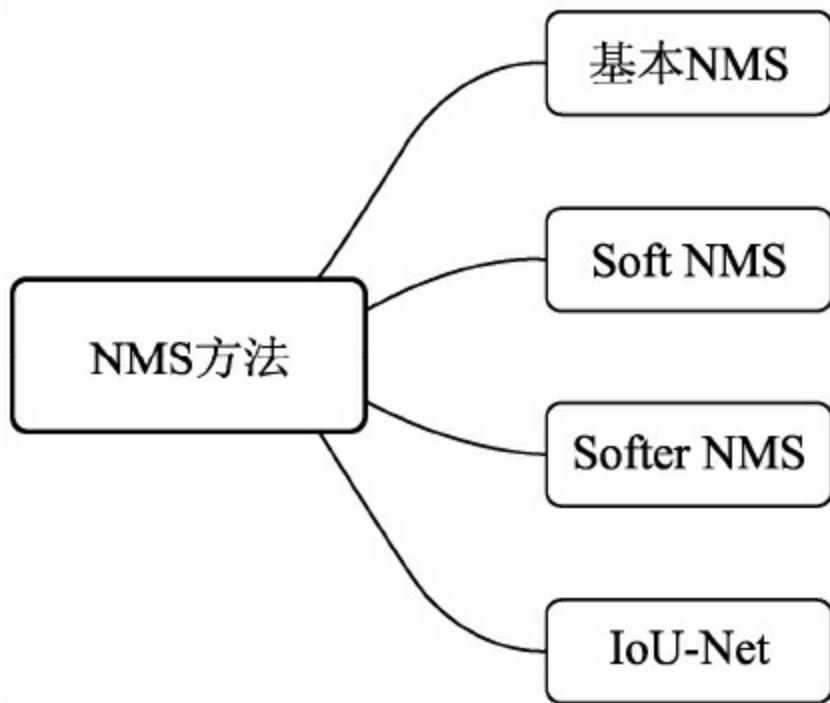


图8.1 NMS及其多种改进方法

基于上述背景，本节将首先介绍NMS的基本过程，然后依次讲解

Soft NMS、Softer NMS及IoU-Net的思想与实现方法。

### 8.1.1 NMS基本过程

为了保证物体检测的召回率，在Faster RCNN或者SSD网络的计算输出中，通常都会有不止一个候选框对应同一个真实物体。如图8.2左边图片中存在3个候选框，但是候选框A与C对应的是同一个物体，由于C的得分比A要低，在评测时，C候选框会被当做一个False Positive来看待，从而降低模型精度。实际上，由于候选框A的质量要比C好，理想的输出是A而不是C，我们希望能够抑制掉候选框C。

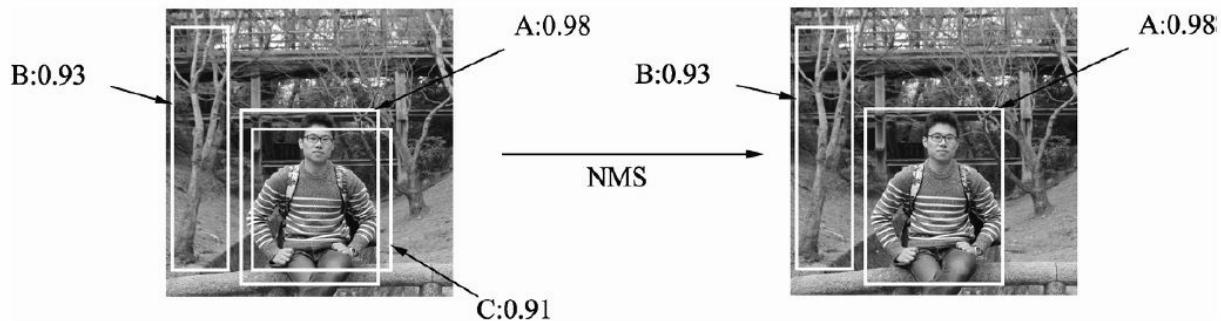


图8.2 NMS处理过程示意图

因此，物体检测网络通常在最后增加一个非极大值抑制操作，即NMS，将重复冗余的预测去掉，如图8.2右图所示。非极大值抑制，顾名思义就是抑制不是极大值的边框，这里的抑制通常是直接去掉冗余的边框。

这个过程涉及以下两个量化指标。

- 预测得分：**NMS假设一个边框的预测得分越高，这个框就要被优先考虑，其他与其重叠超过一定程度的边框要被舍弃，非极大值即是指得分的非极大值。

- IoU：**在评价两个边框的重合程度时，NMS使用了IoU这个指标。

如果两个边框的IoU超过一定阈值时，得分低的边框会被舍弃。阈值通常会取0.5或者0.7。

NMS存在一个非常简约的实现方法，算法输入包含了所有预测框的得分、左上点坐标、右下点坐标一共5个预测量，以及一个设定的IoU阈值。具体流程如下：

(1) 按照得分，对所有边框进行降序排列，记录下排列的索引order，并新建一个列表keep，作为最终筛选后的边框索引结果。

(2) 将排序后的第一个边框置为当前边框，并将其保留到keep中，再求当前边框与剩余所有框的IoU。

(3) 在order中，仅仅保留IoU小于设定阈值的索引，重复第(2)步，直到order中仅仅剩余一个边框，则将其保留到keep中，退出循环，NMS结束。

利用PyTorch，可以很方便地实现NMS模块。具体代码如下：

---

```
def nms(self, bboxes, scores, thresh=0.5):
    x1 = bboxes[:, 0]
    y1 = bboxes[:, 1]
    x2 = bboxes[:, 2]
    y2 = bboxes[:, 3]
    # 计算每个box的面积
    areas = (x2 - x1 + 1) * (y2 - y1 + 1)
    # 对得分进行降序排列，order为降序排列的索引
    _, order = scores.sort(0, descending=True)
    # keep保留了NMS留下的边框box
    keep = []
    while order.numel() > 0:
        if order.numel() == 1:                      # 保留框只剩一个
            i = order.item()
            keep.append(i)
            break
        else:
            i = order[0].item()                     # 保留scores最大的那个框box[i]
```

```
        keep.append(i)
    # 巧妙利用tensor.clamp函数求取每一个框与当前框的最大值和最小值
    xx1 = x1[order[1:]].clamp(min=x1[i])
    yy1 = y1[order[1:]].clamp(min=y1[i])
    xx2 = x2[order[1:]].clamp(max=x2[i])
    yy2 = y2[order[1:]].clamp(max=y2[i])
    # 求取每一个框与当前框的重合部分面积
    inter = (xx2-xx1).clamp(min=0) * (yy2-yy1).clamp(min=0)
    # 计算每一个框与当前框的IoU
    iou = inter / (areas[i]+areas[order[1:]]-inter)
    # 保留IoU小于阈值的边框索引
    idx = (iou <= threshold).nonzero().squeeze()
    if idx.numel() == 0:
        break
    # 这里的+1是为了补充idx与order之间的索引差
    order = order[idx+1]
# 返回保留下所有边框的索引值，类型为torch.LongTensor
return torch.LongTensor(keep)
```

---

NMS方法虽然简单有效，但在更高的物体检测需求下，也存在如下4个缺陷：

- 最大的问题就是将得分较低的边框强制性地去掉，如果物体出现较为密集时，本身属于两个物体的边框，其中得分较低的也有可能被抑制掉，从而降低了模型的召回率。

- 阈值难以确定。过高的阈值容易出现大量误检，而过低的阈值则容易降低模型的召回率，这个超参很难确定。

- 将得分作为衡量指标。NMS简单地将得分作为一个边框的置信度，但在一些情况下，得分高的边框不一定位置更准，因此这个衡量指标也有待考量。

- 速度：NMS的实现存在较多的循环步骤，GPU的并行化实现不是特别容易，尤其是预测框较多时，耗时较多。

### 8.1.2 抑制得分：Soft NMS

NMS方法虽有效过滤了重复框，但也容易将本属于两个物体框中得分低的框抑制掉，从而降低了召回率。以图8.3为例，假设模型对于当前图像给出了两个预测框，类别都为杯子，分类得分分别为0.99与0.94，两个框的IoU为0.6。

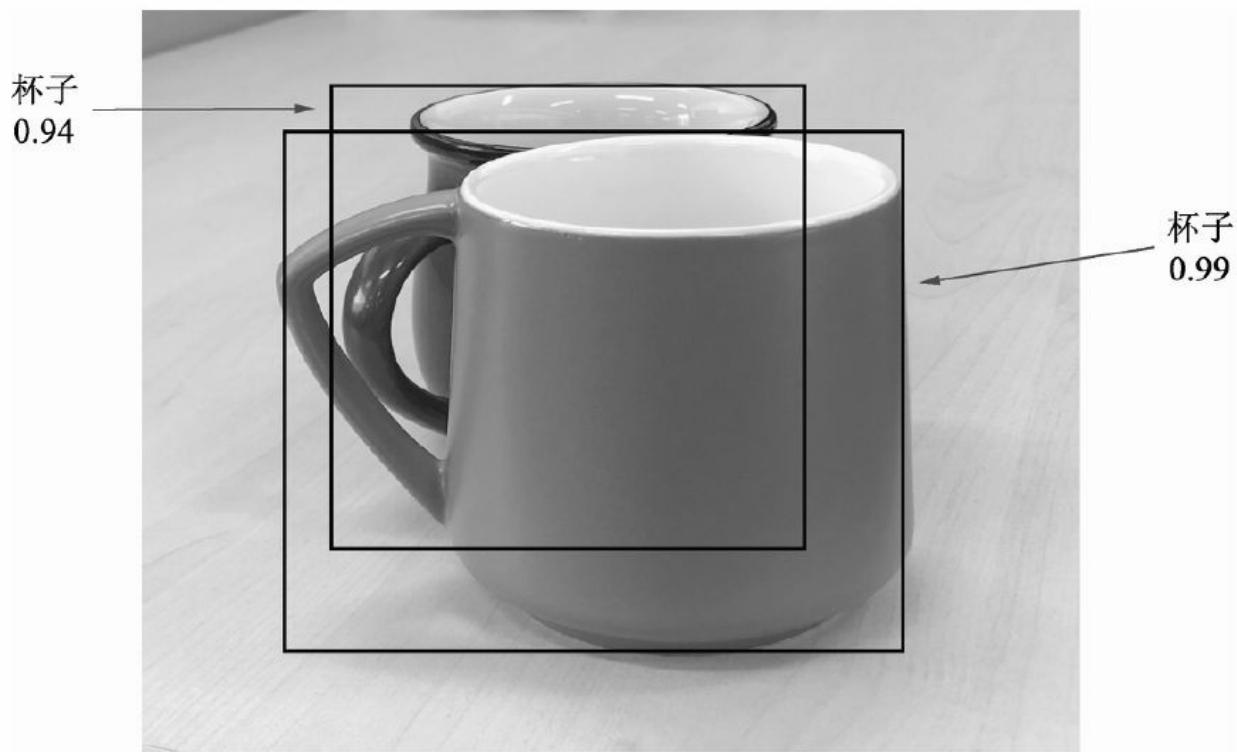


图8.3 NMS降低召回率的示例

按照NMS的方法，由于前面的杯子的分类得分高，并且IoU超过了阈值（假设为0.5），则后面的预测框会被抑制掉。但本身这两个框对应两个不同的真实杯子，因此这种NMS方法会导致漏检的现象。

造成这种现象的原因在于NMS的计算公式，如式（8-1）所示。

$$s_i = \begin{cases} s_i, & iou(M, b_i) < N_t \\ 0, & iou(M, b_i) \geq N_t \end{cases} \quad (8-1)$$

公式中 $s_i$ 代表了每个边框的得分， $M$ 为当前得分最高的框， $b_i$ 为剩余框的某一个， $N_t$ 为设定的阈值，可以看到当IoU大于 $N_t$ 时，该边框的得分直接置0，相当于被舍弃掉了，从而有可能造成边框的漏检。

基于此原因，诞生了Soft NMS方法，利用一行代码即改进了强硬的NMS方法。简而言之，Soft NMS对于IoU大于阈值的边框，没有将其得分直接置0，而是降低该边框的得分，具体方法如式（8-2）所示。

$$s_i = \begin{cases} s_i, & iou(M, b_i) < N_t \\ s_i(1 - iou(M, b_i)), & iou(M, b_i) \geq N_t \end{cases} \quad (8-2)$$

从公式中可以看出，利用边框的得分与IoU来确定新的边框得分，如果当前边框与边框 $M$ 的IoU超过设定阈值 $N_t$ 时，边框的得分呈线性的衰减。

但是，式（8-2）并不是一个连续的函数，当一个边框与 $M$ 的重叠IoU超过阈值 $N_t$ 时，其得分会发生跳变，这种跳变会对检测结果产生较大的波动，因此还需要寻找一个更为稳定、连续的得分重置函数，最终Soft NMS给出了如式（8-3）所示的重置函数。

$$s_i = \begin{cases} s_i e^{-\frac{iou(M, b_i)^2}{\sigma}}, & \forall b_i \notin D \end{cases} \quad (8-3)$$

采用这种得分衰减的方式，对于某些得分很高的边框来说，在后续的计算中还有可能被作为正确的检测框，而不像NMS那样“一棒子打

死”，因此可以有效提升模型的召回率。

Soft NMS的计算复杂度与NMS相同，是一种更为通用的非极大值抑制方法，可以将NMS看做Soft NMS的二值化特例。当然，Soft NMS也是一种贪心算法，并不能保证找到最优的得分重置映射。经过多种实验证明，Soft NMS在不影响前向速度的前提下，能够有效提升物体检测的精度。

### 8.1.3 加权平均：Softer NMS

NMS与Soft NMS算法都使用了预测分类置信度作为衡量指标，即假定分类置信度越高的边框，其位置也更为精准。但很多情况下并非如此，例如下面两种情形：

- 对于一个真实物体，所有的预测边框都不准，那么这时应该选择哪一个？还是综合所有边框得到更为精准的一个结果？
- 具有高分类置信度的边框其位置并不是最精准的。

因此，位置的置信度与分类置信度并不是强相关的关系，直接使用分类置信度作为NMS的衡量指标并非是最佳选择。基于此现象，Softer NMS进一步改进了NMS的方法，新增加了一个定位置信度的预测，使得高分类置信度的边框位置变得更加准确，从而有效提升了检测的性能。

首先，为了更加全面地描述边框预测，Softer NMS方法对预测边框与真实物体做了两个分布假设：

- 真实物体的分布是狄拉克delta分布，即标准方差为0的高斯分布的极限。
- 预测边框的分布满足高斯分布。

基于这两个假设，Softer NMS提出了一种基于KL（Kullback-Leibler）散度的边框回归损失函数KL loss。KL散度是用来衡量两个概率分布的非对称性衡量，KL散度越接近于0，则两个概率分布越相似。

具体到边框上，KL Loss是最小化预测边框的高斯分布与真实物体

的狄克拉分布之间的KL散度。即预测边框分布越接近于真实物体分布，损失越小。

为了描述边框的预测分布，除了预测位置之外，还需要预测边框的标准差，因此Softer NMS提出了如图8.4所示的预测结构。

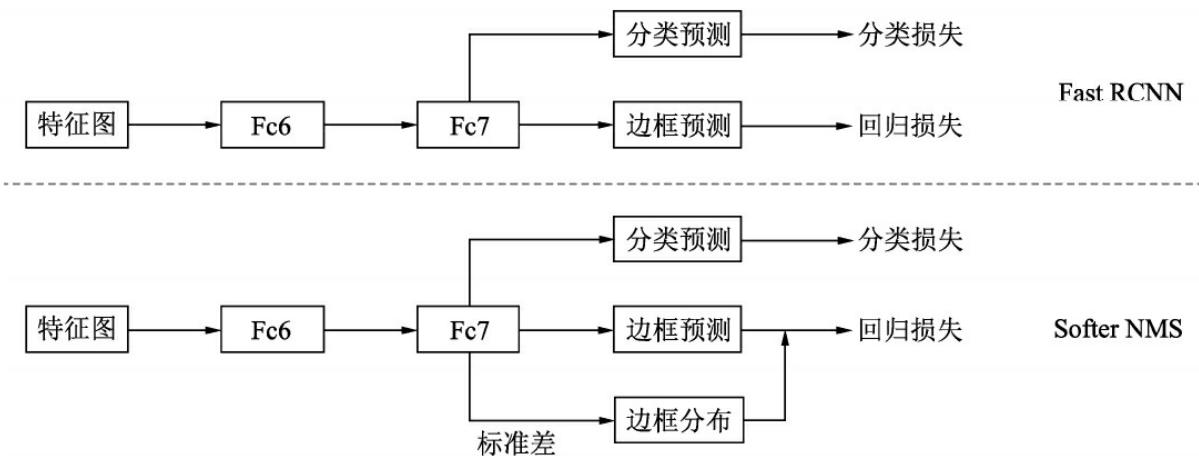


图8.4 Softer NMS网络结构图

图8.4中上半部为原始的Fast RCNN方法的预测，下半部的网络为Softer NMS提出的方法。可以看到，Softer NMS在原Fast RCNN预测的基础上，增加了一个标准差预测分支，从而形成边框的高斯分布，与边框的预测一起可以求得KL损失，由于公式较为复杂，这里就不再展开描述了。

边框的标准差可以被看做边框的位置置信度，因此Softer NMS利用该标准差也改善了NMS过程。具体过程大体与NMS相同，只不过利用标准差改善了高得分边框的位置坐标，从而使其更为精准。

举个例子，在NMS的某次循环中，假设当前边框为i，则Softer NMS会按照式（8-4）的方法更新边框i的坐标。

$$x1_i := \frac{\sum_j x1_j / \sigma_{x1,j}^2}{\sum_j 1 / \sigma_{x1,j}^2} \quad (8-4)$$

公式中j代表与i的IoU大于设定阈值的边框。可以看出，Softer NMS对于IoU大于设定阈值的边框坐标进行了加权平均，希望分类得分高的边框能够利用到周围边框的信息，从而提升其位置的准确度。

总体上，Softer NMS通过提出的KL Loss与加权平均的NMS策略，在多个数据集上有效提升了检测边框的位置精度。

### 8.1.4 定位置信度：IoU-Net

在当前的物体检测算法中，物体检测的分类与定位通常被两个分支预测。对于候选框的类别，模型给出了一个类别预测，可以作为分类置信度，然而对于定位而言，回归模块通常只预测了一个边框的转换系数，而缺失了定位的置信度，即框的位置准不准，并没有一个预测结果。

定位置信度的缺失也导致了在前面的NMS方法中，只能将分类的预测值作为边框排序的依据，然而在某些场景下，分类预测值高的边框不一定拥有与真实框最接近的位置，因此这种标准不平衡可能会导致更为准确的边框被抑制掉。

基于此，旷视提出了IoU-Net，增加了一个预测候选框与真实物体之间的IoU分支，并基于此改善了NMS过程，进一步提升了检测器的性能。

IoU-Net的整体结构如图8.5所示，基础架构与原始的Faster RCNN类似，使用了FPN方法作为基础特征提取模块，然后经过RoI的Pooling得到固定大小的特征图，利用全连接网络完成最后的多任务预测。

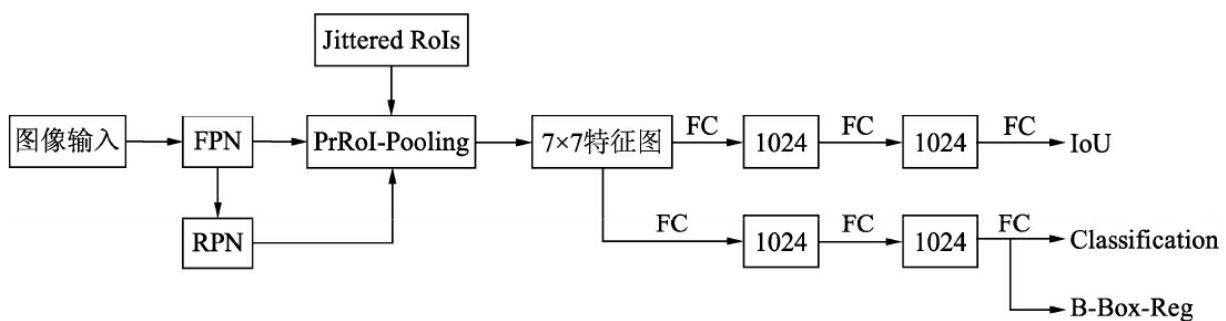


图8.5 IoU-Net网络结构图

同时，IoU-Net与Faster RCNN也有不同之处，主要有3点：

- 在Head处增加了一个IoU预测的分支，与分类回归分支并行。图8.5中的Jittered RoIs模块用于IoU分支的训练。
- 基于IoU分支的预测值，改善了NMS的处理过程。
- 提出了PrRoI-Pooling（Precise RoI Pooling）方法，进一步提升了感兴趣区域池化的精度。

下面对这3个主要的改进点进行详细的介绍。

## 1. IoU预测分支

IoU分支用于预测每一个候选框的定位置信度。需要注意的是，在训练时IoU-Net通过自动生成候选框的方式来训练IoU分支，而不是从RPN获取。

具体来讲，Jittered RoIs在训练集的真实物体框上增加随机扰动，生成了一系列候选框，并移除与真实物体框IoU小于0.5的边框。实验证明这种方法来训练IoU分支可以带来更高的性能与稳健性。

IoU分支也可以方便地集成到当前的物体检测算法中。在整个模型的联合训练时，IoU预测分支的训练数据需要从每一批的输入图像中单独生成。此外，还需要对IoU分支的标签进行归一化，保证其分布在[-1,1]区间中。

## 2. 基于定位置信度的NMS

由于IoU预测值可以作为边框定位的置信度，因此可以利用其来改善NMS过程。IoU-Net利用IoU的预测值作为边框排列的依据，并抑制掉与当前框IoU超过设定阈值的其他候选框。

此外，在NMS过程中，IoU-Net还做了置信度的聚类，即对于匹配到同一真实物体的边框，类别也需要拥有一致的预测值。具体做法是，在NMS过程中，当边框A抑制边框B时，通过式（8-5）来更新边框A的分类置信度。

$$S_A = \max(S_A, S_B) \quad (8-5)$$

### 3. PrRoI-Pooling方法

在第4章中详细介绍了RoI Align的方法，通过采样的方法有效避免了量化操作，减小了RoI Pooling的误差，如图8.5左图所示。但Align的方法也存在一个缺点，即对每一个区域都采取固定数量的采样点，但区域有大有小，都采取同一个数量点，显然不是最优的方法。

以此为出发点，IoU-Net提出了PrRoI Pooling方法，采用积分的方式实现了更为精准的兴趣区域池化，如图8.6中的右图所示。

与RoI Align只采样4个点不同，PrRoI Pooling方法将整个区域看做是连续的，采用如式（8-6）的积分公式求解每一个区域的池化输出值，区域内的每一个点 $(x,y)$ 都可以通过双线性插值的方法得到。这种方法还有一个好处是其反向传播是连续可导的，因此避免了任何的量化过程。

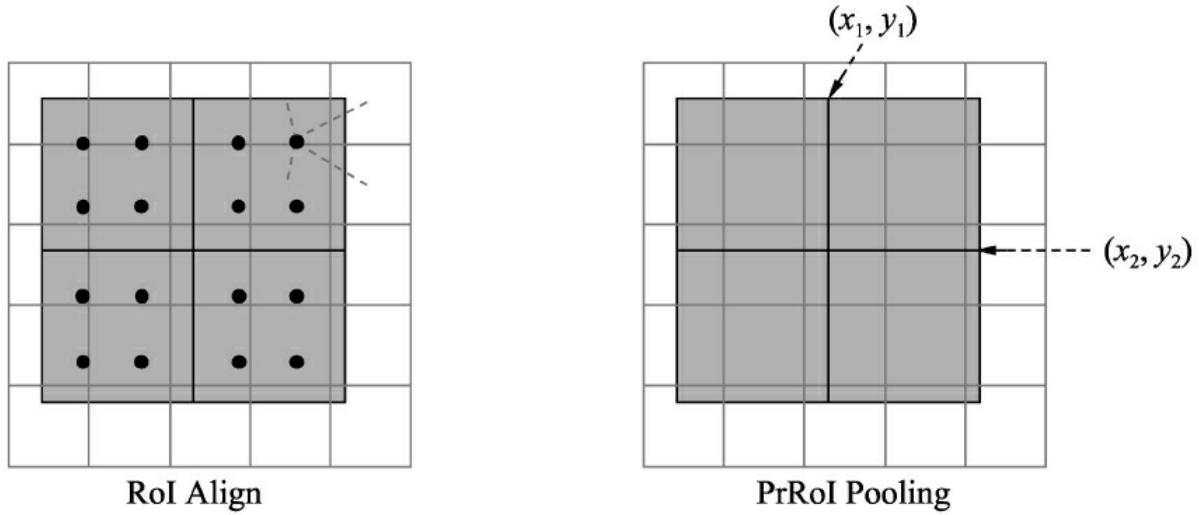


图8.6 PrRoI Pooling与RoI Align的比较图

$$PrPool(bin, F) = \frac{\int_{y_1}^{y_2} \int_{x_1}^{x_2} f(x, y) dx dy}{(x_2 - x_1) \times (y_2 - y_1)} \quad (8-6)$$

除了以上3点，IoU-Net还提出了一种优化的方法来解决模型最后边框位置的修正，在此就不展开叙述了。

总体上，IoU-Net提出了一个IoU的预测分支，解决了NMS过程中分类置信度与定位置信度之间的不一致，可以与当前的物体检测框架一起端到端地训练，在几乎不影响前向速度的前提下，有效提升了物体检测的精度。

## 8.2 样本不均衡问题

当前主流的物体检测算法，如Faster RCNN和SSD等，都是将物体检测当做分类问题来考虑，即先使用先验框或者RPN等生成感兴趣的区域，再对该区域进行分类与回归位置。这种基于分类思想的物体检测算法存在样本不均衡的问题，因而会降低模型的训练效率与检测精度。

本节首先分析样本不均衡带来的问题，随后会讲解两种经典的缓解不均衡问题的方法。

### 8.2.1 不均衡问题分析

在当前的物体检测算法中，由于检测算法各不相同，以及数据集之间的差异，可能会存在正负样本、难易样本、类别间样本这3种不均衡问题，如图8.7所示。本节将详细分析这3种不均衡问题的来源，以及常用的解决方法。

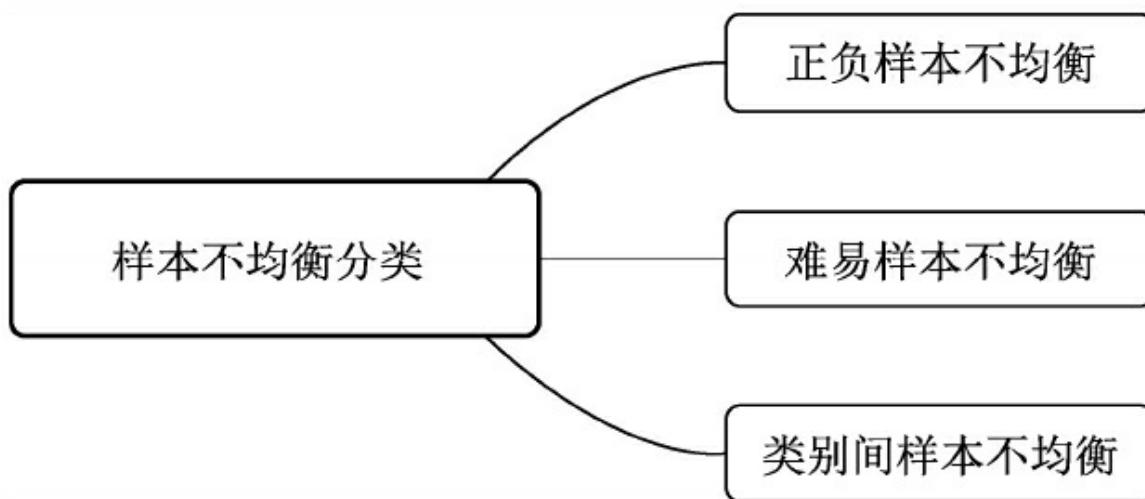


图8.7 3种样本不均衡问题

#### 1. 正负样本不均衡

以Faster RCNN为例，在RPN部分会生成20000个左右的Anchor，由于一张图中通常有10个左右的物体，导致可能只有100个左右的Anchor会是正样本，正负样本比例约为1：200，存在严重的不均衡。

对于物体检测算法，有核心价值的是对应着真实物体的正样本，在训练时会根据其loss来调整网络参数。相比之下，负样本对应着图像的背景，如果有大量的负样本参与训练，则会淹没正样本的损失，从而降低网络收敛的效率与检测精度。

## 2. 难易样本不均衡

除了正负样本，在物体检测中还存在着难易样本的不均衡问题。根据是否容易学习及与标签的重叠程度，可以将所有样本分为4类：简单正样本（Easy Positive）、难正样本（Hard Positive）、简单负样本（Easy Negative）及难负样本（Hard Negative），如图8.8所示。

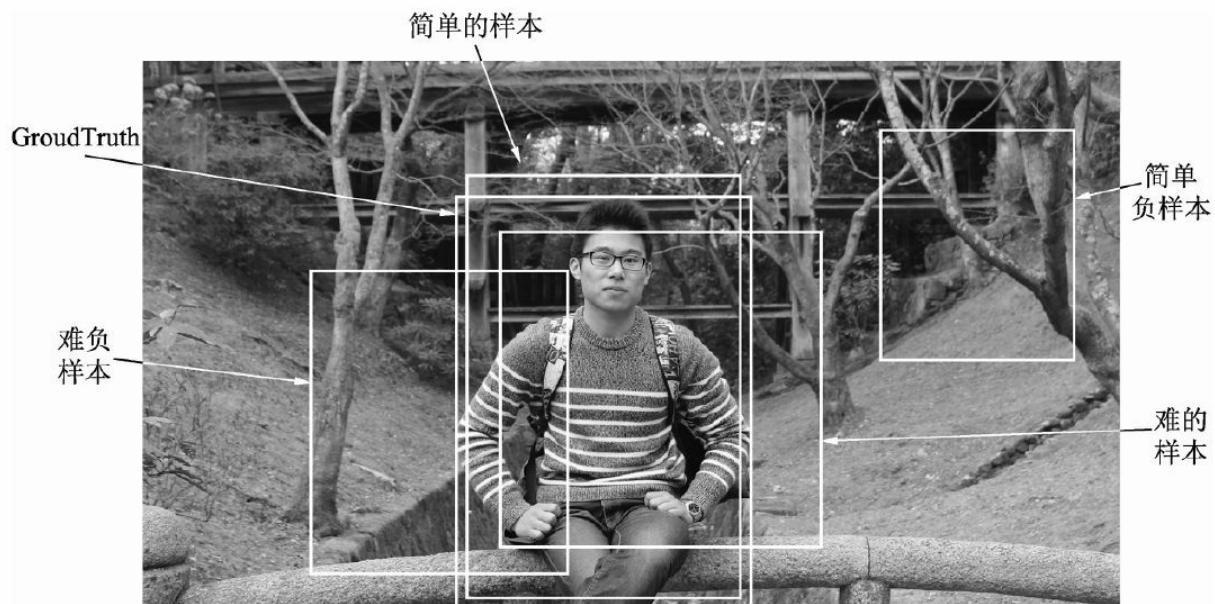


图8.8 各类样本示意图

难样本指的是分类不太明确的边框，处在前景与背景的过渡区域上，在网络训练中难样本损失会较大，也是我们希望模型去学习优化的样本，利用这部分训练可以提升检测的准确率。

然而，大量的样本并非处在前景与背景的过渡区，而是与真实物体没有重叠区域的负样本，或者与真实物体重叠程度很高的正样本，这部分被称为简单样本，单个损失会较小，对参数收敛的作用有限。虽然简单样本单个损失小，但由于数量众多，因此如果全都计算损失的话，其损失也会比难样本大很多，这种难易样本的不均衡也会影响模型的收敛与精度。

值得注意的是，由于负样本中大量的简单样本，导致难易样本与正负样本这两个不均衡问题有一定的重叠，解决方法往往能同时对这两个问题起作用。

### 3. 类别间样本不均衡

在有些物体检测的数据集中，还会存在类别间的不均衡问题。举个例子，数据集中有100万个车辆、1000个行人的实例标签，样本比例为1000：1，属于典型的类别不均衡。

这种情况下，如果不做任何处理，使用该数据集进行训练，由于行人这一类别可参考标签太少，会使得模型主要关注车这一类别的检测，网络中的参数主要根据车辆的损失进行优化，导致行人的检测精度大大下降。

针对以上3种不均衡问题，经典的物体检测算法在处理样本时，总体上有如下4种缓解办法：

- Faster RCNN、SSD等算法在正负样本的筛选时，根据样本与真实物体的IoU大小，设置了3：1的正负样本比例，这一点缓解了正负样本的不均衡，同时也对难易样本不均衡起到了作用。

- Faster RCNN在RPN模块中，通过前景得分排序筛选出了2000个左右的候选框，这也会将大量的负样本与简单样本过滤掉，缓解了前两个不均衡问题。

- 权重惩罚：对于难易样本与类别间的不均衡，可以增大难样本与少类别的损失权重，从而增大模型对这些样本的惩罚，缓解不均衡问题。

- 数据增强：从数据侧入手，可以在当前数据集上使用随机生成和

添加扰动的方法，也可以利用网络爬虫数据等增加数据集的丰富性，从而缓解难易样本和类别间样本等不均衡问题，可以参考SSD的数据增强方法。

## 8.2.2 在线难样本挖掘：OHEM

针对难易样本不均衡的问题，2016年CVPR会议上的OHEM（Online Hard Example Mining）方法高效率地实现了在线难样本的挖掘，在多个数据集上都有着优越的表现，是一个很经典的难样本挖掘方法。

难样本挖掘的思想最初在机器学习中被广泛使用，一般被称为难负样本挖掘（Hard Negative Mining, HNM），用于解决类别的不平衡问题。以SVMs（Support Vector Machines, 支持向量机）为例，HNM方法先让模型收敛于当前的工作数据集，然后固定该模型，在数据集中去除简单的样本，添加一些当前无法判断的样本，进行新的训练。这样的交替训练可以使得模型性能达到最优。

物体检测方法很难直接使用HNM算法进行挖掘。原因在于物体检测算法通常采用随机梯度下降（Stochastic Gradient Descent, SGD）等优化方法来进行优化，往往需要上万次的参数更新；而如果采用HNM交替训练的方法，每迭代几次就固定模型，训练的速度会大大下降。

OHEM可以看做是HNM在物体检测算法上的应用，在实现时选择了Fast RCNN作为基础检测算法。Fast RCNN与Faster RCNN类似，采用了两阶结构，在第二个阶段通过RCNN网络得到了边框的预测值，接下来使用了如下3点标准来确定正、负样本。

- 当前RoI与真实物体的IoU大于0.5时，判定为正样本。
- 当前RoI与真实物体的IoU大于0且小于0.5时，判定为负样本。
- 为了均衡正、负样本的数量，控制正、负样本的比例为1：3，总

数量不超过256。通过这种方式有效缓解了正、负样本的不均衡。

上述方法虽然简单有效，但是容易忽略一些较为重要的难负样本，并且固定了正、负样本的比例与最大数量，显然不是最优的选择。以此为出发点，OHEM将交替训练与SGD优化方法进行了结合，在每张图片的RoI中选择了较难的样本，实现了在线的难样本挖掘。

OHEM实现在线难样本挖掘的网络如图8.9所示。图中包含了两个相同的RCNN网络，上半部的a部分是只可读的网络，只进行前向运算；下半部的b网络即可读也可写，需要完成前向计算与反向传播。

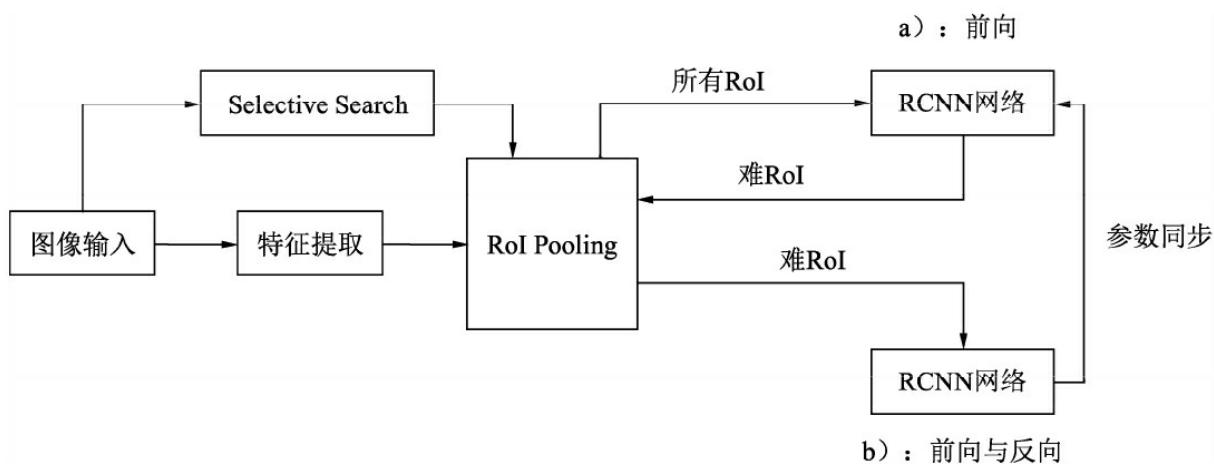


图8.9 OHEM方法示意图

在一个batch的训练中，基于Fast RCNN的OHEM算法可以分为以下5步：

(1) 按照原始Fast RCNN算法，经过卷积提取网络与RoI Pooling得到了每一张图像的RoI。

(2) 上半部的a网络对所有的RoI进行前向计算，得到每一个RoI的损失。

(3) 对RoI的损失进行排序，进行一步NMS操作，以去除掉重叠严重的RoI，并在筛选后的RoI中选择出固定数量损失较大的部分，作为难样本。

(4) 将筛选出的难样本输入到可读写的b网络中，进行前向计算，得到损失。

(5) 利用b网络得到的反向传播更新网络，并将更新后的参数与上半部的a网络同步，完成一次迭代。

当然，为了实现方便，OHEM也可以仅采用一个RCNN网络，在选择完难样本后将剩下的简单样本的损失置0，可以起到相同的作用。但是，由于其特殊的损失计算方式，把简单的样本都舍弃了，导致模型无法提升对于简单样本的检测精度，这也是OHEM方法的一个弊端。

总体上，OHEM是一个很经典的难样本挖掘Trick，实现方式简单，可以显著提升网络训练的效率和检测性能，被广泛地应用于难样本的挖掘场景中，并且数据集越大、难度越高，OHEM对于检测的提升越明显。

### 8.2.3 专注难样本：Focal Loss

当前一阶的物体检测算法，如SSD和YOLO等虽然实现了实时的速度，但精度始终无法与两阶的Faster RCNN相比。是什么阻碍了一阶算法的高精度呢？何凯明等人将其归咎于正、负样本的不均衡，并基于此提出了新的损失函数Focal Loss及网络结构RetinaNet，在与同期一阶网络速度相同的前提下，其检测精度比同期最优的二阶网络还要高。

从前面的叙述中可以得知，Faster RCNN在第一个阶段利用得分筛选出了2000个左右的RoI，可以过滤掉大部分的负样本，在第二个阶段通过固定正、负样本比例或者OHEM等方法，可以有效解决正、负样本的不均衡问题。

而对于SSD等一阶网络，由于其需要直接从所有的预选框中进行筛选，即使使用了固定正、负样本比例的方法，仍然效率低下，简单的负样本仍然占据主要地位，导致其精度不如两阶网络。

为了解决一阶网络中样本的不均衡问题，何凯明等人首先改善了分类过程中的交叉熵函数，提出了可以动态调整权重的Focal Loss。为了形成对比，接下来分别介绍标准交叉熵、平衡交叉熵及Focal Loss。

#### 1. 标准交叉熵损失

首先回顾一下标准的交叉熵（Cross Entropy, CE）函数，其形式如式（8-7）所示。

$$CE(p, y) = \begin{cases} -\log(p) & \text{if } (y=1) \\ -\log(1-p) & \text{otherwise} \end{cases} \quad (8-7)$$

公式中， $p$ 代表样本在该类别的预测概率， $y$ 代表样本标签。可以看出，当标签为1时， $p$ 越接近1，则损失越小；标签为0时 $p$ 越接近0，则损失越小，符合优化的方向。

为了方便表示，按照式（8-8）将 $p$ 标记为 $p_t$ ：

$$p_t = \begin{cases} p & \text{if } (y=1) \\ 1-p & \text{otherwise} \end{cases} \quad (8-8)$$

则交叉熵可以表示为式（8-9）的形式：

$$\text{CE}(p,y)=\text{CE}(p_t)=-\log(p_t) \quad (8-9)$$

标准的交叉熵中所有样本的权重都是相同的，因此如果正、负样本不均衡，大量简单的负样本会占据主导地位，少量的难样本与正样本会起不到作用，导致精度变差。

## 2. 平衡交叉熵损失

为了改善样本的不平衡问题，平衡交叉熵在标准的基础上增加了一个系数 $\alpha_t$ 来平衡正、负样本的权重， $\alpha_t$ 由超参 $\alpha$ 按照式（8-10）计算得来， $\alpha$ 取值在[0,1]区间内。

$$\alpha_t = \begin{cases} \alpha & \text{if } (y=1) \\ 1-\alpha & \text{otherwise} \end{cases} \quad (8-10)$$

有了 $\alpha_t$ ，平衡交叉熵损失公式如式（8-11）所示。

$$\text{CE}(p_t)=-\alpha_t \log(p_t) \quad (8-11)$$

尽管平衡交叉熵损失改善了正、负样本间的不平衡，但由于其缺乏对难易样本的区分，因此没有办法控制难易样本之间的不均衡。

### 3. 专注难样本Focal Loss

Focal Loss为了同时调节正、负样本与难易样本，提出了如式（8-12）所示的损失函数。

$$FL(p_t) = -\alpha_t(1-p_t)^\gamma \log(p_t) \quad (8-12)$$

对于该损失函数，有如下3个属性：

- 与平衡交叉熵类似，引入了 $\alpha_t$ 权重，为了改善正负样本的不均衡，可以提升一些精度。

- $(1-p_t)^\gamma$ 是为了调节难易样本的权重。当一个边框被误分类时， $p_t$ 较小，则 $(1-p_t)^\gamma$ 接近于1，其损失几乎不受影响；当 $p_t$ 接近于1时，表明其分类预测较好，是简单样本， $(1-p_t)^\gamma$ 接近于0，因此其损失被调低了。

- $\gamma$ 是一个调制因子， $\gamma$ 越大，简单样本损失的贡献会越低。

为了验证Focal Loss的效果，何凯明等人还提出了一个一阶物体检测结构RetinaNet，其结构如图8.10所示。

对于RetinaNet的网络结构，有以下5个细节：

- 在Backbone部分，RetinaNet利用ResNet与FPN构建了一个多尺度特征的特征金字塔。

- RetinaNet使用了类似于Anchor的预选框，在每一个金字塔层，使用了9个大小不同的预选框。

·分类子网络：分类子网络为每一个预选框预测其类别，因此其输出特征大小为 $K \times W \times H$ ，A默认为9，K代表类别数。中间使用全卷积网络与ReLU激活函数，最后利用Sigmoid函数输出预测值。

·回归子网络：回归子网络与分类子网络平行，预测每一个预选框的偏移量，最终输出特征大小为 $4A \times W \times W$ 。与当前主流工作不同的是，两个子网络没有权重的共享。

·Focal Loss：与OHEM等方法不同，Focal Loss在训练时作用到所有的预选框上。对于两个超参数，通常来讲，当 $\gamma$ 增大时， $\alpha$ 应当适当减小。实验中 $\gamma$ 取2、 $\alpha$ 取0.25时效果最好。

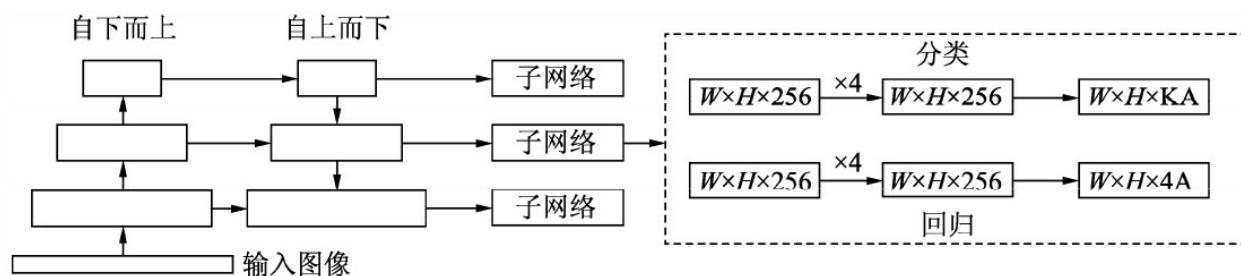


图8.10 RetinaNet网络结构图

下面基于FPN50的基础网络，利用PyTorch来搭建一个RetinaNet。代码如下：

```
class RetinaNet(nn.Module):
    num_anchors = 9                                     # 默认使用9个Anchors
    def __init__(self, num_classes=20):
        super(RetinaNet, self).__init__()
        self.fpn = FPN50()                               # 引入FPN50网络，输出是大小不一的特征图
        self.num_classes = num_classes                   # 默认做20类检测
        # 定位分支的输出是每个Anchor的4个边框位置
        self.loc_head = self._make_head(self.num_anchors*4)
        # 分类分支的输出是每个Anchor的20个类别的得分
        self.cls_head = self._make_head(self.num_anchors*self.num_classes)
    def forward(self, x):
        fms = self.fpn(x)                             # 首先获取Backbone的输出
        loc_preds = []
```

```

cls_preds = []
for fm in fms:
    # 对每个输出的特征图进行定位与分类分支的计算
    loc_pred = self.loc_head(fm)
    cls_pred = self.cls_head(fm)
    # 通道维度变换, 从[N, 36,H,W]转换为[N,H*W*9, 4]
    loc_pred=loc_pred.permute(0,2,3,1).contiguous().view(x.size(0),-1,4)
    # 通道维度变换, 从[N,180,H,W]转换为[N,H*W*9,20]
    cls_pred=cls_pred.permute(0,2,3,1).contiguous().view(x.size(0),
-1,self.num_classes)
    # 将定位与分类分支的输出进行拼接, 作为最终RetinaNet的输出
    loc_preds.append(loc_pred)
    cls_preds.append(cls_pred)
return torch.cat(loc_preds,1), torch.cat(cls_preds,1)

# 搭建分类与定位的分支
def _make_head(self, out_planes):
    layers = []
    for _ in range(4):
        layers.append(nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1))
        layers.append(nn.ReLU(True))
    layers.append(nn.Conv2d(256, out_planes, kernel_size=3, stride=1,
padding=1))
    return nn.Sequential(*layers)

# 由于batch较小, RetinaNet冻结了BN层, 不参与训练, 这一点需要注意
def freeze_bn(self):
    for layer in self.modules():
        if isinstance(layer, nn.BatchNorm2d):
            layer.eval()

```

---

总体上, Focal Loss算法将样本的不均衡视为一阶与两阶网络精度差距的原因, 提出了简单高效的Focal Loss, 使得模型专注于难样本上, 并提出了一阶网络RetinaNet, 实现了SOTA的精度。

### 8.3 模型过拟合

对于物体检测算法来说，模型过拟合是经常发生的一个现象。如图8.11所示为在训练过程中训练集与验证集的误差变化。在训练初期，验证集的误差是随着训练集的误差下降而下降的，这时候模型在充分学习数据的分布空间，对数据拟合得越来越好。

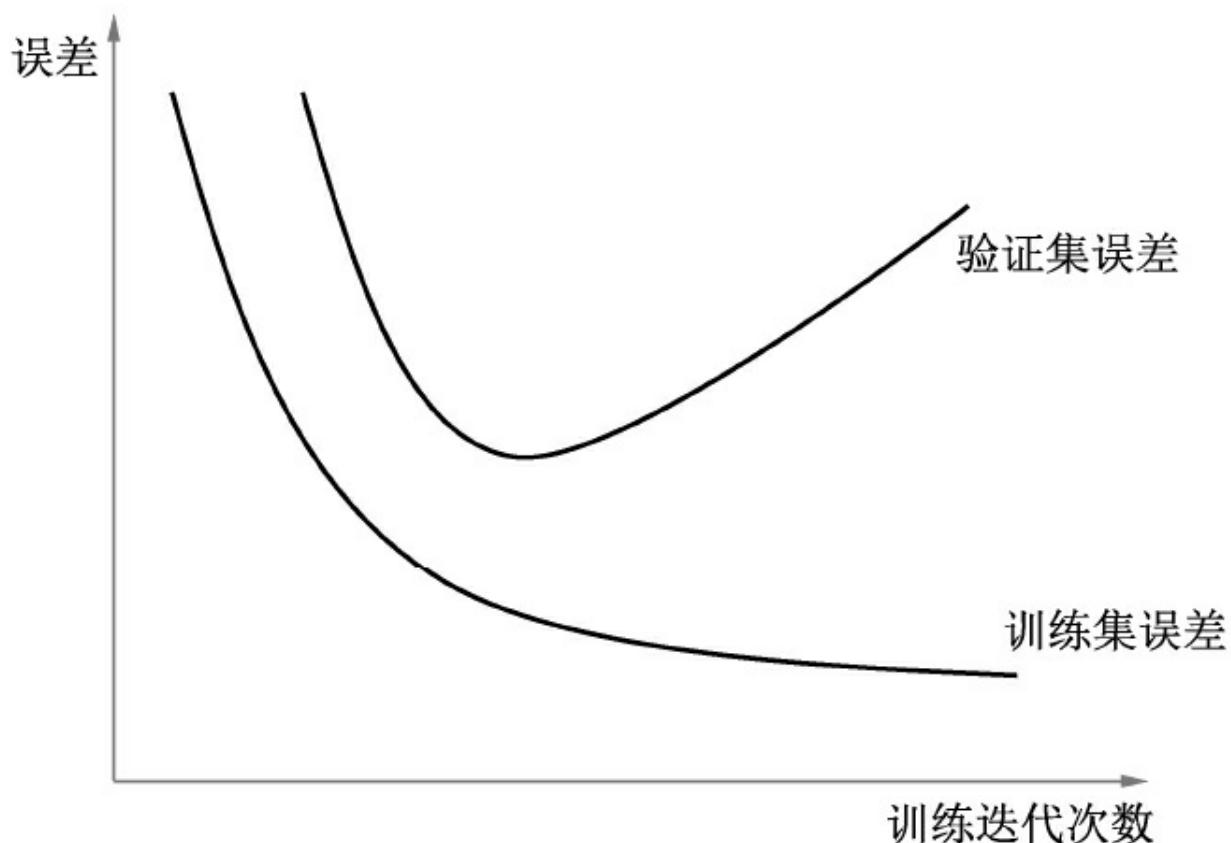


图8.11 验证集与训练集误差示意图

然而，当训练次数超出一定步数后，训练集的误差虽然仍在下降，但验证集的误差却在逐渐上升，此时就发生了过拟合现象。

通常，训练集的数据量有限，只是全局数据的子集，数据分布总是

会与全局数据存在偏差。当训练超过一定程度后，模型对数据分布拟合地很充分了，训练数据的轻微扰动都会导致模型发生显著变化。因此，过拟合现象的本质其实是模型学习到了训练数据自身的特性，非全局的特性。

从深度学习诞生至今，解决模型过拟合的脚步就从未停止过。如图8.12所示为当前几种主流的防止模型过拟合的方法。

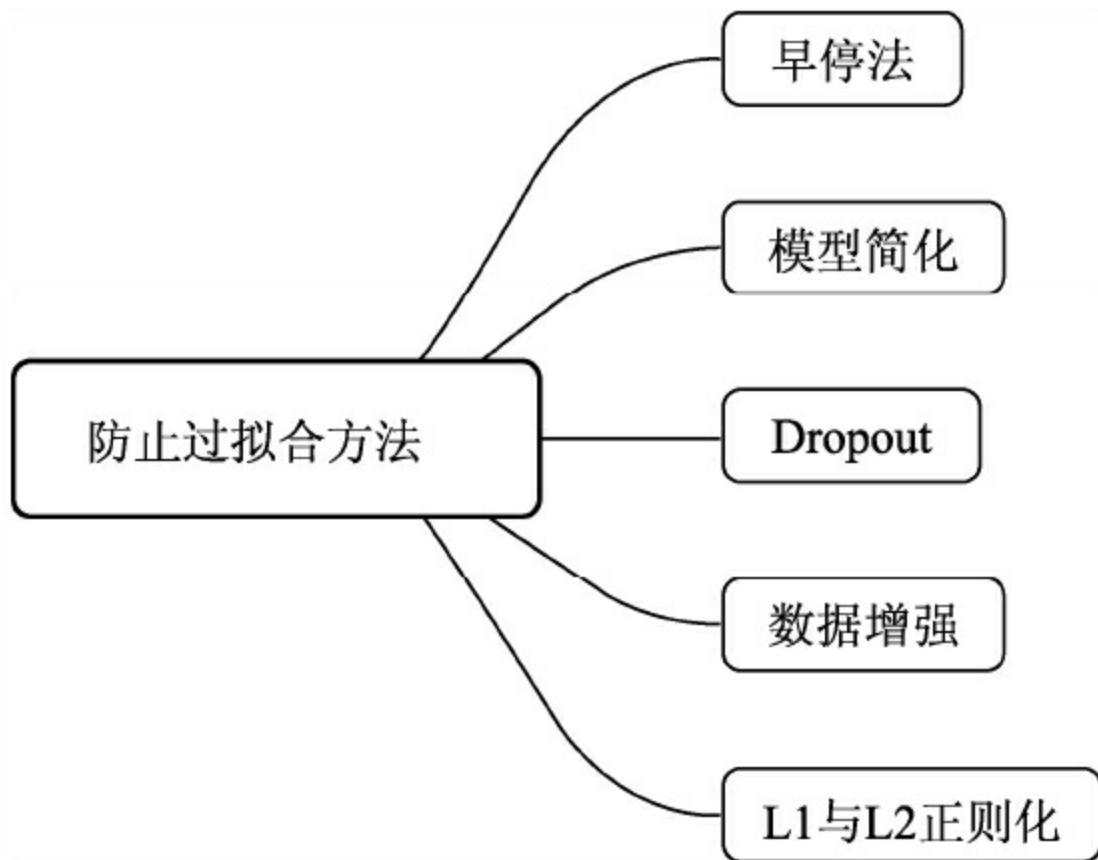


图8.12 防止过拟合方法

由于数据增强与正则化两种手段较为重要，在后面会单独介绍。在此先介绍与分析其他3种方法：

·早停法：在训练时，每隔一段时间就在验证集上做评估，当验证集误差出现增大时则停止训练，这是最实际有效的一个防止过拟合的方

法。

·**简化模型**: 当数据不够充分时, 如果模型的拟合能力过强, 也容易陷入过拟合。因此, 可以在不太影响模型精度的前提下, 通过减少模型的深度等方法, 适当地简化模型结构。

·**Dropout**: 在第3章中已有详细介绍, 其思想主要是在训练阶段以概率 $p$ 保留每个神经元, 以 $1-p$ 的概率丢弃该神经元。Dropout也属于正则化方法的一种, 通过这种方式可以使模型不强依赖于某些神经元, 因而增加其泛化能力, 减少过拟合, 一般应用于全连接网络。

### 8.3.1 数据增强 ---

数据是检测模型的血液，在实际工程中我们会发现，虽然通过使用各种技巧可以减缓网络的过拟合，但都不如增加数据的丰富性效果突出。实际中我们能获得的数据通常是有限的，因此可以通过数据增强的手段来扩展数据集。

SSD中的数据预处理部分是一个经典的数据增强过程。通常来讲，我们可以从4个方面去尝试进行数据增强，如图8.13所示。

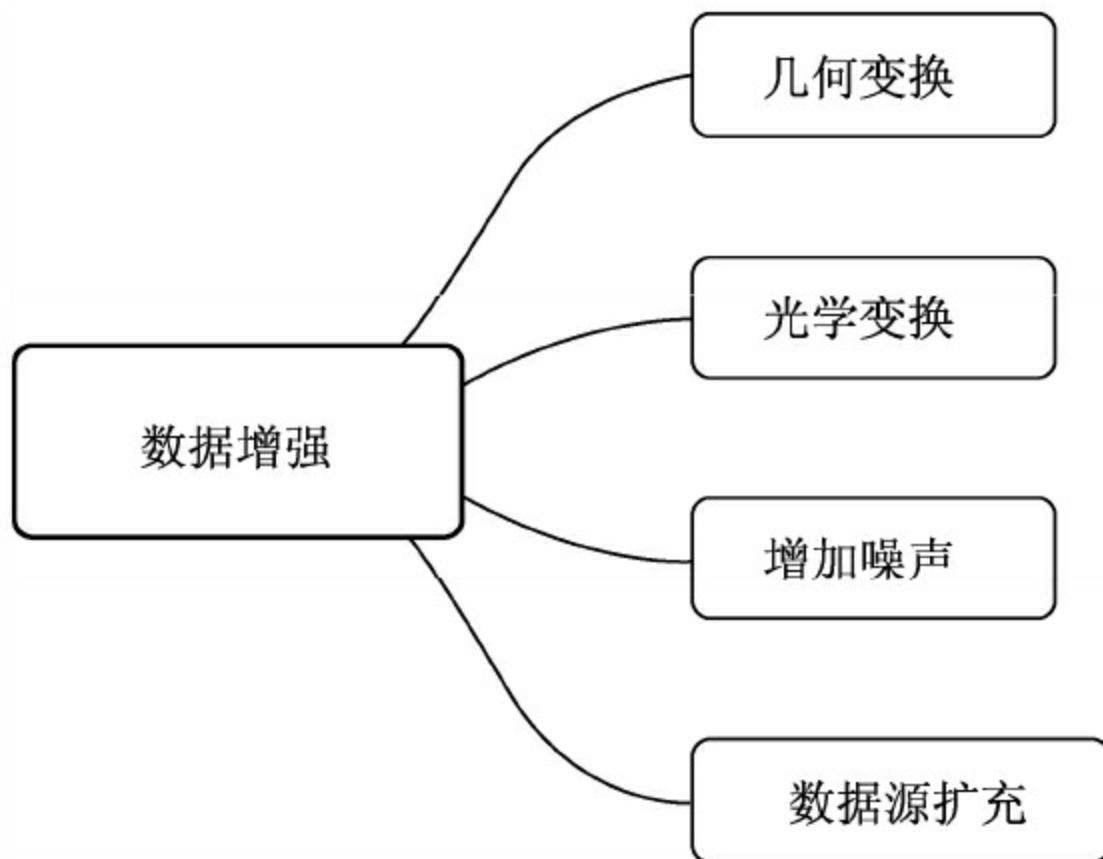


图8.13 数据增强的多种手段

下面对几种增强方式具体说明如下：

·**几何变换**: 可以丰富物体在图像中出现的位置和尺度等, 从而满足模型的平移不变性与尺度不变性, 例如平移、翻转、缩放和裁剪等操作。尤其是水平翻转 $180^\circ$ , 在多个物体检测算法中都有使用, 效果很好。

·**光学变换**: 可以增加不同光照和场景下的图像, 典型操作有亮度、对比度、色相与饱和度的随机扰动、通道色域之间的交换等。

·**增加噪声**: 通过在原始图像上增加一定的扰动, 如高斯噪声, 可以使模型对可能遇到的噪声等自然扰动产生鲁棒性, 从而提升模型的泛化能力。需要注意噪声不能过大, 以免影响模型的输出。

·**数据源头**: 有时为了扩充数据集, 可以将检测物体与其他背景图像融合, 通过替换物体背景的方式来增加数据集的丰富性。

数据增强不仅可以防止模型的过拟合, 对于模型的检测性能也通常会有较大的提升。

### 8.3.2 L1与L2正则化

正则化通常是指对模型参数进行限制，增加模型的泛化能力，是一个非常有效的防止模型过拟合的方法。可以从奥卡姆剃刀原理的角度去解释，即在所有可以选择的模型中，能够很好拟合当前数据，同时又十分简单的模型才是最好的。

L1与L2正则是最常用的两种正则化手段，主要是通过在损失函数中增加一项对网络参数的约束，使得参数渐渐变小，模型趋于简单，以防止过拟合。

L1正则化的损失函数如式（8-13）所示。

$$Loss_{L1} = Loss + \lambda \sum \|\omega\| \quad (8-13)$$

公式中， $\omega$ 代表网络中的参数，超参数 $\lambda$ 需要人为指定。需要注意的是，L1使用绝对值来约束参数，导致其在0点不可微分，这种情况下参数 $\omega$ 很有可能最终被约束为0。

在此举一个直观的例子，如图8.14所示，图中方块代表L1正则下的参数限制空间，两个参数的绝对值和如果相同，呈现出的形状正是一个方块。模型优化与训练其实是在优化空间与限制空间的参数当中，寻找最优参数值的过程。

从图8.14中可以看出，优化空间与参数限制空间有很大的概率相交于 $\omega_1$ 与 $\omega_2$ 的坐标轴上，即使扩展到更高的参数维度，L1的参数限制空间始终存在尖锐的凸点，因此L1正则化会导致参数的稀疏化。如果需要做模型的压缩，L1正则是一个不错的选择。

相比于L1正则化，L2正则化则使用了平方函数来约束网络参数，其损失函数如式（8-14）所示。

$$Loss_{L2} = Loss + \lambda \sum \|\omega\|^2 \quad (8-14)$$

如图8.15所示，由于L2正则化使用了平方函数，而如果两个参数的平方和相同，呈现出的形状会是一个圆，与优化空间的交点在参数0点的概率很低。因此，L2正则化可以使参数尽可能地小，但不至于为0，这样既保留了模型的拟合能力，同时也增加了泛化能力，因此L2一般情况下更为常用。

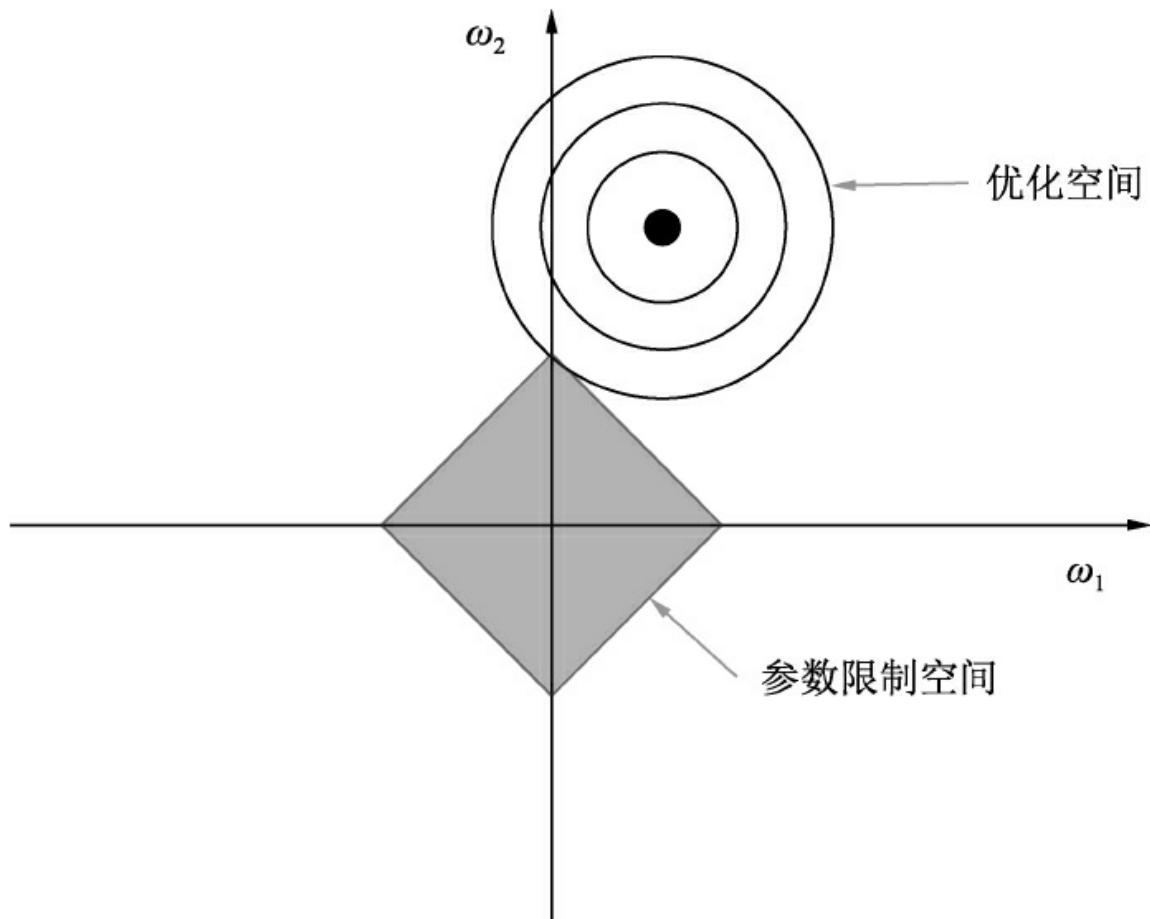


图8.14 L1正则化的参数空间与优化空间

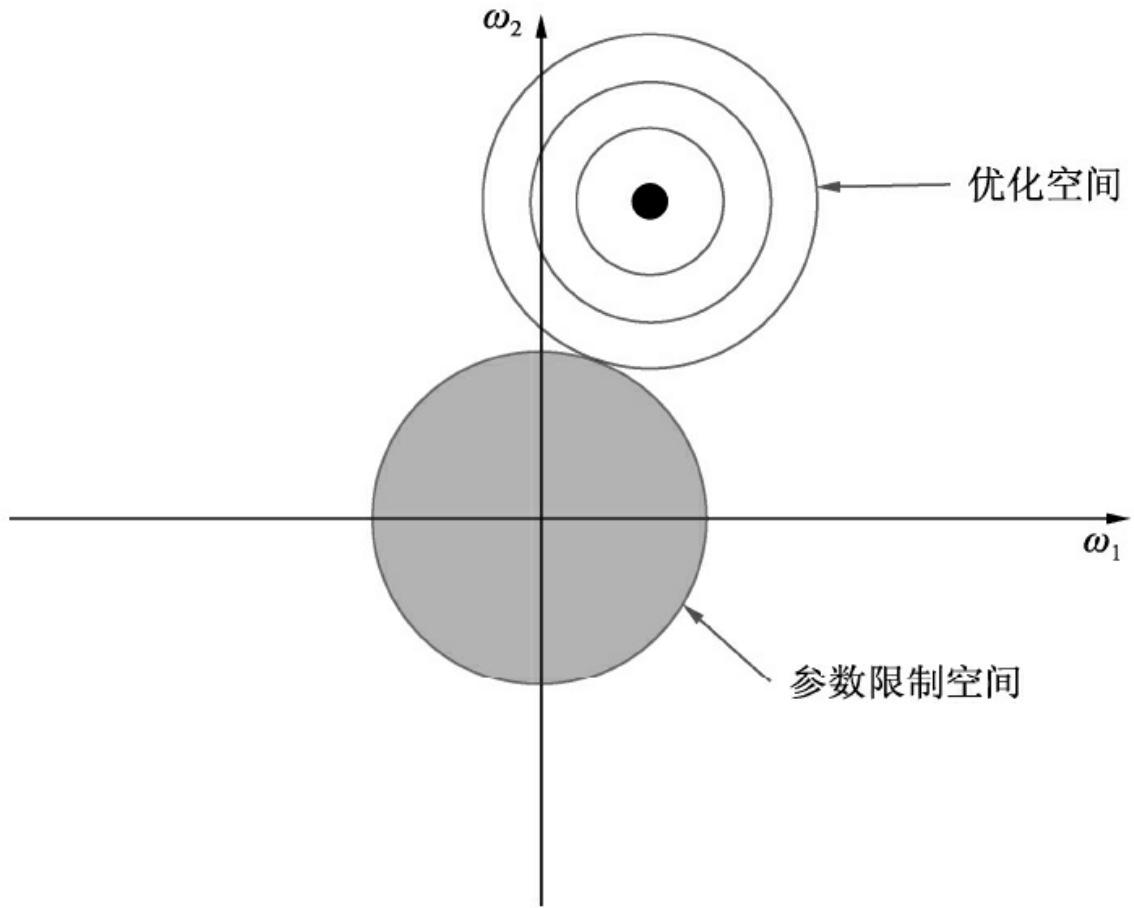


图8.15 L2正则化的参数空间与优化空间

## 8.4 总结

受当前物体检测框架的限制，模型通常需要NMS做后处理，但NMS存在“先天”的缺陷，并且样本的生成中容易产生不均衡，如果训练方法不当也容易出现过拟合现象。本章对这3个问题进行了详细的分析，并给出了多种缓解方案，能够在一定程度上解决这几个问题。对模型细节的改进也是近几年物体检测文章的一大热点。

在了解完细节问题后，接下来的一章我们将接触物体检测中较难检测的问题，攀登物体检测的“高峰”。

## 第9章 物体检测难点

经过前面几章的学习，我们已经可以利用多种算法很好地解决通用的物体检测问题。然而在一些难点问题上，如多尺度物体及拥挤物体的检测上，当前算法还存在很大的提升空间。

这两个问题也是当前学术界物体检测领域研究的热点，因此本章将从特定难点问题的角度出发，介绍一些有效的解决算法。

## 9.1 多尺度检测

传统卷积网络通常采用从上到下的单行结构。对于大物体而言，其语义信息将出现在较深的特征图中；而对于小物体，其语义信息出现在较浅的特征图中，随着网络的加深，其细节信息可能会完全消失。因此对于传统的检测算法，大尺度物体与小尺度物体的问题始终没能得到较好的平衡。

多尺度检测也是当今物体检测领域最为活跃的研究主题之一，本节首先会分析多尺度检测的背景，然后将重点介绍多种解决多尺度检测的经典算法。

### 9.1.1 多尺度问题

在物体检测的各种实际应用场景中，为满足需求，我们通常希望检测出不同大小的物体。如图9.1所示，最前方放风筝的人约有几百个像素高，而在更远处的沙滩上的人则可能只有30个像素高。

多尺度也是物体检测与图像分类两个任务的一大区别。分类问题通常针对同一种尺度，如ImageNet中的224大小；而物体检测中，模型需要对不同尺度的物体都能检测出来，这要求模型对于尺度要具有鲁棒性。

在多尺度的物体中，大尺度的物体由于面积大、特征丰富，通常来讲较为容易检测。难度较大的主要是小尺度的物体，而这部分小物体在实际工程中却占据了较大的比例。小物体通常有如下两种定义方式：



## 图9.1 COCO数据集中的多尺度物体示例

- 绝对尺度：**一般尺寸小于 $32 \times 32$ 的物体可以视为小物体。
- 相对尺度：**物体宽高是原图宽高的 $1/10$ 以下，可以视为小物体。

小物体由于其尺寸较小，可利用的特征有限，这使得其检测较为困难。当前的检测算法对于小物体并不友好，体现在以下4个方面：

- 过大的下采样率：**假设当前小物体尺寸为 $15 \times 15$ ，一般的物体检测中卷积下采样率为16，这样在特征图上，小物体连一个点都占据不到。
- 过大的感受野：**在卷积网络中，特征图上特征点的感受野比下采样率大很多，导致在特征图上的一个点中，小物体占据的特征更少，会包含大量周围区域的特征，从而影响其检测结果。
- 语义与空间的矛盾：**当前检测算法，如Faster RCNN，其Backbone大都是自上到下的方式，深层与浅层特征图在语义性与空间性上没有做到更好的均衡。
- SSD缺乏特征融合：**SSD虽然使用了多层特征图，但浅层的特征图语义信息不足，没有进行特征的融合，致使小物体检测的结果较差。

多尺度的检测能力实际上体现了尺度的不变性，当前的卷积网络能够检测多种尺度的物体，很大程度上是由于其本身具有超强的拟合能力。

总体来讲，目前有效解决多尺度问题的方法在此总结了6种，如图9.2所示。

在这些方法中，降低下采样率与空洞卷积可以显著提升小物体的检测性能；设计更好的Anchor可以有效提升Proposal的质量；多尺度的训

练习可以近似构建出图像金字塔，增加样本的多样性；特征融合可以构建出特征金字塔，将浅层与深层特征的优势互补。这4条是较为通用的提升多尺度检测的经典方法。

除此之外，还有两个最新的方法：SNIP与TridentNet，这两个方法的思想简洁，实现优雅，对多尺度问题的提升很明显，也将重点介绍。

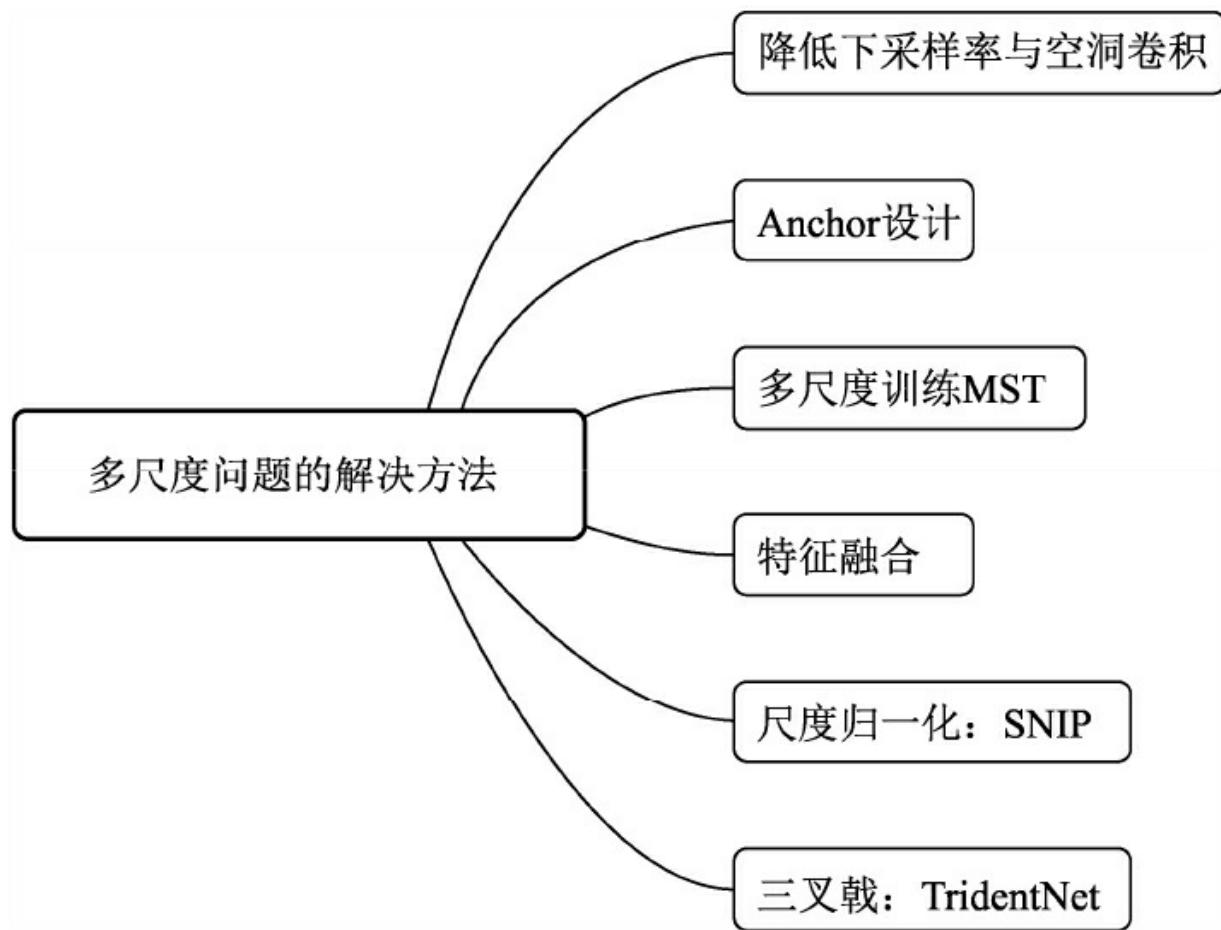


图9.2 多尺度物体的解决方法汇总

### 9.1.2 降低下采样率与空洞卷积

对于小物体检测而言，降低网络的下采样率也许是最为简单的提升方式，通常的做法是直接去除掉Pooling层。例如，将原始的VGGNet-16作为物体检测的Backbone时，通常是将第5个Pooling层之前的特征图作为输出的特征图，一共拥有4个Pooling层，这时下采样率为16。

为了降低下采样率，我们可以将第4个Pooling层去掉，使得下采样率变为8，减少了小物体在特征图上的信息损失。但是，如果仅仅去除掉Pooling层，则会减小后续层的感受野。如果使用预训练模型进行微调（Fine-tune），则仅去除掉Pooling层会使得后续层感受野与预训练模型对应层的感受野不同，从而导致不能很好地收敛。

因此，我们需要在去除Pooling的前提下增加后续层的感受野，空洞卷积就派上用场了。由第3章可知，空洞卷积可以在保证不改变网络分辨率的前提下增加网络的感受野。具体做法如图9.3所示，去掉第4个Pooling层后，将后续的一个 $3 \times 3$ 卷积变为空洞数为2的卷积，可以达到简单有效的降低下采样的目的。

需要注意的是，采用空洞卷积也不能保证修改后与修改前的感受野完全相同，但能够最大限度地使感受野在可接受的误差内。

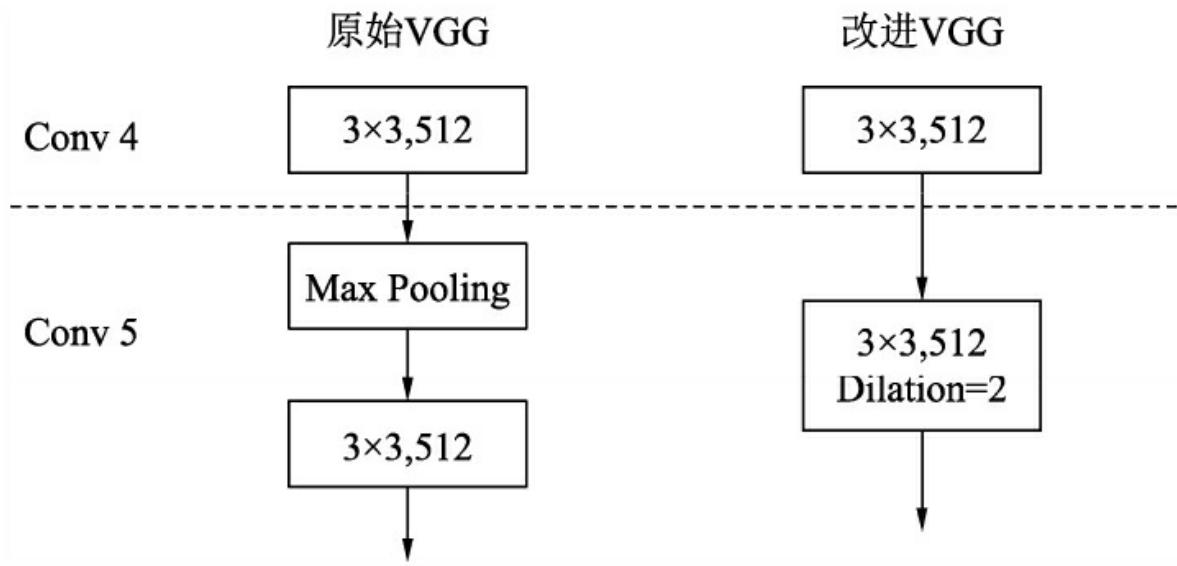


图9.3 VGGNet降低下采样率的常用处理方法

### 9.1.3 Anchor设计

现今较为成熟的检测算法大都采用Anchor作为先验框，如Faster RCNN和SSD等，因为Anchor可以提供很好的先验信息。模型在Anchor的基础上只需要去预测其与真实物体边框的偏移即可，可以说是物体检测算法发展中的一个相当经典的设计。

Anchor通常是由多个不同大小与宽高的边框组成的，这个大小与宽高是一组超参数，需要我们手动配置。在不同的数据集与任务中，由于物体的尺度、大小会有差距，例如行人检测的数据集中，行人标签宽高比通常为0.41，与通用物体的标签会有所区别，这时就需要相应地调整Anchor的大小与宽高。如果Anchor设计的不合理，与数据集中的物体分布存在差距，则会给模型收敛带来较大的困难，影响模型的精度，甚至不会收敛。

另外，Anchor的设计对于小物体的检测也尤为重要，如果Anchor过大，即使小物体全部在Anchor内，也会因为其自身面积小导致IoU低，从而造成漏检。

通常来讲，可以从以下两个角度考虑如何设计一组好的Anchor。

#### 1. 统计实验

首先回顾一下Faster RCNN是如何处理Anchor的：在RPN阶段，所有Anchor会与真实标签进行匹配，根据匹配的IoU值得到正样本与负样本，正样本的IoU阈值为0.7。在这个过程中，Anchor与真实标签越接近，正样本的IoU会更高，RPN阶段对于真实标签的召回率会越高，正样本也会更丰富，模型效果会更好。

因此，我们可以抛开物体检测的算法，仅仅利用训练集的标签与设计的Anchor进行匹配试验，试验的指标是所有训练标签的召回率，以及正样本的平均IoU值。当然，也可以增加每个标签的正样本数、标签的最大IoU等作为辅助指标。

为了方便地匹配，在此不考虑Anchor与标签的位置偏移，而是把两者的中心点放在一起，仅仅利用其宽高信息进行匹配。这种统计实验实际是通过手工设计的方式，寻找与标签宽高分布最为一致的一组Anchor。

## 2. 边框聚类

相比起手工寻找标签的宽高分布，我们也可以利用聚类的思想，在训练集的标签上直接聚类出一组合适的Anchor。由于一组Anchor会出现在特征图的每一个位置上，因此没有位置区别，可以只关注标签里的物体宽高，而没必要关心物体出现的位置。

边框聚类时通常使用K-Means算法，这也是YOLO采用的Anchor聚类方法。作为最简单的聚类算法之一，K-Means算法的计算过程如图9.4所示，输入超参数K，即最终想要获得的边框数量，首先随机选取K个中心点，然后遍历所有的数据，并将所有的边框划分到最近的中心点中。

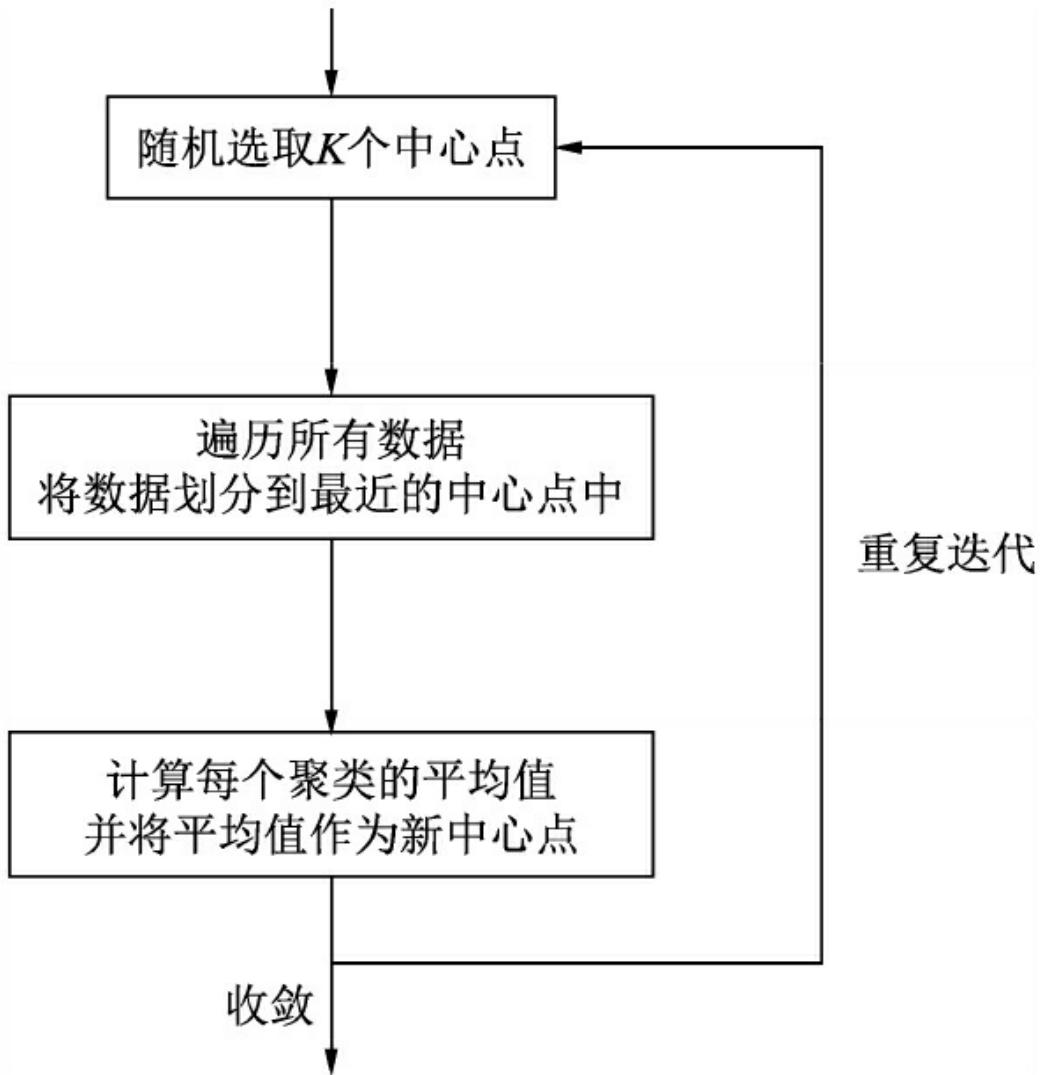


图9.4 K-Means算法的主要过程

在每个边框都落到不同的聚类后，计算每一个聚类的平均值，并将此平均值作为新的中心点。重复上述过程，直到算法收敛。

在聚类过程中，Anchor的数量K是一个较为重要的超参，数量越多，精度越高，但与此同时会带来计算量的增加。对于使用Anchor的物体检测算法而言，设计一组好的Anchor是基础，这对于多尺度、拥挤等问题都有较大的帮助。

#### 9.1.4 多尺度训练

关于多尺度，我们首先可以联想到数字图像处理中的图像金字塔，即将输入图片缩放到多个尺度下，每一个尺度单独地计算特征图，并进行后续的检测。这种方式虽然一定程度上可以提升检测精度，但由于多个尺度完全并行，耗时巨大。

当前的多尺度训练（Multi Scale Training, MST）通常是指设置几种不同的图片输入尺度，训练时从多个尺度中随机选取一种尺度，将输入图片缩放到该尺度并送入网络中，是一种简单又有效的提升多尺度物体检测的方法。虽然一次迭代时都是单一尺度的，但每次都各不相同，增加了网络的鲁棒性，又不至于增加过多的计算量。

而在测试时，为了得到更为精准的检测结果，也可以将测试图片的尺度放大，例如放大4倍，这样可以避免过多的小物体。

多尺度训练是一种十分有效的trick方法，放大了小物体的尺度，同时增加了多尺度物体的多样性，在多个检测算法中都可以直接嵌入，在不要求速度的场合或者各大物体检测竞赛中尤为常见。

### 9.1.5 特征融合

传统的卷积网络通常是自上而下的模式，随着网络层数的增加，感受野会增大，语义信息也更为丰富。这种自上而下的结构本身对于多尺度的物体检测就存在弊端，尤其是小物体，其特征可能会随着深度的增加而渐渐丢失，从而导致检测性能的降低。

既然深浅层的特征各有优势，一个自然的想法就是将深层的语义信息添加到浅层的特征图中，融合两者的特征，优势互补，从而提升对于小物体的检测性能。

在近些年的物体检测研究中，诞生了众多从特征融合角度提升多尺度检测的方法，如图9.5所示，在前面的多个章节也有很多介绍，这里做一个总结。

- FPN**: 将深层信息上采样，与浅层信息逐元素地相加，从而构建了尺寸不同的特征金字塔结构，性能优越，现已成为物体检测算法的一个标准组件。

- DetNet**: 专为物体检测而生的Backbone，利用空洞卷积与残差结构，使得多个融合后的特征图尺寸相同，从而也避免了上采样操作。

- Faster RCNN**系列中，HyperNet将第1、3、5个卷积组后得到的特征图进行融合，浅层的特征进行池化、深层的特征进行反卷积，最终采用通道拼接的方式进行融合，优势互补。

- SSD**系列中，DSSD在SSD的基础上，对深层特征图进行反卷积，与浅层的特征相乘，得到了更优的多层特征图，这对于小物体的检测十分有利。

·RefineDet将SSD的多层特征图结构作为了Faster RCNN的RPN网络，结合了两者的优势。特征图处理上与FPN类似，利用反卷积与逐元素相加，将深层特征图与浅层的特征图进行结合，实现了一个十分精巧的检测网络。

·YOLO系列中，YOLO v3也使用了特征融合的思想，通过上采样与通道拼接的方式，最终输出了3种尺寸的特征图。

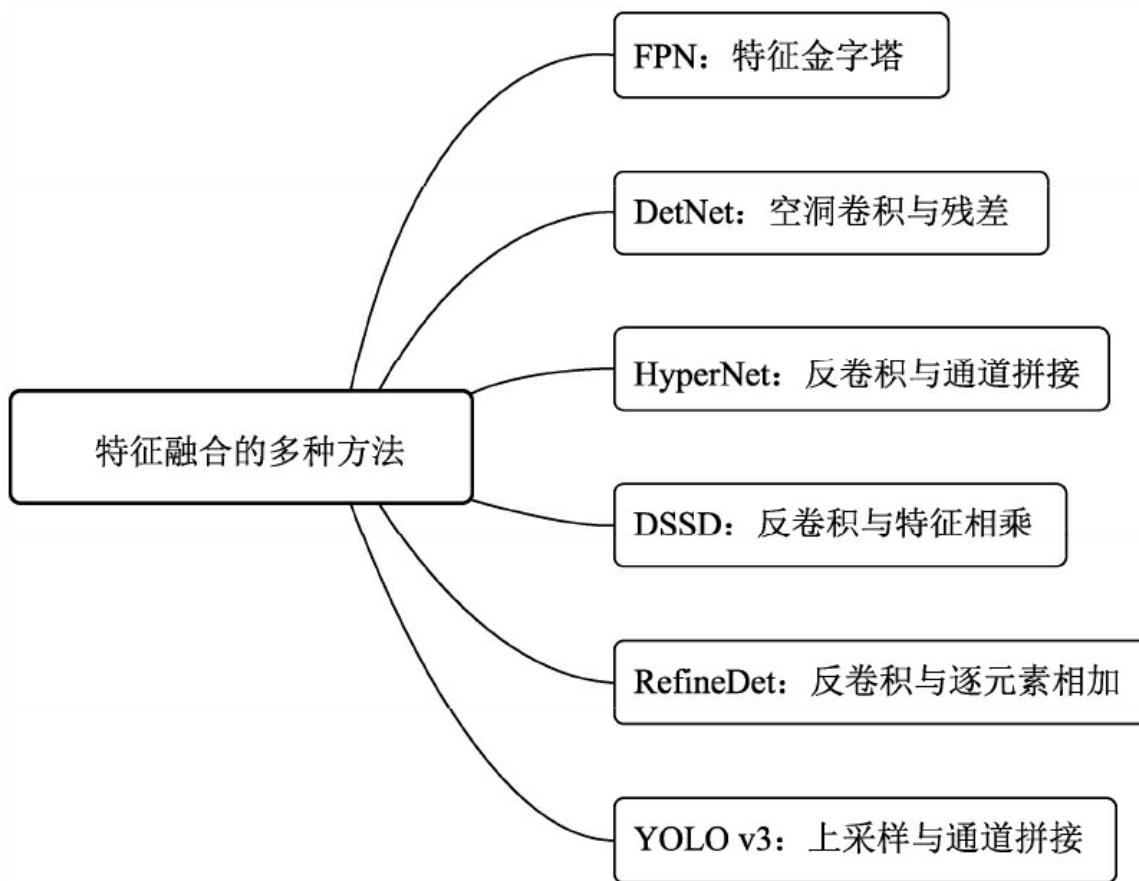


图9.5 特征融合的多种方法

可以看出，特征融合有多种方式，增大特征图尺寸可以使用上采样、反卷积等，融合方法有逐元素相加、相乘和通道拼接等，具体哪种效果更好，还要看实际的检测任务及使用的检测算法。特征融合的普遍缺点是通常会带来一定计算量的增加。

### 9.1.6 尺度归一化：SNIP

提到多尺度物体的检测，就不能不提2018年CVPR的SNIP（Scale Normalization for Image Pyramid），相比起其他改进方法，SNIP更为深入地指出了当前多尺度检测问题的原因，使用简单的方法就得到了较高的检测精度提升，是一个令人印象深刻的工作。

当前的物体检测算法通常使用微调的方法，即先在ImageNet数据集上训练分类任务，然后再迁移到物体检测的数据集上，如COCO来训练检测任务。我们可以将ImageNet的分类任务看做 $224 \times 224$ 的尺度，而COCO中的物体尺度大部分在几十像素的范围内，并且包含大量小物体，物体尺度差距更大，因此两者的样本差距太大，会导致映射迁移（Domain Shift）的误差。

为了克服多尺度检测的问题，有以下两种经典的方法，在上面的方法中也提到了。

- 图像金字塔：将输入图像做成多尺度，或者随机取一个尺度。
- 特征金字塔：对深层特征上采样，融合多层的特征，将语义性与空间性进行优势互补。

但SNIP对这两种方法提出了质疑，即能否避开上采样的方法，在训练时使用所有尺度的样本进行训练呢？基于此，SNIP的作者做了一个非常关键的实验，分析了多种方法在COCO数据集上小物体（尺度小于 $32 \times 32$ ）上的表现，测试图像尺度统一为 $1400 \times 2000$ 像素，实验结果如表9.1所示，以分析影响多尺度检测的原因。

表9.1 多种训练策略在COCO小物体上的mAP

$1400_{<80px}$	$800_{all}$	$1400_{all}$	MST	SNIP
16.4	19.6	19.9	19.5	21.4

表9.1中 $1400_{<80px}$ 代表将图像上采样后只选择尺度小于80个像素的样本进行训练， $800_{all}$ 与 $1400_{all}$ 分别是将图像的短边缩放到800与1400的大小，然后使用所有尺度的样本进行训练，MST代表了随机的多尺度训练方法。

通过该实验结果，我们可以对比分析出以下3点：

- 既然多尺度物体难以训练，那么在检测小物体时，能否将大物体样本去掉以提升小物体检测性能呢？ $1400_{<80px}$ 与 $1400_{all}$ 两者的结果表明是否定的，去掉了大量的大物体会导致物体多样性的损失。

- 如果使用更大的尺寸进行训练，小物体的检测应该会有明显的提升，但 $1400_{all}$ 相比于 $800_{all}$ 仅仅提升了0.3%。

- MST将图片缩放到了多个尺度上，增加了样本的多样性，理应有较好的检测效果，但实验结果表明并没带来明显的提升。

从上述结果可以得知，即使充分增强了数据的多样性，由于卷积网络本身不具备尺度不变性，依靠模型去学习多尺度的物体仍然存在很大难度。当前卷积网络对于多个尺度都还能检测出的原因在于卷积网络强大的拟合能力，搜索空间足够大来学习不同的尺度，这也浪费了网络的拟合能力。

基于此实验结论，SNIP让模型更专注于物体本身的检测，剥离了多尺度的学习难题。在网络搭建时，SNIP也使用了类似于MST的多尺度训练方法，构建了3个尺度的图像金字塔，但在训练时，只对指定范围内的Proposal进行反向传播，而忽略掉过大或者过小的Proposal。具体的实现细节如下：

·3个尺度分别拥有各自的RPN模块，并且各自预测指定范围内的物体。

·对于大尺度的特征图，其RPN只负责预测被放大的小物体，对于小尺度的特征图，其RPN只负责预测被缩小的大物体，这样真实的物体尺度分布在较小的区间内，避免了极大或者极小的物体。

·在RPN阶段，如果真实物体不在该RPN预测范围内，会被判定为无效，并且与该无效物体的IoU大于0.3的Anchor也被判定为无效的Anchor。

·在训练时，只对有效的Proposal进行反向传播。在测试阶段，对有效的预测Boxes先缩放到原图尺度，利用Soft NMS将不同分辨率的预测结果合并。

·实现时SNIP采用了可变形卷积的卷积方式，并且为了降低对于GPU的占用，将原图随机裁剪为 $1000 \times 1000$ 大小的图像。

SNIP方法虽然实现简单，但其背后却蕴藏深意，更深入地分析了当前检测算法在多尺度检测上的问题所在，在训练时只选择在一定尺度范围内的物体进行学习，在COCO数据集上有3%的检测精度提升，可谓是大道至简。

### 9.1.7 三叉戟：TridentNet

传统的解决多尺度检测的算法，大都依赖于图像金字塔与特征金字塔，这在前面也有详细的阐述。与上述算法不同，图森组对感受野这一因素进行了深入的分析，并利用了空洞卷积这一利器，构建了简单的三分支网络TridentNet，对于多尺度物体的检测有了明显的精度提升。

TridentNet网络的作者首先分析了不同大小的感受野对于检测结果的影响。实验分别采用了ResNet-50及ResNet-101作为Backbone，仅仅改变最后一个阶段的每个 $3 \times 3$ 卷积的空洞数，这也得益于空洞卷积可以在相同的结构、参数量与下采样率时，改变网络的感受野。实验结果如表9.2所示。

表9.2 Faster RCNN不同感受野在COCO数据集上的检测结果

Backbone	空 洞 数	AP	AP <sub>s</sub>	AP <sub>m</sub>	AP <sub>l</sub>
ResNet-50	1	0.332	<b>0.174</b>	0.384	0.464
	2	0.342	0.168	<b>0.386</b>	0.486
	3	0.341	0.162	0.383	<b>0.492</b>
ResNet-101	1	0.379	<b>0.200</b>	<b>0.430</b>	0.528
	2	0.380	0.191	0.427	<b>0.538</b>
	3	0.371	0.181	0.410	<b>0.538</b>

表9.2中，AP<sub>s</sub>、AP<sub>m</sub>与AP<sub>l</sub>分别代表小、中、大3个不同尺度的评测子集。从结果可以很直观地看出，不同尺度的检测性能与感受野呈正相关，即大的感受野对于大物体更友好，小的感受野对于小物体更友好。基于此结论，进一步联想，能否将这3种不同的检测性能结合，实现优势互补呢？

基于此，TridentNet网络的作者将这3种不同的感受野网络并行化，提出了如图9.6所示的检测框架。采用ResNet作为基础Backbone，前三个

stage沿用原始的结构，在第四个stage，使用了三个感受野不同的并行网络。

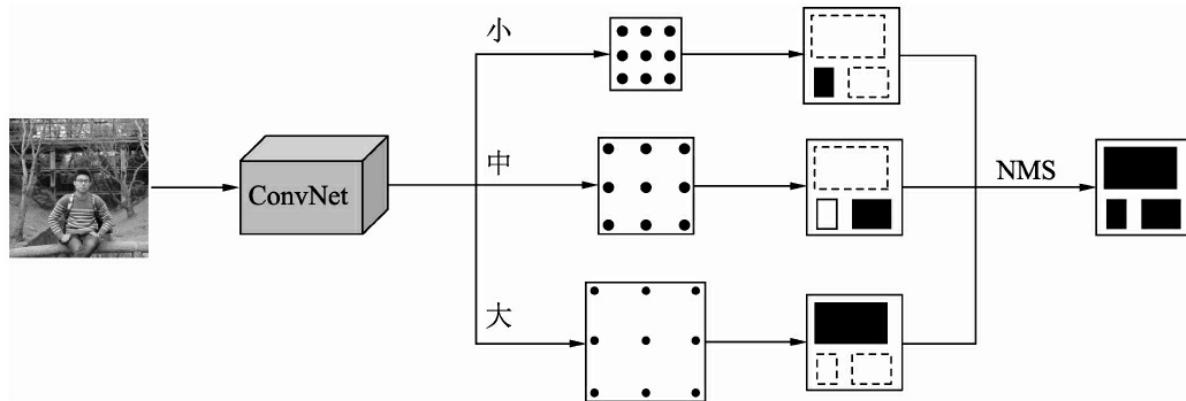


图9.6 TridentNet网络结构图

对于该结构，有以下3个细节：

·3个不同的分支使用了空洞数不同的空洞卷积，感受野由小到大，可以更好地覆盖多尺度的物体分布。

·由于3个分支要检测的内容是相同的、要学习的特征也是相同的，只不过是形成了不同的感受野来检测不同尺度的物体，因此，3个分支共享权重，这样既充分利用了样本信息，学习到更本质的物体检测信息，也减少了参数量与过拟合的风险。

·借鉴了SNIP的思想，在每一个分支内只训练一定范围内的样本，避免了过大与过小的样本对于网络参数的影响。

在训练时，TridentNet网络的三个分支会接入三个不同的head网络进行后续损失计算。在测试时，由于没有先验的标签来选择不同的分支，因此只保留了一个分支进行前向计算，这种前向方法只有少量的精度损失。

TridentNet网络的作者对该网络的性能做了详尽的分析，对于当前

的基础Backbone等方法均有明显的精度提升。简而言之，TridentNet思路清晰、方法简单有效，对于多尺度物体的检测提供了很好的研究视角。

## 9.2 拥挤与遮挡

除了多尺度问题，物体之间的拥挤与遮挡是另一个常见的检测难点。拥挤与遮挡会带来物体的信息缺失，物体的部分区域是不可见的，边界模糊，容易造成检测器的误检与漏检，从而降低检测性能。

与通用的物体检测相比，遮挡问题在行人的检测中更为普遍，该问题也是行人检测中最为棘手的问题之一。本节将会从行人检测的角度，介绍遮挡问题的背景，以及如何解决行人检测中的遮挡问题。

### 9.2.1 遮挡背景

在自动驾驶、智能监控等应用场景中，行人的遮挡问题十分常见。根据遮挡物体的性质，可以进一步将遮挡分为以下两种情况：

#### 1. 行人之间的自遮挡

行人之间由于拥挤造成的自遮挡是最为常见的遮挡情形，在日常道路上随处可见。图9.7是CityPersons数据集中的两个行人拥挤的场景。

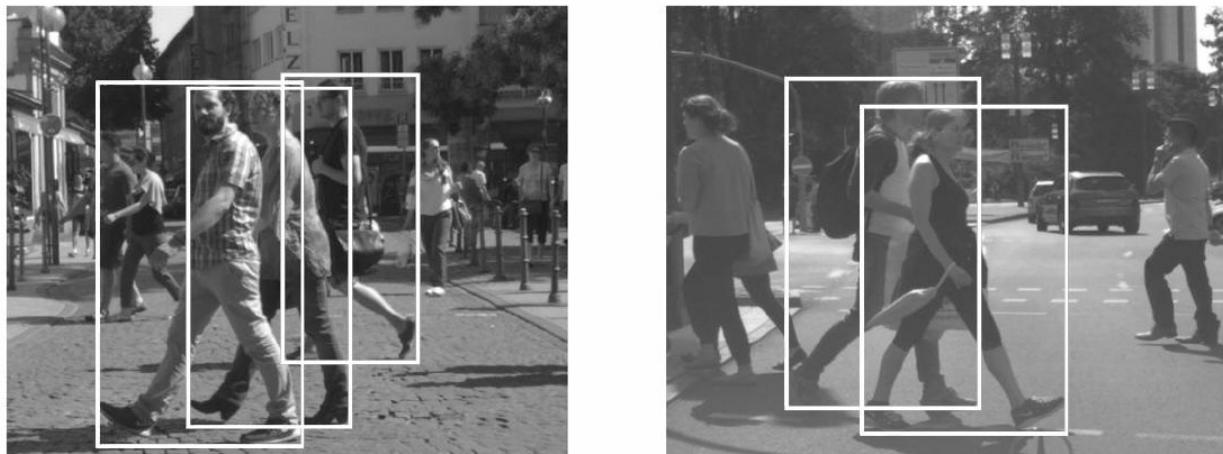


图9.7 CityPersons数据集中的行人自遮挡示例

表9.3展示了CityPersons验证集上的行人遮挡比例，可以看出，行人之间的自遮挡情形十分普遍。

表9.3 CityPersons验证集上行人遮挡比例

总行人标签	行人之间遮挡IoU大于 0.1	行人之间遮挡IoU大于 0.3
3157	48.8%	26.4%

行人之间的自遮挡会严重影响检测的性能，主要体现在以下两个方面：

·定位不准确：在提取行人的特征时，遮挡会带来特征的缺失，并且距离很近的行人特征会相互影响，带来干扰，这都会降低行人定位的准确性。

·对NMS的阈值更为敏感：用于行人靠得很近，如果NMS的阈值较低，则很容易将本属于两个行人的预测框抑制掉一个，造成漏检；而如果阈值较高，很可能会在两个行人之间多了一个错误预测框，造成误检。因此，行人的自遮挡对NMS十分不友好，从而导致检测性能的降低。

## 2. 行人被其他物体遮挡

除了行人之间的相互遮挡，行人还有可能被各种周围物体遮挡，如图9.8中，行人分别被自行车和轿车遮挡住。这种遮挡会导致大面积的信息缺失，即使肉眼也不容易判断行人的边框具体位置。同时，这种遮挡也会带来较高的行人漏检。



图9.8 CityPersons中行人被其他物体遮挡的示例

关于行人检测的数据集，当前比较主流的有Caltech与CityPersons。Caltech是一个规模较大的行人检测数据集，其包含了大约25万张图片，

2300个行人标签，是一个十分常用的行人检测数据集。

相比之下，CityPersons则是在Cityscape城市数据集的基础上，标注了更为全面的行人标签，从行人的数量、小物体、遮挡的丰富性来讲，都胜过以往的数据集，现已成为行人检测领域的首选性能评测数据集。

在行人检测中，多尺度检测的问题同样严重，因此使用9.1节的多种方法，如上采样、降低下采样率、SNIP、特征融合等，也可以有效提升行人检测的性能。尤其需要指出的是，由于行人具有特殊的形状，其高宽比集中在2.41左右，因此设计针对性更强的Anchor，也可以有效提升检测的精度。

对于行人检测的遮挡问题，这里总结了目前较为有效的5个解决方法，如图9.9所示。

首先对前3种方法进行简要介绍：

·改进NMS：由于NMS对行人遮挡检测影响很大，因此改进NMS是一个出路，像第8章中的Soft NMS和IoU-Net等方法都能在一定程度上提升遮挡检测的性能。

·增加语义信息：遮挡会造成行人部分信息的缺失，因此可以尝试引入额外的特征，如分割信息、梯度和边缘信息等，详细方法可见CVPR 2017中的HyperLearner。

·划分多个part处理：由于行人之间的形状较为相似，因此可以利用这个先验信息，将行人按照不同部位，如头部、上身、手臂等划分为多个part进行单独处理，然后再综合考虑，可以在一定程度上缓解遮挡带来的整体信息缺失。

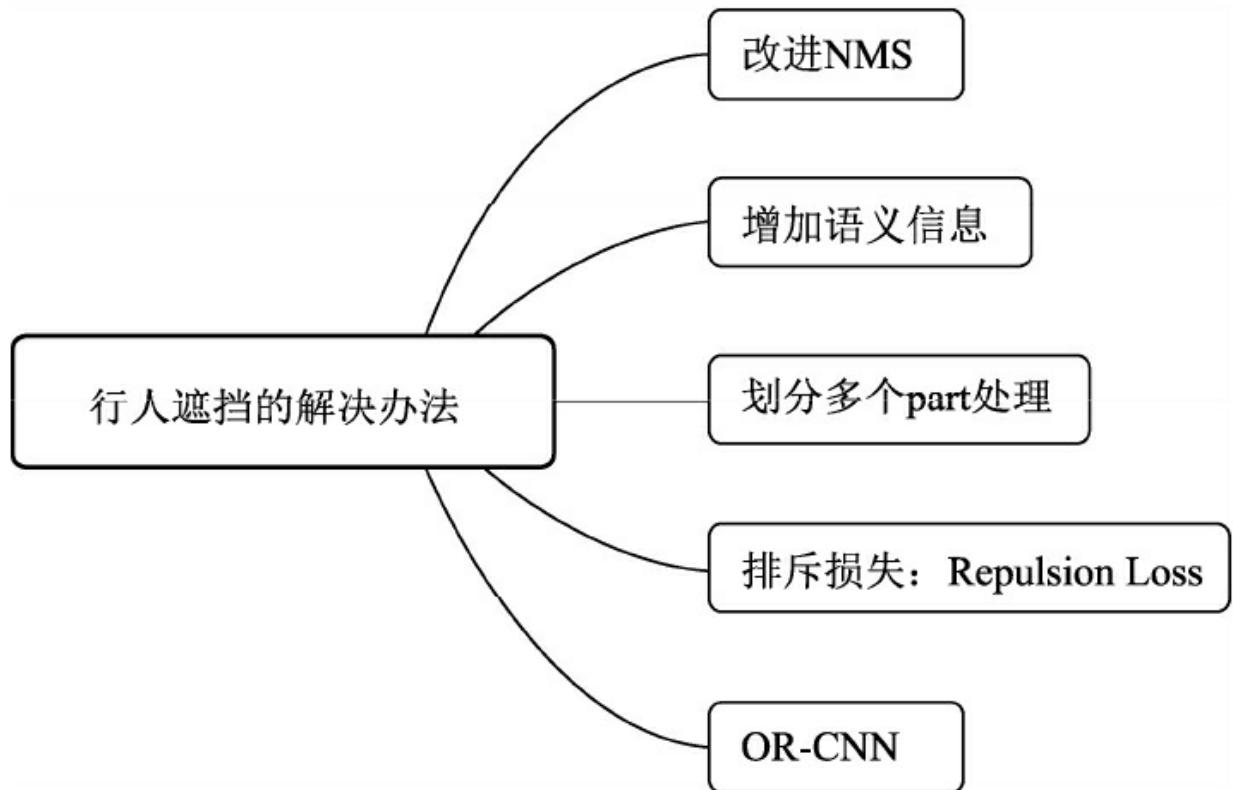


图9.9 行人遮挡的解决办法汇总

最后的两个方法，Repulsion Loss及OR-CNN，则是近两年在行人检测领域涌现出的性能优越的方法，因此在本节接下来的篇幅中，将对这两种方法进行详细的介绍。

## 9.2.2 排斥损失：Repulsion Loss

针对行人检测的遮挡问题，王鑫龙等人发表于CVPR 2018的 Repulsion Loss方法从损失函数的角度，提出了一种新颖的优化方法，有效缓解了行人遮挡情况下的检测，在多个数据集上达到了较好的效果。

此外，作者进一步分析了当前Faster RCNN对于遮挡问题的检测能力。在漏检的行人中，被遮挡的行人占到了60%，而在被遮挡的行人中，受拥挤而被其他行人遮挡的行人比例占据了60%。从这个实验结果足以说明，遮挡问题是影响行人检测性能非常重要的原因，尤其是行人的自遮挡。

Faster RCNN由于其两阶的设计，在行人检测领域比SSD等一阶网络更有优势，通常作为行人检测的基础结构。我们知道，Faster RCNN对于边框的位置采用了smoothL1函数来计算损失，这里的目的是使RoI更加贴合真实物体的边框。

但是，当物体靠得太近时，周围物体会对真实物体的回归造成干扰。基于此，王鑫龙等研究人员在回归损失的基础上，创造性地提出了排斥损失Repulsion Loss，来缓解物体之间的相互影响。笔者个人认为这里的排斥吸引损失与路径规划中的人工势场法有异曲同工之妙。

Repulsion Loss的回归损失如式（9-1）所示。

$$L = L_{attr} + \alpha \times L_{RepGT} + \beta \times L_{RepBox} \quad (9-1)$$

可以看出，损失由以下3部分组成：

· $L_{attr}$ 代表当前样本预测框与真值的吸引回归损失，沿用了Faster

RCNN的smooth<sub>L1</sub>函数。

·L<sub>RepGT</sub>表示当前预测框与周围真实物体的排斥损失。

·L<sub>RepBox</sub>表示当前预测框与周围其他预测框之间的排斥损失。

这3部分损失使用了 $\alpha$ 与 $\beta$ 来调节损失权重，实验表明，当 $\alpha$ 与 $\beta$ 都为0.5时，检测性能最佳。接下来将分别介绍两种排斥损失的计算过程。

## 1. 与其他标签的排斥：RepGT

RepGT损失的设计思想是为了让当前预测框尽可能地远离周围的标签物体，这里的周围标签物体指的是，除了预测框本身要回归的物体之外，与该预测框有最大IoU的物体标签，选取公式如式（9-2）所示。

$$G_{Rep}^P = \operatorname{argmax}_{G \in g \setminus \{G_{Attr}^P\}} IoU(G, P) \quad (9-2)$$

受IoU Loss的启发，这里衡量预测框与周围物体标签的距离使用了如（式9-3）所示的IoG（Intersection over GroundTruth）函数。

$$IoG(B, G) \triangleq \frac{\operatorname{area}(B \cap G)}{\operatorname{area}(G)} \quad (9-3)$$

由公式（9-3）可以看出，IoG的分母仅仅是物体标签的面积，这样分母在优化的过程中始终是常量，减小IoG只能通过减小B与G的重叠面积。如果这里采用IoU作为距离函数来优化的话，由于分母不是常量，简单地增大预测框的面积也可以达到减小IoU的目的，这与我们的期望并不一致，会带来模型优化不稳定的问题。

在完成了 $G_{Rep}^P$ 的选择及IoG的计算后，就可以计算排斥损失了。为

了最小化IoG，采用了如式（9-4）所示的Smooth<sub>ln</sub>作为优化函数。

$$Smooth_{ln} = \begin{cases} -\ln(1-x) & x \leq \sigma \\ \frac{x-\sigma}{1-\sigma} - \ln(1-\sigma) & x > \sigma \end{cases} \quad (9-4)$$

与smooth<sub>L1</sub>类似，Smooth<sub>ln</sub>也是采用了一阶与更高阶函数的组合，其中引入了超参 $\sigma$ 来控制高阶函数的有效范围。需要注意的是，IoG的取值范围为[0,1]，这一点与smooth<sub>L1</sub>是不同的。

最终形式的所有预测框的RepGT损失如式（9-5）所示，其中 $\rho_+$ 为所有有效的预测框。

$$L_{RepGT} = \frac{\sum_{p \in \rho_+} Smooth_{ln}(IoG(B^p, G_{Rep}^p))}{|\rho_+|} \quad (9-5)$$

## 2. 与其他预测框的排斥：RepBox

在当前的检测框架中，NMS是非常重要的一个环节，用来抑制掉重叠的边框。然而在拥挤严重的行人检测中，NMS会容易造成边框的漏检。

针对此问题，Repulsion Loss增加了RepBox损失，目的是让预测框尽可能地远离周围预测框，降低两者之间的IoU，从而避免本属于两个物体的预测框，其中一个被NMS抑制掉。

根据图像中的物体标签，我们可以将最后的预测框划分为多个组。假设有 $g$ 个物体，则划分形式如式（9-6）所示，同组之间的预测框回归的是同一个物体标签，不同组之间的预测框对应的是不同的物体标签。

$$\rho_+ = \rho_1 \cap \rho_2 \cap \dots \cap \rho_{|g|} \quad (9-6)$$

然后，对于不同组之间的预测框 $\rho_i$ 与 $\rho_j$ ，我们希望 $\rho_i$ 与 $\rho_j$ 之间的重叠区域越小越好，可以使用IoU来衡量重叠区域，因此只需要最小化IoU。

RepBox损失同样也使用了Smooth<sub>ln</sub>作为优化的函数，整体损失公式如下：

$$L_{RepBox} = \frac{\sum_{i \neq j} Smooth_{ln}(IoU(B^{p_i}, B^{p_j}))}{\sum_{i \neq j} 1[IoU(B^{p_i}, B^{p_j}) > 0] + \epsilon} \quad (9-7)$$

其中， $\epsilon$ 是为了避免分母为0引进的很小的常量。

总体上，Repulsion Loss并没有大张阔斧地对网络结构进行改动，而是从损失层面引入了新颖的排斥损失，在多个行人检测数据集上都取得了优异的性能。

### 9.2.3 OR-CNN

作为通用物体检测的一个特定分支，行人的检测一般采用Faster RCNN作为基础框架。当前有效的行人检测策略一般有两种，一是从损失层面出发，设计更好的损失函数，如Repulsion Loss；另一种则是分解人为多个不同的部位，分别计算最后再融合。

张工峰等人发表于ECCV 2018的OR-CNN（Occlusion-aware R-CNN）在Faster RCNN的基础上，对损失函数及RoI Pooling两处进行了改进，引入了part-based的思想，有效地缓解了行人遮挡的问题。

为了更好地处理遮挡问题，张工峰等研究者设计了一种新的损失Aggregation Loss，使得多个匹配到同一个物体标签的Anchor尽量地靠近；在RoI Pooling处根据行人部位分为了5个不同的模块，提出了PORoI Pooling（Part Occlusion-aware RoI Pooling）方法。下面针对这两个改进进行详细介绍。

#### 1. 聚集损失：Aggregation Loss

在RPN阶段，聚集损失除了可以使Anchor更靠近物体标签之外，还可以将对应于同一个物体标签的所有Anchor紧密地分布，其损失函数如式（9-8）所示。

$$L_{agg}(\{p_i^*\}, \{t_i\}, \{t_i^*\}) = L_{reg}(\{p_i^*\}, \{t_i\}, \{t_i^*\}) + \beta \times L_{com}(\{p_i^*\}, \{t_i\}, \{t_i^*\})$$

其中， $L_{reg}$ 代表常规的回归损失，可以使Anchor更加靠近要回归的物体标签，沿用了Faster RCNN的smooth<sub>L1</sub>损失函数，在此不展开。

公式（9-7）中的第二项为紧密损失 $L_{com}$ （Compactness Loss），目的是使得要回归到同一个物体标签的所有Anchor尽可能靠的更近，具体函数形式如下：

$$L_{com}(\{p_i^*\}, \{t_i\}, \{t_i^*\}) = \frac{1}{N_{com}} \sum_{i=1}^{\rho} \Delta(\tilde{t}_i^* - \frac{1}{|\Phi_i|} \sum_{j \in \Phi_i} t_j) \quad (9-8)$$

公式中的参数含义如下：

- $\Phi_i$ 代表对应同一个物体标签的Anchor数量， $\frac{1}{|\Phi_i|} \sum_{j \in \Phi_i} t_j$  代表这些Anchor的位置均值，通过最小化每个Anchor与位置均值的差距，可以尽可能地将Anchor聚集到一起。

- $\rho$ 代表有多个Anchors对应的物体标签的数量。

- $N_{com}$ : 紧密损失的权重，一般与 $\rho$ 相同。

聚集损失也可以作用于Faster RCNN的第二个阶段，通过聚集的手段可以有效降低拥挤行人之间的误检，这一点与Repulsion Loss的思想有一点像，只不过使用了同类聚集取代了排斥的方法。

## 2. 行人部位拆解的池化：PORoI Pooling

将行人分多个部位拆解，分别提取不同部位的特征，再融合检测，这是较为常见的处理遮挡问题的解决办法。OR-CNN借鉴了该思想，并将这种思想融合到了RoI Pooling的过程中。

首先，利用行人的先验知识，将行人分成了5个部位，如图9.10所示。从图中可以看出，5个部位的大小与行人的宽W、高H为固定的比例关系，这种分部位的方法可以有效缓解遮挡带来的信息缺失。



图9.10 OR-CNN中行人的分部位示意图

RORoI Pooling将这种分部位的思想融入到了RoI Pooling中，如图9.11所示。图中行人的整体以及5个子部位，对这6个区域分别进行Pooling计算。

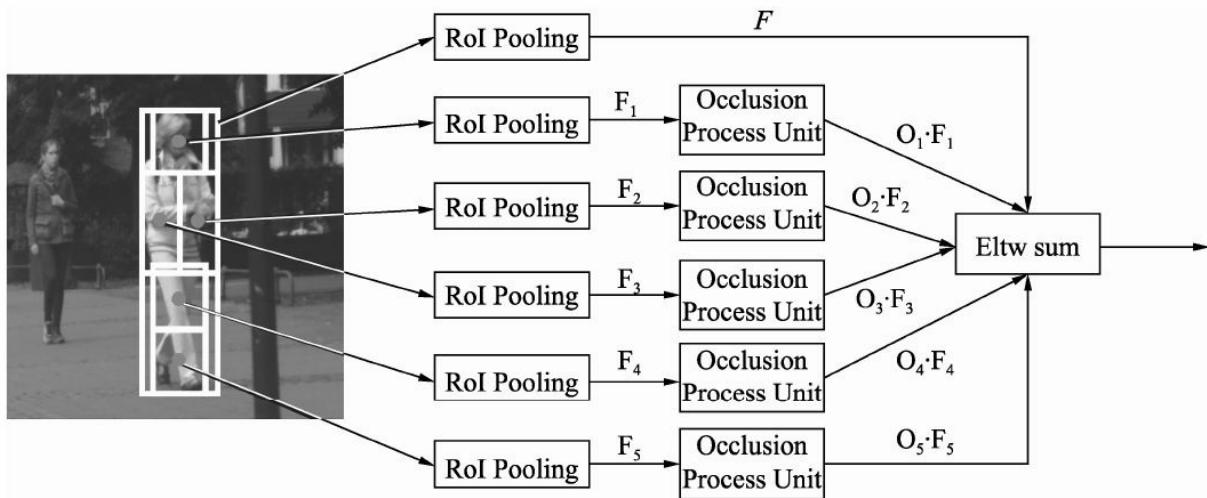


图9.11 PORoI Pooling网络结构图

子部位的特征进行Pooling后，还需要经过一个作者提出的遮挡处理单元（Occlusion Process Unit），目的是判断该子部位被遮挡的程度。最后，作者采用了逐元素相加的方式将6个区域的特征进行融合。

遮挡处理单元的具体形式如图9.12所示，这里的处理实际上是将遮挡的程度施加到原特征上，遮挡单元经过Softmax后得到了被遮挡的概率值，这部分可以与整个检测框架进行端到端地训练。

简而言之，OR-CNN思路简洁而有效，通过改善损失与RoI Pooling的方式，有效缓解了行人遮挡难检测的问题，在Caltech和CityPersons等多个行人数据集上达到了同期最高的检测性能。

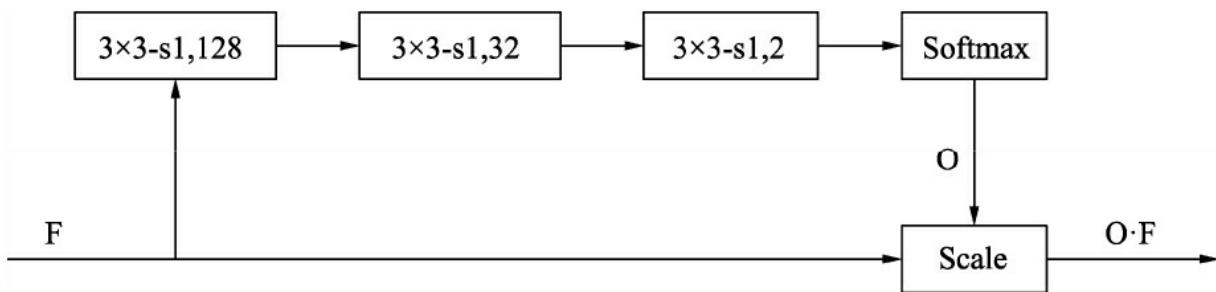


图9.12 遮挡处理单元的网络结构图

## 9.3 总结

多尺度问题是物体检测中极为常见又难以解决的问题，尤其是大量的小物体，存在天然的信息不足问题。本章首先分析了多尺度问题的背景，然后介绍了一系列缓解多尺度检测的方法，如设计更好的Anchor、SNIP和TridentNet等。

物体检测中另一个难点则是遮挡与拥挤，物体间的遮挡会导致信息的缺失与干扰，从而影响检测精度。拥挤与遮挡在行人检测中尤为突出，本章介绍了两个经典的用于解决行人拥挤遮挡的方法，分别是Repulsion Loss与OR-CNN。

在下一章中，笔者将系统性地总结物体检测算法，尤其是其未来发展的趋势。之后，重点介绍近几年发展迅速的摆脱锚框的检测算法。

## 第10章 物体检测的未来发展

经过前面章节的学习，读者应该对当前主流的物体检测算法，以及如何使用PyTorch搭建网络框架有了较为深入的理解。作为本书的最后一章，为了扩宽读者的思维空间，笔者将从更为宏观的角度介绍物体检测的未来发展趋势，尤其是有较大创新意义的方法，希望能对读者有所启发。

## 10.1 重新思考物体检测

在过去的几年的时间里，基于深度学习的检测算法取得了巨大成功。基于Faster RCNN、SSD和YOLO等改善算法你方唱罢我登场，在每年的顶级会议中都能看到它们的身影，其在工业界也有了诸多落地应用。

学习了诸多算法后，我们需要回过头来重新审视物体检测，思考当前有哪些关键点在限制着检测性能的进一步提升，以及卷积网络为什么有效、未来会有哪些发展趋势等。在此，笔者列举了7点物体检测还有待解决的问题，如图10.1所示。

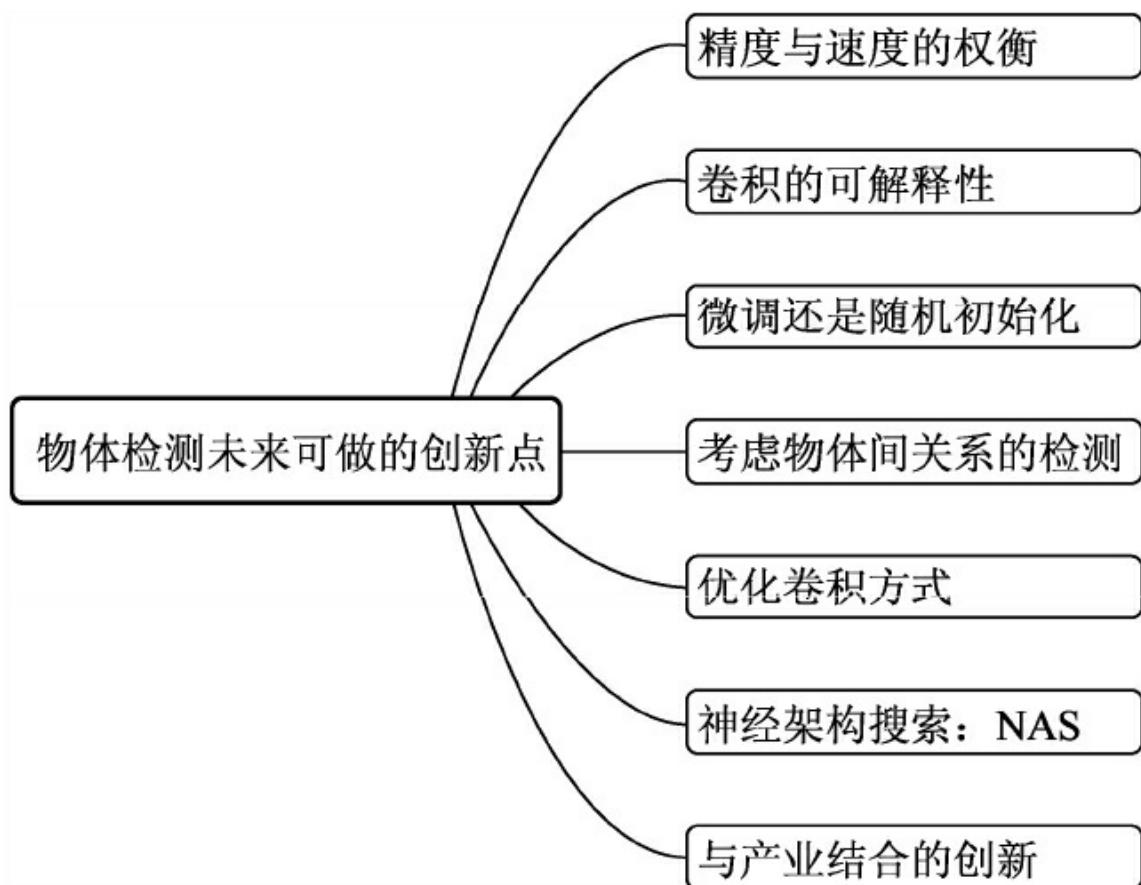


图10.1 物体检测未来可做的创新点

### 10.1.1 精度与速度的权衡

在前面的章节中可以看到，物体检测的发展始终围绕着精度与速度这两个指标，新提出的算法要么在检测的精度上有了新的突破，要么大幅提升了检测的速度。

然而，在当前的物体检测大框架下，精度与速度往往不可兼得，精度的提升往往伴随着计算量的增加，例如Cascade RCNN增加了网络阶数，FPN增加了特征之间的融合计算等。速度的提升往往会有精度的牺牲，如网络参数的量化和通道数的减少等。

虽然当前通用的检测框架已经较为成熟，检测指标也难有突破性的提升，但在特定的难点问题上仍有广阔的提升空间，这也是当前检测算法的薄弱环节。例如在智能安防、智能驾驶领域中，行人、车辆的多尺度与遮挡问题非常严重，检测器可以更好地与实际场景做结合。

当然，在实际工程应用中，不同的场景对于检测器的要求也各不相同，可以根据场景选择更为合适的检测器。下面介绍4个常见的算法需求场景。

·**速度需求：**自动驾驶等场景下，通常需要对图像处理达到非常低的时延才能保证足够的安全，这时检测器需要达到实时性；而在机械臂自动分拣等系统中，速度并不是第一考虑因素。

·**召回率：**在交通流量统计系统中，首先需要保障的指标是车辆、行人等物体的召回率，这会直接影响流量统计，相比之下，检测的边框精准度是次要的指标。

·**边框精准度：**在智能测量、机械臂自动分拣等应用中，检测边框

的精准度直接影响系统的成功率，因此需要选择边框精准度更高的网络，这是首要因素。

·移动端：当前，移动端的检测算法需求越来越高，如手机等ARM平台、边缘计算平台等，这对于模型的移动部署、轻量化提出了更高的需求。例如旷视的ThunderNet模型，利用两阶的结构，在ARM平台上实现了实时的物体检测。

### 10.1.2 卷积网络的可解释性与稳定性

当前，新的网络结构、检测思想往往是通过大量的实验对比得出的，并没有经过严谨的公式推导理论证明。卷积网络为什么有效，并没有得到一个公认的解释。

在实际场景使用时，检测器更像是一个黑箱，在有明确输入的场景下，我们并没有办法准确估计检测器的输出，这种不确定性，也限制了检测模型的应用范围。对于网络的可解释性，有一个名词叫做 XAI (Explainable Artificial Intelligence, 可解释人工智能)，即人工智能应当拥有可解释性、可靠性、透明性与负责属性，尤其是在涉及人身安全、政治、军事时，可解释性尤为重要。因此，深入分析卷积网络的可视化与可解释性，也是未来物体检测领域需要解决的问题。

此外，卷积网络可能并没有我们想象的那么稳定，有时会存在一些怪异的现象。例如，房间里的大象 (The Elephant in the Room) 这篇文章就针对检测器的鲁棒性做了很有趣的实验，将数据集中某图像中的一只大象移植到了别的图像中，并不断地平移，观察检测的结果，发现了以下3个现象：

- 检测不稳定**: 随着大象的移动，图像中的物体有时无法被检测到，或者检测的得分置信度发生剧烈变化。

- 物体错分类**: 随着大象的移动，图像中的物体可能会被错分为成其他类别的物体。

- 非物体边框内的干扰**: 即使大象与图像中的物体没有发生重叠，其检测类别、边框也会发生明显变化，甚至检测不出。

另外，该文章的作者还进一步分析了物体边框外的特征及边框内但不属于该物体的特征对检测的影响，最终将这种检测异常的现象归结于特征的干扰。

我们知道，目前物体的检测是将矩形边框内的所有特征考虑在内，并且较大的感受野使得最终的RoI会包含边框外较多的特征，而这两种本不属于物体的特征势必会影响检测的性能，这也是当前检测器的一个缺点。

在我们的认知里，由于拥有池化等操作，卷积网络对于微小的位移、变形等具有一定的鲁棒性。然而，最近的一个实验却发现，深度卷积网络并没有表现出预期的性质，当图像中的物体发生肉眼难以辨别的微小平移时，检测置信度发生了巨大的变化。该文章的作者将造成这种现象的原因归结于网络中的降采样，当网络中拥有了具有一定步长的降采样操作后，图像必须平移降采样率的整数倍后，才会体现出平移的不变性。这也是当前卷积网络的明显缺点之一。

### 10.1.3 训练：微调还是随机初始化

当前较为通用的训练检测模型的方法是，先在ImageNet数据集上进行预训练，然后再利用自己的数据进行微调，由于ImageNet的预训练模型可以共享，很多情况下并不需要自己亲自去训练，因此这种方法可以大大缩短模型收敛的时间。

然而，ImageNet本身是分类任务，那么这种微调的方式对于检测任务来讲真的是最有利的吗？何凯明等人通过做实验得出了以下3点结论：

- (1) 虽然微调的方式可以加速模型在训练初期的收敛，但预训练与微调的时间总和与随机初始化训练的时间大致相同。使用随机初始化的方式得到的模型，其精度可以匹配微调训练得出的模型。
- (2) 微调的方式并不能有效地防止过拟合，即无法提供较好的正则化效果。
- (3) 在某些检测精度要求高、位置敏感的任务中，其数据集与ImageNet数据集会存在较大差距，这种情况下采用微调的训练方法会限制检测器的性能。

因此，ImageNet的预训练并不是必要的，如果有足够的计算资源与数据，完全可以使用随机初始训练的方法。预训练的方式可以加速训练过程，并且适用于自己的数据集规模比较小的情况，但不一定能提升最终的检测精度。

在当前的物体检测模型中，使用随机初始化进行训练的检测算法有DSOD（Deeply Supervised Object Detector）、DropBlock等，这种一阶

网络相比于两阶网络，其结构更容易端到端，因此也更容易利用随机初始化来使得模型收敛。训练结果表明，随机初始化的方法并不比微调的方法精度低。

#### 10.1.4 考虑物体间关系的检测

当前，主流的物体检测框架是以独立的物体为单元进行检测，物体之间并没有任何关联。然而在某些场景下，考虑周围物体的因素，会增加很多有用的信息。在此举一个例子，从下列两种描述中，试图判别描述的物体是什么：

·一个半椭圆形的黑色物体。

·一个半椭圆形的黑色物体，其引出了一条细线，周围有一个小方块很多的长条状物体，还有一个面积较大的矩形物体。

相比之下，通过第二个描述，我们更容易想到该物体是一个鼠标，周围的物体分别是键盘与显示器，这正是利用了周围物体的特征信息来辅助判断，相互之间可以促进彼此的识别。

为了充分利用这种信息，微软亚洲研究院提出了一个针对物体检测的物体关系网络（Relation Networks），通过特征与几何之间的交互，对物体关系进行了建模，能够有效提升检测的性能。这也证明了物体关系建模对卷积网络的有效性。

考虑特征之间的关系思想，在自然语言处理（Natural Language Processing, NLP）领域中较为常见，因为单词特征简单且相互之间依赖性很强。相比之下，图像中的物体具有不同的尺度、位置，其位置特征更为复杂。

为了对物体关系建模，Relation Networks的作者提出了一个物体关系模块（Object Relation Module），考虑物体间的几何空间，不同数量的输入之间可以并行运算，可以方便地添加到一般的特征提取模块中，

对于模块的细节这里就不展开叙述了。

值得一提的是，考虑物体间关系的方法还有利于改善NMS，可以根据需求来自适应地进行NMS，而不是固定的超参数阈值。

随着检测算法的继续沉淀与深入发展，相信会涌现出更多构建物体间关系的方法，这也有利于增进对场景的理解，从而提升检测的语义性，向更智能的方向发展，渐渐靠近人类的思维方式。

### 10.1.5 优化卷积方式

Faster RCNN和SSD等检测算法，由于其优雅的框架与出色的普适性，生命力很强大，可以广泛地应用于各种不同的场景需求中。

在学术界，近几年更多的成果也基本是建立在这几种框架之上，然后针对某些薄弱环节与特殊场景进行精修。例如Repulsion Loss针对拥挤行人设计了更优越的损失函数，IoU-Net等探索了更优的NMS设计方法，Grid R-CNN利用网络grid优化了传统的边框回归方式等。

当然，我们也看到了一些更为本质的探索与改进，例如卷积方式的改善，这里整理了当前几种不同的卷积方式，如图10.2所示。

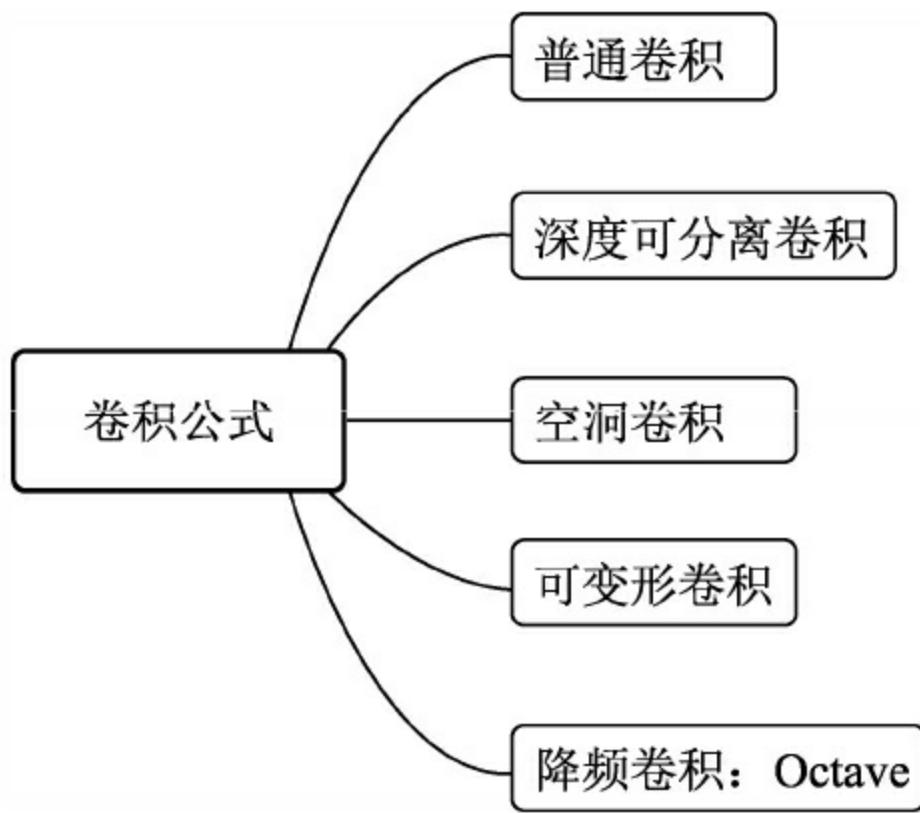


图10.2 几种不同的卷积方法

前三种卷积方式在前面的章节中已有诸多介绍，在此做一个简要总结：

- 普通卷积方法作为卷积神经网络的基础单元，可以出色地完成特征提取任务，但计算复杂度较高。

- 为了减少卷积的冗余计算，深度可分离卷积应运而生，可以有效地提升网络运行速度。

- 在物体检测中，当我们在增加网络的感受野的同时，又不想网络有太大的下采样率与计算量时，空洞卷积提供了很好的解决办法，尤其是在最新的检测算法中，空洞卷积尤为常见。

除了这三种卷积外，可变形与Octave卷积这两种新型的卷积方式，则会颠覆你对卷积网络的原有认识。

## 1. 可变形卷积

传统的卷积核一般是正方形或者长方形的，但微软亚洲研究院的学者却提出了新的卷积方式，其卷积核的形状是可以变化的，称之为可变形卷积。这种可变形卷积可以只提取感兴趣的区域，而不必是固定的矩形，这样特征会更加纯净，性能会更好。

在实现时，可变形卷积需要在传统的卷积层之前增加一层过滤器，目的是学习下一个卷积层卷积核的位置偏移量，这种实现方式移植方便，只会增加较少的计算量，但可以带来更好的检测性能。这种可变形的卷积方式更为灵活，摆脱了固定卷积的限制，应该说朝着人工智能又迈进了一步。

## 2. 降频卷积：Octave

除了可变形卷积，Octave卷积为提升卷积计算的速度又提供了强有力的工具。Octave一词本意是八音阶，在音乐里每降低八个音阶，频率就会减半，而Octave卷积的思想正是通过降低了低频率特征的尺寸，从而减少了计算量，因此取名Octave卷积。

在图像处理领域，图像可以分为描述基础背景结构的低频特征与描述快速变化细节的高频特征两部分，其中高频特征占据了主要的图像信息。

与此类似，卷积特征也可以分为高频特征与低频特征两种，Octave卷积通过相邻位置的特征共享，减小了低频特征的尺寸，进而减小了特征的冗余与空间内存的占用。

实验结果表明，Octave卷积可以即插即用，能够方便地部署到ResNet、MobileNet等多种经典的卷积结构中，并且性能会有明显的提升。很期待Octave卷积在更多的工作中展现出其强大的能力。

### 10.1.6 神经架构搜索：NAS

目前的深度卷积神经网络通常是由人手动设计实现的，在实验中，研究者们会根据自身经验，尝试多种不同的卷积核大小、层数、通道数、连接方式和融合方式等，从实验结果中寻找最优的网络配置，也诞生了如VGGNet、ResNet等经典的网络结构。

然而，手工实验调参的过程是烦琐、费时费力的一件事情，随着卷积网络的深入发展，人们更希望能够自动地搜索设计出最优的网络结构，因此神经架构搜索（Neural Architecture Search, NAS）方法应运而生。

当前较为成熟的网络结构包含了上百个卷积、池化、加乘的组件，这些组件之间并不是单纯的直接连接，相互之间会存在依赖与交叉，构成一个复杂的有向无环图（Directed Acyclic Graph, DAG），因此很难通过大量的暴力搜索来得到最优解。在此仅介绍3种典型的搜索方法，如图10.3所示。

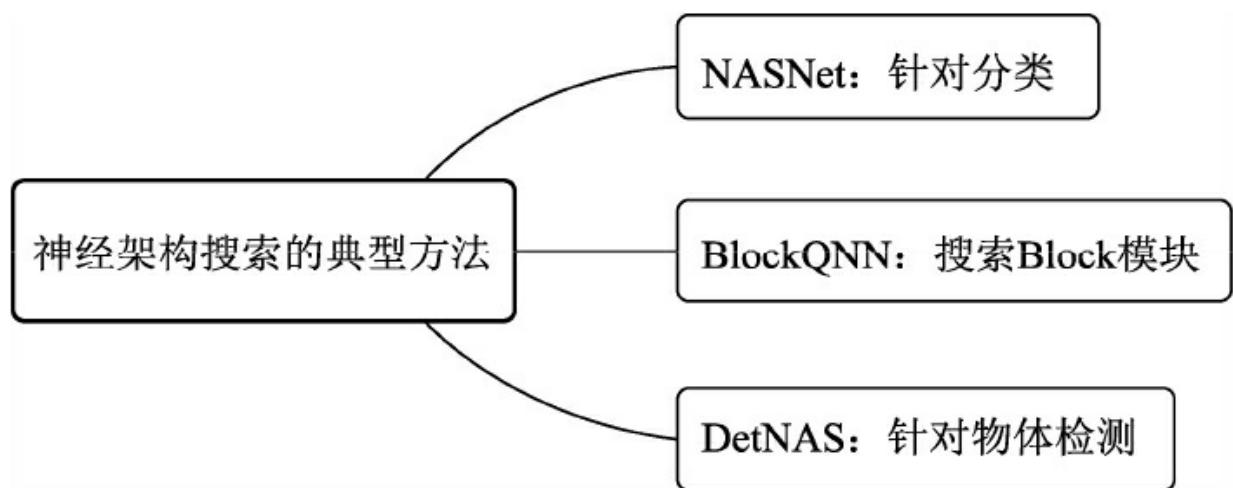


图10.3 3种神经架构搜索的典型方法

## 1. NASNet方法

由于图像分类的任务相对较为成熟，因此NAS的方法首先在图像分类任务中取得了大规模的应用，例如谷歌在CVPR 2018上提出的NASNet结构，使用了如下几种实现方式，取得了较好的效果。

- 由于ImageNet数据集较大，直接使用NAS代价太大，因此NASNet先在较小的CIFAR-10数据集上进行搜索，然后再迁移到ImageNet数据集上训练。

- NASNet的宏观网络结构还是需要人工设计，搜索空间是由多个卷积层Cell组成的，这些卷积层结构相同，权重不相同。

- NASNet使用了RNN（Recurrent Neural Network，循环神经网络）来实现卷积层Cell的自动搜索，具体来讲，控制器RNN是一个单层的LSTM（Long Short-Term Memory，长短期记忆网络），使用该搜索方法最终得到了3个最佳的结构。

## 2. BlockQNN方法

同样，在CVPR 2018会议上，商汤提出了基于分布式训练的深度增强学习算法BlockQNN，来实现自动的网络架构搜索。当前主流的深度网络通常是由多个重复的子网络来组成，即Block。BlockQNN正是借鉴了该思想，搜索Block的结构，在大大减少搜索空间的同时，也提升了网络的泛化能力。

为了更好地表示Block结构，BlockQNN首先对网络结构做了编码，将神经网络看做一个有向无环图，将每一个卷积层看做一个节点，然后对每一层的层数序号、类型、卷积核大小及两个前序节点的序号进行编码。

为了进一步减小搜索的计算量，BlockQNN利用了强化学习来获取最优的网络结构，具体使用了Q-learning算法进行网络学习。最终，BlockQNN使用了32块GPU卡进行训练，经过3天的搜索时间，在CIFAR数据集上达到了手工设计网络的精度。

### 3. DetNAS方法

近期在物体检测任务中也掀起了一股NAS的热潮，例如旷视提出的DetNAS方法，实现了物体检测器主干网络的自动搜索。

DetNAS将搜索空间编码为超网，主要包含超级网络训练、使用进化算法EA进行超级网络的搜索，以及结果网络再训练这三个步骤。DetNAS主要基于ShuffleNet v2的架构，在COCO数据集上实现了超越ResNet101的检测精度。

总体上，NAS方法是神经网络走向自动化与智能化的重要环节，当前的方法还有广阔的提升空间。网络的自动搜索可以减轻算法工程师大量的“炼丹”工作，相信未来在更广阔的应用领域里，网络的自动搜索能够大放异彩。

### 10.1.7 与产业结合的创新

当前物体检测算法的发展日新月异，不断刷新各大公开数据集的检测指标，已经达到了非常可观的地步。然而，实际的产业场景中要比公开数据集复杂得多，例如自动驾驶领域，具体体现在以下4点，如图10.4所示。

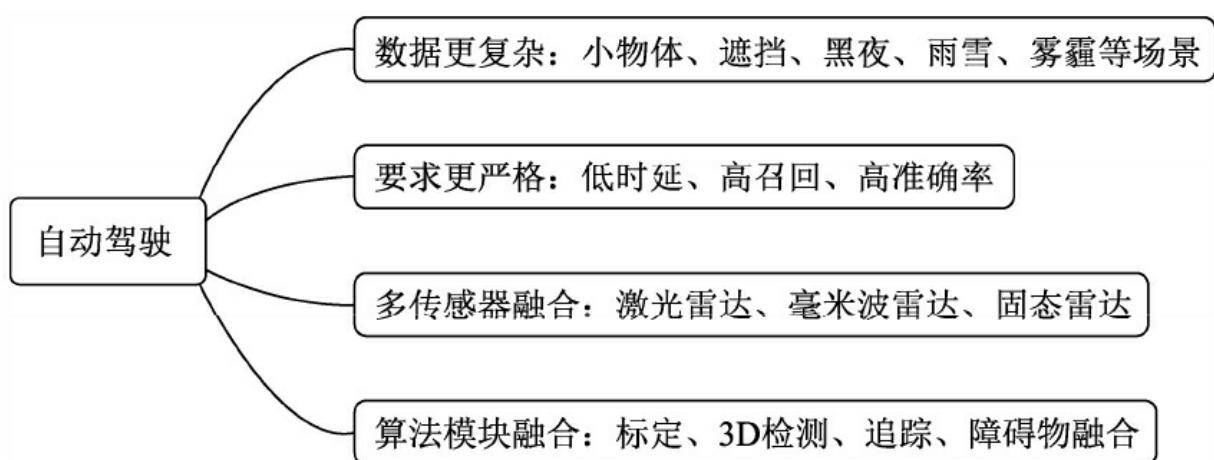


图10.4 自动驾驶场景对物体检测的要求

·**数据更复杂：**自动驾驶车辆通常配有很多不同位置的相机用来识别障碍物，其中会包含大量的遮挡、小物体等，也会存在夜晚、雨雪及雾霾等难检测的场景，这些都对检测模型提出了更高的要求。

·**指标要求严格：**安全是自动驾驶的第一原则，因此时延、检测准确率、准确率这些指标都有着极高的要求。对于很少出现的障碍物、甚至新出现的障碍物类别，如何实现稳定的检测？

·**传感器融合：**与激光、雷达相比，相机的语义信息更丰富，但是精度相对较差，因此当前成熟的解决方案通常是两种传感器的融合。此外，毫米波雷达与近两年兴起的固态雷达，在某些场景下也都有使用，

因此，多传感器算法的融合也对2D物体检测算法提出了新的要求。

·算法融合：对于自动驾驶车辆，其最终的检测对象应该是具有时序性质的3D世界坐标系下的障碍物信息，2D的物体检测仅仅是算法中的一环，还包括了标定、3D检测、追踪和障碍物融合等。因此，如何将该领域中的其他算法模块与物体检测相融合，也是一个亟待解决的方向。

简而言之，从产业应用的角度来看，物体检测与其他算法相融合，是一个更为有效的思路。笔者在此列举了4个可以进行算法融合的方面，如图10.5所示。

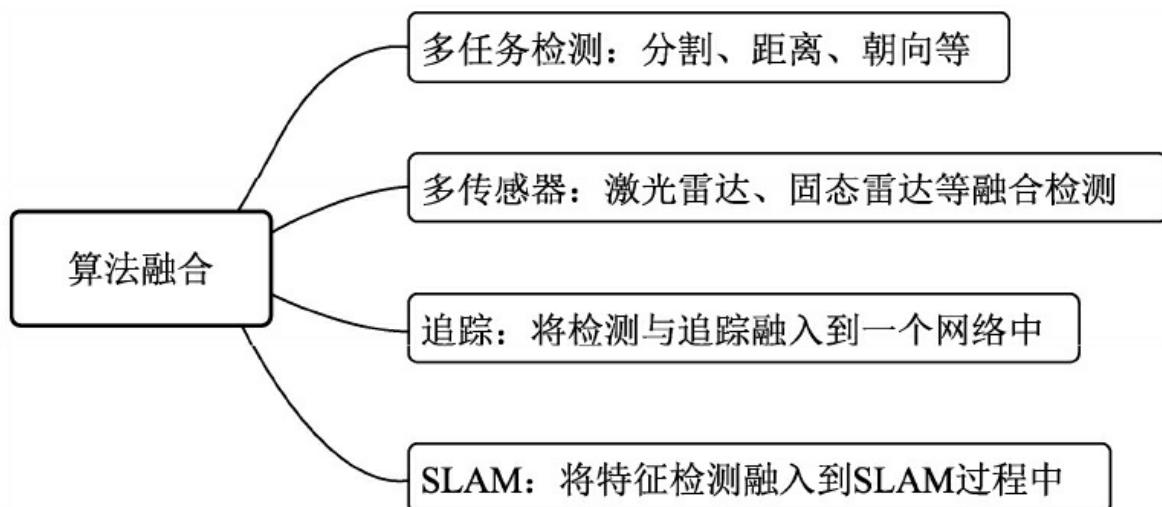


图10.5 自动驾驶领域算法的融合思路

·多任务融合：对于可行驶区域、形状多变的对象来讲，只靠2D的物体检测是远远不够的，还需要更为精细的图像分割算法。此外2D的检测还包括车是否在路上、尾灯判断等多个任务。然而，在一个车载单元上运行多个单独的模型是十分奢侈的，如何用一个网络实现多个任务的学习十分重要。

·多传感器的融合：自动驾驶领域通常采用了激光雷达、相机等多

种传感器来充分保障行驶的安全，当前更多的传感器数据处理方法还是相互独立的，因此如何更好地用神经网络实现多个传感器数据的检测，也是一个值得探索的方向。

·与追踪的融合：追踪是物体检测的下游算法模块，通常也是将检测与追踪独立成两个模块处理，当前已有多种基于神经网络的追踪算法，因此这两者也完全可以实现更好的结合。

·与定位建图的结合：实时定位与地图构建（*Simultaneous Localization and Mapping, SLAM*）算法模块是自动驾驶中非常重要的一环，可以为自动驾驶车辆提供精确的地图与定位信息。由于其中包含图像或者点云的特征提取，因此完全可以将卷积神经网络的优势引入到该算法中。

从发展轨迹来看，人工智能技术与自动驾驶产业近几年经历了极为迅速的技术迭代与产业变革，可以称得上是相互成全，对于未来的发展，我们拭目以待。

## 10.2 摆脱锚框：Anchor-Free

在物体检测领域中，Anchor通常被看做是一组不同大小形状的先验框，由于其具有启发性的先验信息，在Faster RCNN和SSD等框架中发挥了巨大的作用，基于Anchor的算法也占据了物体检测的大半“江山”。

然而，近一两年来，不依赖锚框的检测算法如雨后春笋般地涌现了出来，并且达到了与依赖锚框（Anchor-Based）算法相同甚至更好的检测效果。

因此，本节将详细地分析Anchor对检测模型的影响，并重点介绍多种摆脱锚框的检测算法。

### 10.2.1 重新思考Anchor

回顾一下物体检测算法，在发展的初始阶段是没有Anchor这个概念的，例如Fast RCNN使用随机搜索（Selective Search）的方法提取感兴趣区域，虽然共享了卷积，但耗时严重，正是Faster RCNN引入了Anchor作为先验框，才将实时的物体检测变为可能，达到了检测的第一个高峰，其中Anchor可以说是居功至伟。

如图10.6是当前主流的基于Anchor的检测算法中边框的变化，对于二阶的算法，第一阶段RPN会对Anchor进行有效的筛选，生成更有效、精准的Proposal，送入第二个阶段，最终得到预测的边框。相比之下，一阶的算法相当于把固定的Anchor当做了Proposal，通过高效的特征与正、负样本的控制，直接预测出了物体。



图10.6 检测算法中边框的变化

然而，随着检测性能的提升，Anchor的弊端也渐渐地暴露了出来，使用Anchor通常会面临如下3个问题：

- 正、负样本不均衡：我们通常在特征图所有点上均匀采样Anchor，而在大部分地方都是没有物体的背景区域，导致简单负样本数量众多，这部分样本对于我们的检测器没有任何作用。

·超参难调：Anchor需要数量、大小、宽高等多个超参数，这些超参数对检测的召回率和速度等指标影响极大。此外，人的先验知识也很难应付数据的长尾问题，这显然不是我们乐意见到的。

·匹配耗时严重：为了确定每个Anchor是正样本还是负样本，通常要将每个Anchor与所有的标签进行IoU的计算，这会占据大量的内存资源与计算时间。

以这3个问题为出发点，近期出现了大量Anchor-Free的算法。Anchor-Free的思想最早可见于2015年的DenseBox，以及前面讲解过的YOLO。在各种Anchor-Free的算法中，根据其表征一个物体的方法，大体可以分为以下两类，如图10.7所示。

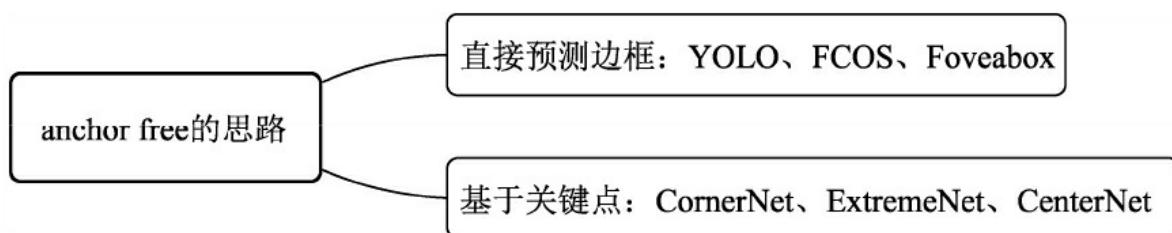


图10.7 摆脱锚框的物体检测算法思路

·直接预测边框：根据网络特征直接预测物体出现的边框，即上、下、左、右4个值。典型算法如YOLO、自动学习在FPN上分配标签的FSAF（Feature Selective Anchor-Free）、利用分割思想解决检测的FCOS（Fully Convolutional One-Stage），以及改善了边框回归方式的Foveabox算法。

·关键点的思想：使用边框的角度或者中心点进行物体检测，这类算法通常是受人体姿态的关键点估计启发，开辟了物体检测算法的一个新天地，典型有CornerNet、ExtremeNet及CenterNet等。

从实验结果来看，Anchor-Free的算法已经达到了与基于Anchor的检测器相同甚至更好的检测效果，虽然距离两阶的基于Anchor的检测器还有些差距，完全取代Anchor仍有一段距离。当然，抛弃Anchor这个先验知识，还能把物体检测做到如此高的水平，也依赖与FPN和Focal Loss等当前公认的优秀经验。

为了更好地理解Anchor-Free的检测思想，本节选取了CornerNet、CenterNet及Guided Anchoring这3个典型的算法进行详细的讲解。

## 10.2.2 基于角点的检测：CornerNet

由于Anchor会带来较多的超参数与正、负样本的不均衡，发表于ECCV 2018的CornerNet算法另辟蹊径，舍弃了传统Anchor与区域建议框的检测思路，利用关键点的检测与匹配，出色地完成了物体检测的任务。

CornerNet的思路实际是受多人体姿态估计的方法启发。在多人体姿态估计领域中，一个重要的解决思路是Bottom-Up，即先使用卷积网络检测整个图像中的关键点，然后对属于同一个人体的关键点进行拼接，形成姿态。

CornerNet将这种思想应用到了物体检测领域中，将传统的预测边框思路转化为了预测边框的左上角与右下角两个角点问题，然后再对属于同一个边框的角点进行组合，整体网络结构如图10.8所示。

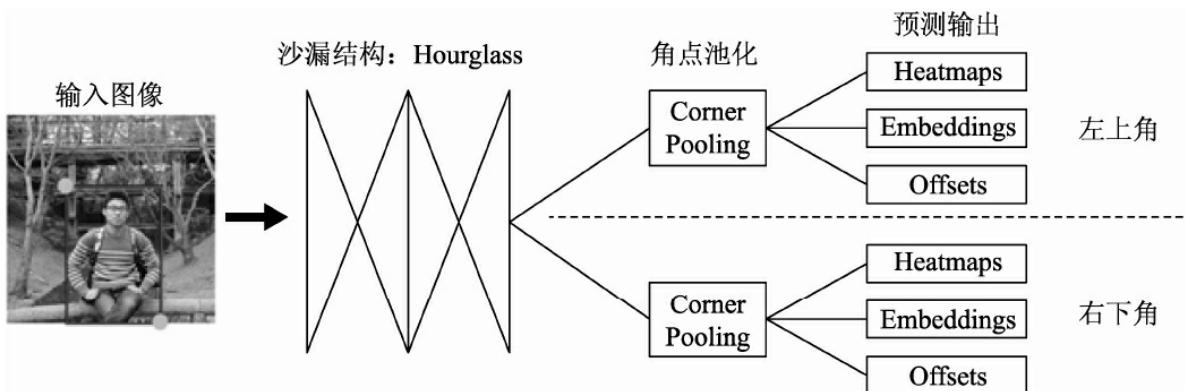


图10.8 CornerNet算法的整体框架

从图10.8中可以看出，CornerNet的主要结构主要由以下3部分组成：

- 沙漏结构Hourglass：特征提取的Backbone，能够为后续的网络预

测提供很好的角点特征图。

·角点池化：作为一个特征的池化方式，角点池化可以将物体的信息整合到左上角点或者右下角点。

·预测输出：传统的物体检测会预测边框的类别与位置偏移，而CornerNet则与之完全不同，其预测了角点出现的位置Heatmaps、角点的配对Embeddings及角点位置的偏移Offsets。

为了很好地理解CornerNet，下面分别详细介绍这3个模块。

### 1. 沙漏结构：Hourglass

为了提取图像中的关键点，CornerNet使用了沙漏结构Hourglass作为网络特征提取的基础模块，其结构如图10.9所示。顾名思义，Hourglass的整体形状类似于沙漏，两边大，中间小。Hourglass结构是从人体姿态估计领域中借鉴而来，通过多个Hourglass模块的串联，可以十分有效地提取人体姿态的关键点。

在图10.9中，左半部分表示传统的卷积与池化过程，语义信息在增加，分辨率在减小。右半部分表示上采样与融合过程，深层的特征通过上采样操作与浅层的特征进行融合，在增大分辨率的同时，保留了原始的细节信息。

CornerNet首先通过一个步长为2的 $7 \times 7$ 卷积层，以及步长为2的残差模块，将图像尺寸缩小为原图的 $1/4$ ，然后将得到的特征图送入两个串联的Hourglass模块。

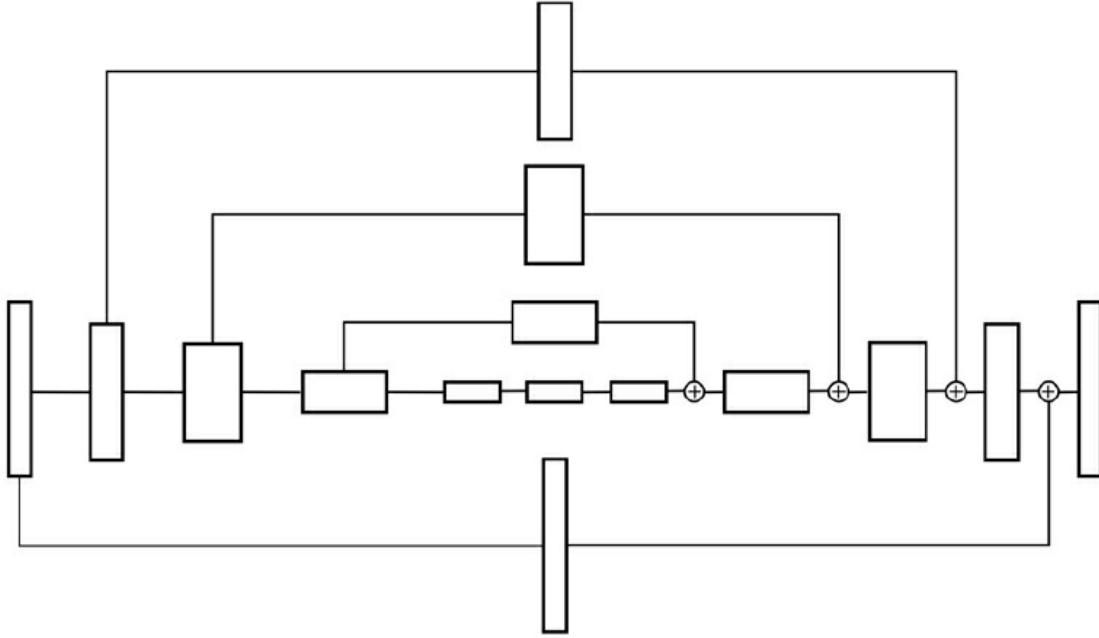


图10.9 Hourglass模块结构示意图

## 2. 角点池化: Corner Pooling

在传统卷积网络中，通常使用池化层来进行特征融合，扩大感受野，也可以起到缩小特征图尺寸的作用。以 $3\times 3$ 的最大池化层为例，通常是以当前位置的点为中心点，融合周围共9个点的信息，取最大值输出。

然而，CornerNet的思想是利用左上与右下两个关键点进行物体检测，对于一个物体的左上点，其右下区域包含了物体的特征信息，同样对于物体的右下点，其左上区域包含了物体的特征信息，这时角点的周围只有四分之一的区域包含了物体信息，其他区域都是背景，因此传统的池化方法就显然不适用了。

为了达到想要的池化效果，CornerNet提出了Corner Pooling的方法，左上点的池化区域是其右侧与下方的特征点，右下点的池化区域是其左侧与上方的特征点，如图10.10所示为左上点的Corner Pooling过

程。

在图10.10中，假设当前点的坐标为 $(x,y)$ ，特征图宽为 $W$ ，高为 $H$ ，则Corner Pooling的计算过程如下：

(1) 计算该点到其下方所有点的最大值，即 $(x,y)$ 到 $(x,H)$ 所有点的最大值。

(2) 计算该点到其最右侧所有点的最大值，即 $(x,y)$ 到 $(W,y)$ 所有点的最大值。

(3) 将两个最大值相加，作为Corner Pooling的输出。

工程实现时，可以分别从下到上、从右到左计算最大值，这样效率会更高。右下点的Corner Pooling过程与左上点类似，在此不再展开。

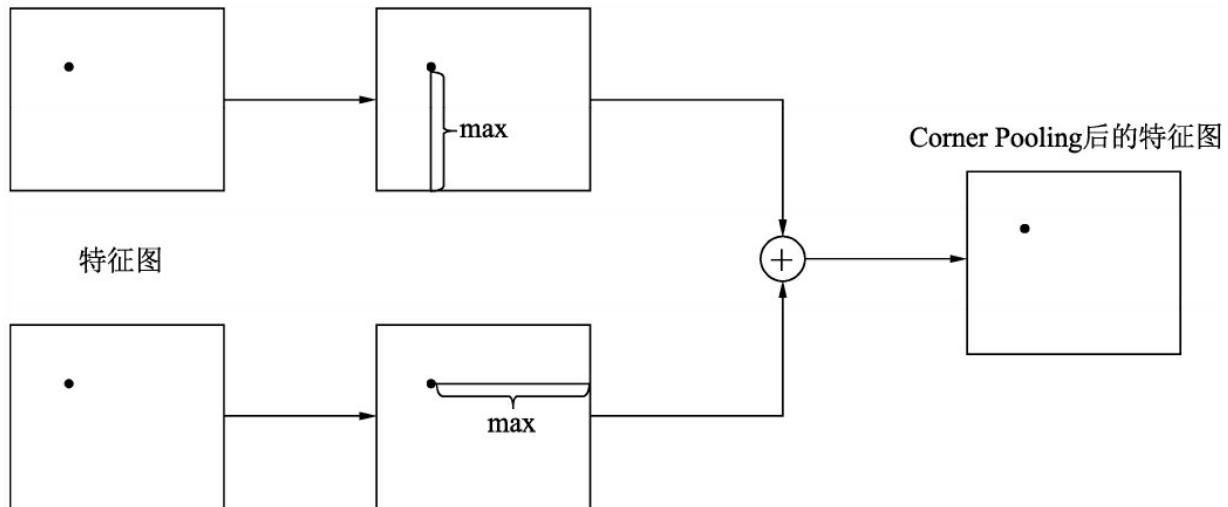


图10.10 Corner Pooling方法示意图

### 3. 预测输出

在介绍完网络的整体结构后，最后一个重要的部分就是CornerNet的预测输出，以及损失的计算方式。如图10.8所示，左上角与右下角两个

Corner Pooling层之后，分别接了3个预测量，这3个预测量的意义分别如下：

·**Heatmaps**: 角点热图，预测特征图中可能出现的角点，大小为  $C \times W \times H$ ， $C$  代表类别数，以左上角点的分支为例，坐标为  $(c, x, y)$  的预测点代表了在特征图上坐标为  $(x, y)$  的点是第  $c$  个类别物体的左上角点的分数。

·**Embeddings**: Heatmaps 中的预测角点都是独立的，而一个物体需要一对角点，因此 Embeddings 分支负责将左上角点的分支与右下角点的分支进行匹配，找到属于同一个物体的角点，完成检测任务，其大小为  $1 \times W \times H$ 。

·**Offsets**: 第三个预测 Offsets 代表在取整计算时丢失的精度，以进一步提升检测的精度。关于为何会丢失精度，在前面章节的 ROI Pooling 部分已有详细说明。这种取整的丢失对于小物体检测影响很大，因此 CornerNet 引入了偏差的预测来修正检测框的位置，其大小为  $2 \times W \times H$ 。

CornerNet 在损失计算时借鉴了 Focal Loss 的思想，对于不同的负样本给予了不同的权重，总体的损失公式如式（10-1）所示。

$$L = L_{det} + \alpha L_{pull} + \beta L_{push} + \gamma L_{off} \quad (10-1)$$

这4部分损失的含义说明如下：

· $L_{det}$ : 角点检测的损失，借鉴了 Focal Loss 权重惩罚的思想。CornerNet 为了减小负样本的数量，将以标签角点为中心，半径为  $r$  区域内的点都视为正样本，因为这些点组成的边框与标签会有很大的 IoU，仍有可能是我们想要的正样本。

· $L_{pull}$ : Embeddings 中，对于属于同一物体的两个角点的惩罚。具体

实现时，提取Embeddings中属于同一个物体的两个角点，然后求其均值，并希望两个角点的值与均值的差尽可能地小。

· $L_{push}$ : Embeddings中，对不属于同一物体的两个角点的惩罚。具体实现时，利用 $L_{pull}$ 中配对的角点的平均值，期望没有配对的角点与该平均值的差值尽可能地大，可以有效分离开无效的角点。

· $L_{off}$ : 位置偏差的损失，与Faster RCNN相似的是，CornerNet使用了smooth<sub>L1</sub>损失函数来优化这部分位置偏差。

受篇幅限制，CornerNet的其他细节就不具体介绍了。总体上，CornerNet巧妙地利用了一对关键点来实现物体检测，从而避免了Anchor带来的问题，在检测精度上相比其他单阶检测器有了一定提升。此外，CornerNet的工作也推动了一系列利用关键点做物体检测的算法的诞生。

### 10.2.3 检测中心点：CenterNet

CornerNet虽然实现了利用关键点做物体检测，但其需要两个角点进行匹配，匹配过程耗时较长，并且容易出现误检的样本。于是我们进一步思考，既然两个角点来做物体检测容易带来误检，使用一个关键点做检测是否可行？

诞生于2019年的CenterNet算法成功地实现了这种思想，将物体检测问题变成了一个关键点的估计问题，通过预测物体的中心点位置及对应物体的长与宽，实现了当前检测精度与速度最好的权衡，整体结构相当地优雅。

#### 1. CenterNet思想与网络

如图10.11对比了传统的基于Anchor及CenterNet的检测思想，左边的图代表了基于Anchor的检测，常见做法是将Anchor与所有样本计算重叠的IoU，大于一定IoU阈值的作为正样本，小于阈值的视为负样本。

图10.11的右边图像代表了CenterNet的思想，与基于Anchor的算法相比，有如下3点区别：

- 没有使用Anchor作为先验框，而是预测物体的中心点出现位置，因此也就不会存在先验框与标签的匹配，正、负样本的筛选过程。
- 每个物体标签仅仅选择一个中心点作为正样本，具体实现是在关键点热图上提取局部的峰值点，因此也就不会存在NMS的过程。
- 由于CenterNet专注在关键点的检测，因此其可以使用更大的特征图，而无须使用多个不同大小的特征图。在CenterNet的论文中其使用的网络下采样率为4。

CenterNet尝试了串联Hourglass、ResNet等多种网络用来提取特征，生成了特征点的热图。实验结果表明，Hourglass的网络能够提供更精确的检测精度，而更轻量的ResNet的检测速度会更快。

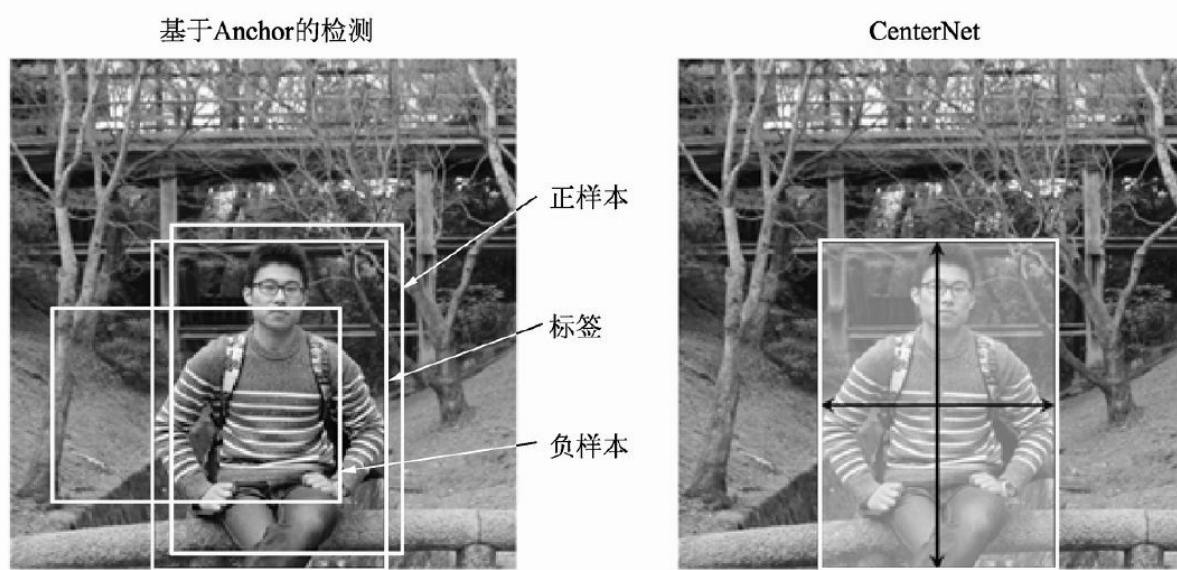


图10.11 基于Anchor的检测与CenterNet的思想对比

## 2. 网络预测与损失计算

CenterNet参考了CornerNet的思想，网络输出了以下3个预测值：

·关键点热图：这里的关键点热图与CornerNet类似，只是这里只预测一个中心点的位置。对于标签的处理，CenterNet将标签进行下采样，然后通过式（10-2）的高斯核函数分散到热图上。

$$Y_{xyc} = \exp\left(-\frac{(x - \bar{p}_x)^2 + (y - \bar{p}_y)^2}{2\sigma_p^2}\right) \quad (10-2)$$

·中心点偏差：CenterNet对每个中心点增加了一个偏移的预测，并且所有类别共享同一个偏移预测值。

·宽与高的预测：CenterNet不需要预测Embeddings来处理配对，而是预测了物体的宽与高，这里的预测是原图像素坐标的尺度。

总体上，对于特征图上的一个点，CenterNet会预测C+4个值，其中包括C个类别的中心点得分、中心点(x,y)的偏差以及该物体的宽高(w,h)。

式（10-3）代表CenterNet的整体损失函数：

$$L_{det} = L_k + \lambda_{size} L_{size} + \lambda_{off} L_{off} \quad (10-3)$$

其中， $L_k$ 为关键点的损失，使用了Focal Loss的形式； $L_{size}$ 为宽与高的预测损失， $L_{off}$ 为偏移预测的损失； $L_{size}$ 与 $\lambda_{off}$ 是为了平衡各部分损失而引入的权重。

CenterNet最强大的一点在于其不仅可以做2D的物体检测，还可以方便地实现人体姿态估计、3D检测等任务，如图10.12所示。

与2D检测不同，单目3D的物体检测需要预测物体相对于当前相机的距离、物体的长宽高及朝向信息，这些可以在CenterNet网络的基础上通过增加Head分支来实现，相比起其他3D物体检测，CenterNet的实现更为简洁快速。3D的物体检测一个非常重要的应用领域就是自动驾驶，车辆需要感知的是障碍物在3D空间中的信息来进行路径规划、避障，仅靠2D检测是远远不够的。

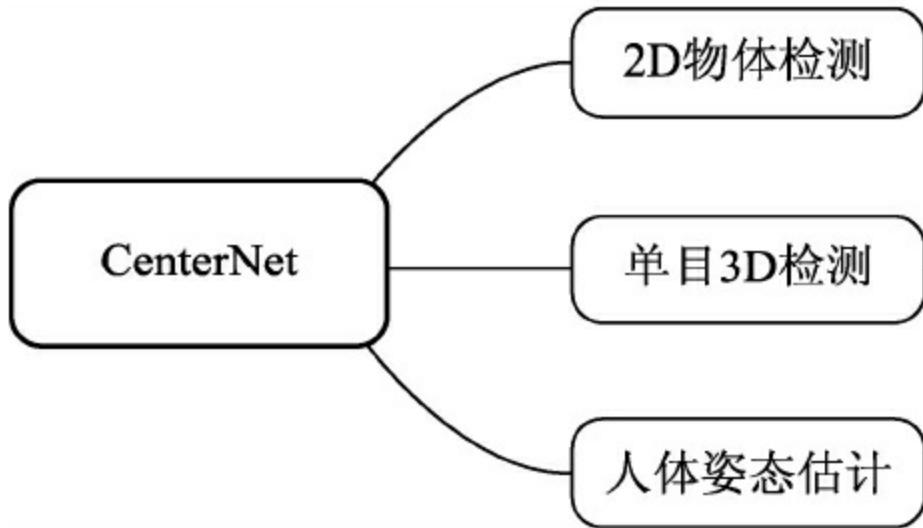


图10.12 CenterNet算法可以应用的领域

对于人体姿态估计，CenterNet的做法是将每一个关节点作为中心点的偏移量，直接在中心点位置回归出这些偏移量的值。简而言之，CenterNet可以说将基于中心点做检测的思想发挥到了一定的高度，并且由于其实现方法简洁、优雅，相比起基于Anchor的算法，CenterNet也更容易部署到工程应用中。

#### 10.2.4 锚框自学习：Guided Anchoring

物体检测技术的发展过程中，Anchor发挥了巨大的作用，但如今其带来的负面影响渐渐制约了检测器更高的性能。CornerNet与CenterNet两种算法利用关键点有效避开了Anchor的使用，取得了优异的成果。

但是深入思考，Anchor的弊端是超参的人工设计与正负样本不均衡，并且无法很好地处理极端大小宽高物体，如果能自动地设计出更加优越高效的Anchor，也是一种解决思路。基于此思想，商汤提出了锚框自学习（Guided Anchoring）的算法，根据图像的特征能够自动地预测Anchor的位置和形状，生成一组稀疏但高效的Anchor，全程无须人工的设计。

Guided Anchoring的网络框架如图10.13所示，主要分为锚框预测与特征自适应两部分，其中锚框预测部分负责预测Anchor出现的位置与形状，而特征自适应模块可以将Anchor作用到特征图上，从而提取出更有针对性的特征。

下面将分别详细介绍这两个模块。

##### 1. 锚框预测：Anchor Generation

在传统算法中，Anchor可以被看做是均匀分布在整个特征图上，而实际物体在图像上的分布表达式则类似于冲激函数，稀疏且形状不定。在此，Guided Anchoring利用网络特征自动地预测了Anchor的分布，其中包含Anchor出现的位置与形状两个特征。

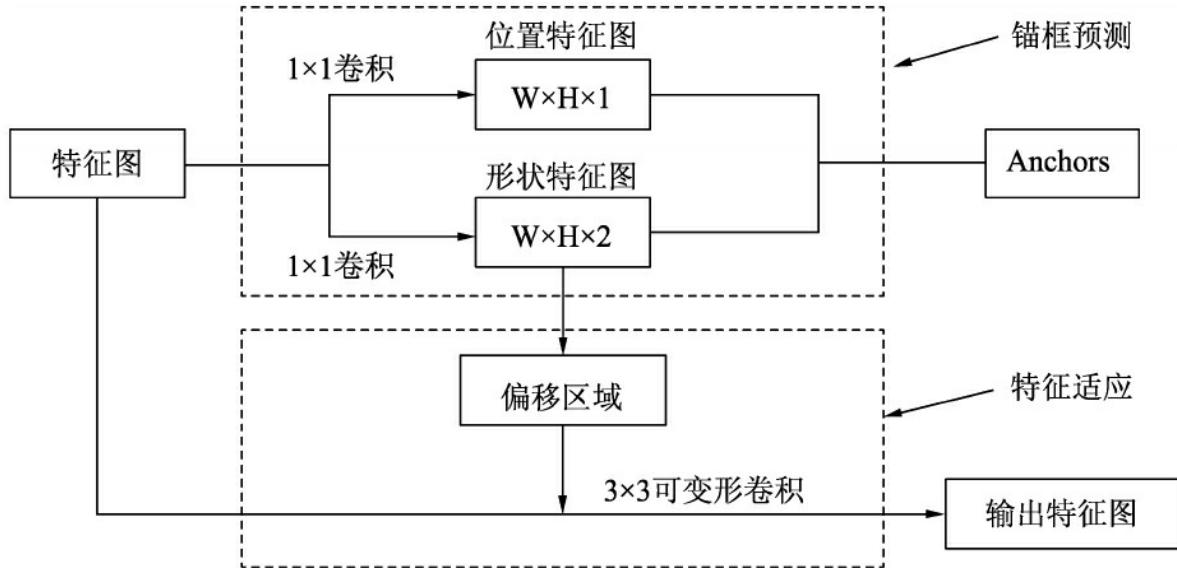


图10.13 Guided Anchoring算法框架图

具体地，Guided Anchoring使用了与CenterNet预测物体类似的思想，采用了两个分支来预测Anchor。

·中心点位置：使用了一个 $1 \times 1$ 卷积来预测Anchor可能出现的位置，这里的输出大小为 $W \times H \times 1$ ，每个点表示Anchor的中心点出现在此位置的概率。

·形状预测：使用了一个 $1 \times 1$ 卷积来预测Anchor的宽与高两个特征，输出大小为 $W \times H \times 2$ 。

在计算损失时，Guided Anchoring采用了IoU作为衡量指标，希望IoU最大化。需要注意的是，由于缺乏先验的边框，我们没有办法通过先验框与标签匹配的方法来确定优化的标签对象，因此对于一个预测Anchor，它应该去优化与哪一个标签的IoU？

为了解决这个问题，Guided Anchoring在每一个点上采样了9组不同宽高的边框，来取代预测的Anchor，进而完成匹配关系，确定优化对象。

## 2. 特征自适应：Feature Adaption

在主流的一阶网络中，Anchor的大小与形状在各个位置上都是相同的，因此可以在特征图上使用固定的卷积提取特征，以表征每一个Anchor的特征。

然而，Guided Anchoring中的Anchor形状不一，特征图事先并无法得到其需要预测的Anchor的形状，但是后续却要预测该Anchor的类别与位置偏移，这里就存在Anchor与表征Anchor的特征不匹配的问题。

为了解决该问题，Guided Anchoring在Anchor预测后增加了一个特征自适应层（Feature Adaption），如图10.7所示。具体实现过程中，Feature Adaption使用了一个 $3 \times 3$ 可变形卷积作用于特征图上，以适配Anchor的形状。

与普通可变形卷积不同，这里的偏置取自于预测的Anchor，具体是利用一个 $1 \times 1$ 卷积作用于预测的Anchor宽高，实现极为巧妙。从功能上理解，这里的特征自适应有些类似于RoI Pooling层。

Guided Anchoring的总体损失函数如式（10-4）所示，在常规的分类损失 $L_{cls}$ 与回归损失 $L_{reg}$ 之外，还包括Anchor的位置损失 $L_{loc}$ 与形状损失 $L_{shape}$ 。

$$L = \lambda_1 L_{loc} + \lambda_2 L_{shape} + L_{cls} + L_{reg} \quad (10-4)$$

在实验结果上，Guided Anchoring对于当前基于Anchor的一阶网络与二阶网络都有明显的精度提升，而网络上仅仅增加了3个 $1 \times 1$ 卷积及一个可变形卷积，对速度影响并不大，可谓十分高效。

## 10.3 总结

相信读者读到这里，对卷积神经网络与物体检测算法有了更深的理解，但这并不是学习的结束，而是新的开始。本书的初衷是对物体检测算法做一个全面而深入的总结，希望能够帮助到入门者与初学者。也希望本书能作为一个工具书，当你想要了解某个方法、某个实现细节时，可以速查获取。

受限于篇幅，本书无法对所有的检测算法面面俱到，做到代码级的讲解。对于后续的学习，笔者有以下5点建议：

·如果对书中的某些算法感兴趣，可以上arXiv官方网站搜索该论文进行细读，然后在GitHub网站搜索相应的实现方法，一般都能找到，代码级的学习才能使你掌握检测方法的精髓与脉络。当然，基础比较扎实后可以尝试自己复现论文，这也会带给你很多新的启发与收获。

·实践出真知：不要长期停留在论文与方法论上，尝试将算法应用到实际工程中，或者参加COCO、Kaggle、天池之类的比赛，相信你也会在工程与比赛中收获很多乐趣。

·多记录：物体检测算法众多，每年都会涌现出大量的新算法，因此建议读者在日常读论文、写代码时养成记录的习惯，可以是备忘录、博客、笔记的形式。通过长期的积累，也有益于形成自己的知识体系。

·对于PyTorch的学习，其官方文档思路清晰而简洁，建议读者通读一下。与此同时，PyTorch官方论坛也较为完备，初学者遇到的PyTorch问题通常都可以在上面找到答案。

·当前物体检测等深度学习算法的门槛相对较低，如果认为读了几

篇文章，看几个成熟框架的代码，就掌握了算法精髓，这种想法是极片面的。实际上，计算机视觉有非常丰富的知识体系，如图像去噪、增广和视觉几何等，相信读者在扩展了这些知识后，也会启发深度学习算法的创新。

种一棵树最好的时间是十年前，其次是现在。希望大家享受本书及物体检测带来的快乐！