第4章 图像分类

图像分类是计算机视觉最基础的一个任务，也是几乎所有的基准模型都要进行比较的任务。从最初比较简单的 10 个分类的灰度图像手写数字识别任务 MNIST，到后来更大一点的 10 个分类的 CIFAR10 和 100 个分类的 CIFAR100 任务，再到后来的 ImageNet 任务和 WebVision 任务，图像分类模型伴随着数据集的增加，一步一步提升到了今天的水平。现在，在 ImageNet 这样的超过 1000 万张图像和 2 万个类的数据集中，计算机的图像分类水准已经超过了人类。

本章要点

- 4.1 节讲述图像分类的基础，包括图像分类的概念与划分，图像分类经典模型与评估方法。
- 4.2 节讲述多标签图像分类经典模型。
- 4.3 节讲述细粒度图像分类经典模型。
- 4.4 节讲述半监督与无监督图像分类经典模型。
- 4.5 节讲述其他图像分类问题的典型难点与解决方案。
- 4.6 节将讲述一个基础的图像分类任务，以一个简单的基准模型为例，展示图像分类的完整流程。
- 4.7 节讲述一个细粒度级别的图像分类任务，比较一般的模型与细粒度模型的精度在相同任务中的表现，验证细粒度模型设计的重要性。

4.1 图像分类基础

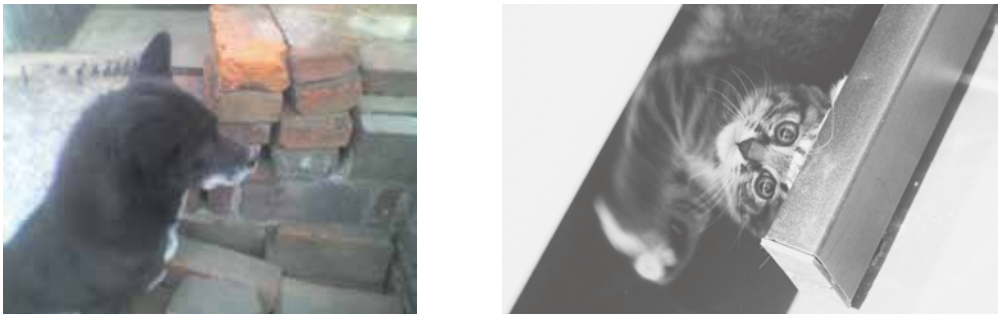
图像分类顾名思义就是一个模式分类问题，它的目标是将不同的图像划分到不同的类别下，实现最小的分类误差。本节将介绍图像分类问题的种类，以及当前基于深度学习的图像分类发展水平。

4.1.1 单标签图像分类问题

当图像分类的结果为唯一的类别时，称为单标签图像分类问题，根据图像分类的不同粒度，可以分为跨物种语义级别的图像分类、子类细粒度图像分类和实例级图像分类 3 个大方向。

1. 跨物种语义级别的图像分类

所谓跨物种语义级别的图像分类，是在不同物种的层次上识别不同类别的对象，比较常见的如猫、狗分类等，如图 4.1 所示。这样的图像分类，各个类别因为属于不同的物种或大类，往往具有较大的类间方差，而类内则具有较小的类内误差。



狗　　猫

图 4.1　跨物种语义级别的图像分类

以 Kaggle 提供的猫狗分类竞赛数据集为例，它包含猫和狗图片各 12 500 幅，这是一个非常典型的跨物种语义级别的图像分类问题。它的特点是类间相似性低、方差大，类内相似性高、方差小，属于粗粒度的图像分类。

2. 子类细粒度图像分类

细粒度图像分类相对于跨物种的图像分类粒度更小一些。它往往是在同一个大类中的子类的分类，如不同猫的分类，如图 4.2 所示。

虎斑猫　　橘猫

图 4.2　子类细粒度图像分类

以不同鸟类的细粒度分类任务为例，比较著名的数据集是 Caltech-UCSD Birds-200-

2011[1]。这是一个包含 200 个类、11 788 张图像的鸟类数据集，每一张图像提供了 15 个局部区域位置和 1 个标注框，还有语义级别的分割图。在该数据集中，以 Woodpecker 数据集为例，总共包含 6 类，即 American Three toed Woodpecker、Pileated Woodpecker、Red bellied Woodpecker、Red cockaded Woodpecker、Red headed Woodpecker 和 Downy Woodpecker，这些种类的鸟的纹理形状都很像，想要区分，只能靠头部的颜色和纹理。因此要想训练出这样的分类器，就必须让分类器能够识别到这些区域，这是比跨物种语义级别的图像分类更难的问题。

另外还有一些比较经典的相似任务，例如李飞飞实验室的数据集 Stanford Dogs[2]，它包含 120 种不同狗的图像数据，共 20 580 张图，每一个主体都提供了一个标注框。

同样来自李飞飞实验室的 Stanford Cars[3]，提供了 196 种品牌的车辆数据，共 16 185 张图像，其中，8144 张图像为训练数据，8041 张图像为测试数据。每个类别都按照年份、制造商和型号进行区分并且提供了标注框。对于不同车型的分类，在安防监控中的需求非常大。

以上数据集都是用于评测细粒度分类算法的基准，子类细粒度图像分类的研究目前仍然有很大的发展空间。

在 ImageNet1000 类中，既包括不同物种，如车与猫，也包括同一物种的不同子类，如不同类别的猫，ImageNet1000 类的分类任务的难度介于这两者之间。

除了以上数据集之外，还有一些与细粒度分类任务相关的国际比赛。2011 年在 CVPR 大会上举办了第一届 FGVC（Fine-Grained Visual Categorization，细粒度视觉分类工作组）研习会，专门研究细粒度分类的问题。之后，在 2013 年、2015 年又分别举办了第 2 次和第 3 次比赛，到 2017 年已经举办了 4 届比赛。

随着计算机视觉的快速发展和普通分类问题的研究趋于饱和，再加上 CVPR 和 FGVC 影响力的增加，从 2017 年开始 FGVC 从两年一次的比赛改为了一年一次的比赛，并且从 2017 年开始，FGVC 拆分为 iNaturalist 与 iMaterialist 两个单元。

2018 年的 FGVC 比赛包括植物、家具分类挑战和产品图像的时尚属性挑战，其中，植物分类比赛包含 8000 多种植物、动物和真菌类别，拥有超过 45 万张训练图像。商品类的分类比赛多来自工业界的赞助，因为这些分类的技术落地对于相关的公司具有很大的价值。另外还有一系列规模较小但仍然重要的挑战如 iWildCamp、iFood，这是以食物和艺术为内容的识别挑战。

与通用的图像分类数据集如 ImageNet 不同的是，细粒度分类任务 iNaturalist 挑战中的数据集呈现长尾分布，许多种类的图像非常少。因此一个好的细粒度模型必须能够处理长尾类别，这也符合在自然世界中严重不平衡的类别分布。

总之，细粒度的分类问题仍然是一个需要广泛研究的问题。

3. 实例级图像分类

如果我们要区分不同的个体，而不仅仅是物种类或者子类，那么就是一个识别问题，或者说是实例级别的图像分类，最典型的任务就是身份识别，如图 4.3 中的猫脸识别。

实例级别的任务具有较大的挑战性，如人脸识别一直是计算机视觉的重大课题，它可以用于考勤和支付等任务中。虽然经历了几十年的发展，但仍然没有被完全解决，存在年龄变化大、妆造污染、大姿态与遮挡等经典难题，读者可以参考更多资料去学习。

猫身份识别：墨墨

图 4.3　实例级图像分类

4.1.2　深度学习图像分类经典模型

图像分类是计算机视觉的基础任务，从传统的方法到基于深度学习的方法，经历了几十年的发展。本书主要关注深度学习领域的进展，许多经典数据集和模型都见证了图像分类任务的算法演变，下面我们重点讲述几个重要的节点。

1. MNIST 与 LeNet5

在计算机视觉分类算法的发展中，MNIST 是首个具有通用学术意义的基准。这是一个手写数字的分类标准，包含 6 万个训练数据，1 万个测试数据，图像均为灰度图，通用的版本大小为 28 × 28 像素。

在 20 世纪 90 年代末到 21 世纪初，支持向量机和 k 最近邻方法使用得比较多，以 SVM 为代表的方法可以将 MNIST 分类错误率降低到 0.56%，彼时仍然超过以神经网络为代表的方法，即 LeNet 系列网络。LeNet 网络诞生于 1994 年，后经过多次的迭代才有了 1998 年的 LeNet5，是为业界人士所广泛知晓的版本。如图 4.4 所示为 LeNet5 的经典结构，图片来源于 Lecun 等人在 1998 年发表的文章 *Gradient-based learning applied to document recognition*[4]。

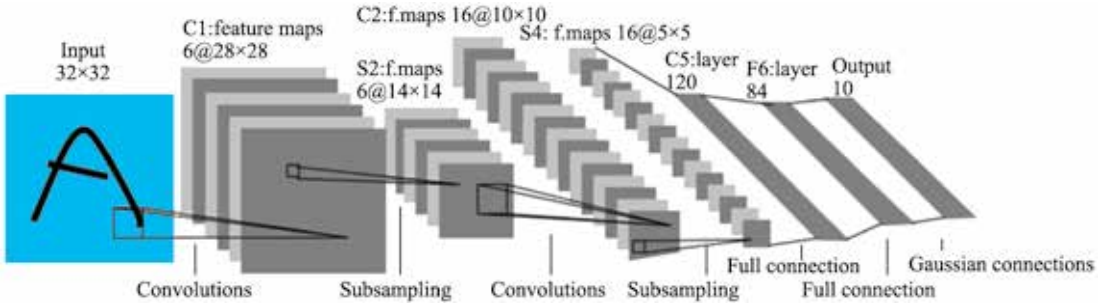


图 4.4　LeNet5 网络结构

LeNet5 是一个经典的卷积神经网络，它包含一些重要的特性，这些特性仍然是现在 CNN 网络的核心。

- 典型的卷积模块由卷积、池化、非线性激活函数依次串接构成。从 1998 年开始，经过 20 年的发展后，卷积神经网络依然遵循着这样的设计思想。其中，卷积发展出了很多的变种，池化则逐渐被带步长的卷积所替代，非线性激活函数更是演变出了很多的变种，这些在前面的章节中已经详细介绍过。
- 稀疏连接，也就是局部连接，这是以卷积神经网络为代表的技术能够发展至今的最大

前提。利用图像的局部相似性这个区别于传统全连接的方式，推动了整个神经网络技术的发展。

虽然 LeNet5 的错误率仍然停留在 0.7%，不如同时期最好的 SVM 方法，但是随着网络结构的发展，神经网络方法很快就超过了其他方法，错误率也降低到了 0.23%，甚至有的方法其错误率接近于 0。

由于 MNIST 是一个灰度图像数据集，而大部分现实的任务为彩色图像，所以 Alex Krizhevsky 等学者从 TinyImage 数据集中整理出了 CIFAR10 和 CIFAR100。在很长一段时间里，在 CIFAR10 这个数据集上没有像 LeNet5 这样的经典网络出现，直到 2012 年，Alex Krizhevsky 等人提出了 AlexNet，正式掀起了深度学习的热潮。

2. ImageNet 与 AlexNet

在 21 世纪早期，虽然神经网络开始有复苏的迹象，但是受限于数据集的规模和硬件的发展，神经网络的训练和优化仍然是非常困难的。MNIST 和 CIFAR 数据集都只有 6 万张图，这对于 10 个分类这样的简单任务来说或许足够，但是要想在工业界落地更加复杂的图像分类任务，仍然是远远不够的。

经过李飞飞等人的数年整理，2009 年他们发布了 ImageNet 数据集，从 2010 年开始，每年举办一次 ImageNet 大规模视觉识别挑战赛，即 ILSVRC。ImageNet 数据集总共有 1400 多万幅图片，涵盖 2 万多个类别，在论文方法的比较中常用的是 1000 类的基准。

ImageNet 发布早期，仍然是以 SVM 和 Boost 为代表的分类方法占据优势，直到 2012 年 AlexNet 模型 [5] 出现。

AlexNet 是第一个真正意义上的深度网络，与 LeNet5 的 5 层相比，它的层数增加了 3 层，网络的参数量也大大增加，输入图像的边长也从 28 变成了 224，深度学习模型的设计从此进入快速发展时代，原始的 AlexNet 结构如图 4.5 所示。

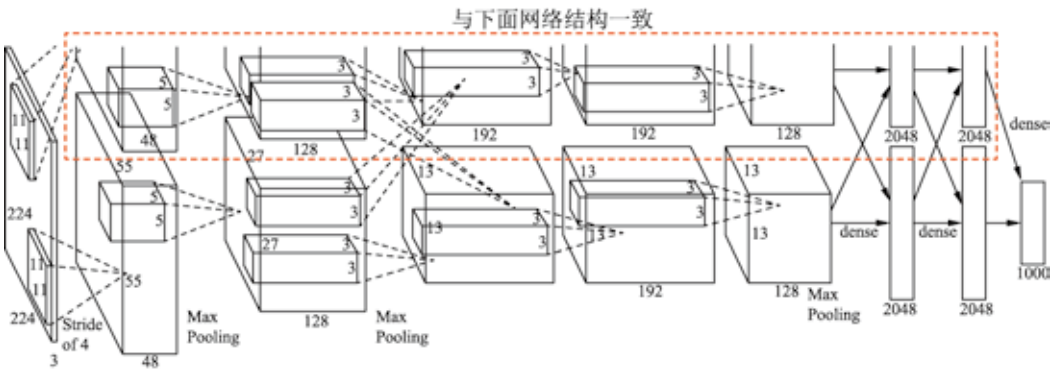


图 4.5　AlexNet 网络结构

AlexNet 有以下特点：

- 网络比 LeNet5 更深，包括 5 个卷积层和 3 个全连接层。模型的参数体积大概为 240MB，远大于 LeNet5。
- 使用 ReLU 激活函数，收敛很快，解决了 Sigmoid 在网络较深时出现的梯度弥散问题，目前，ReLU 激活函数及它的变种几乎是神经网络的“标配”。

- 加入了 Dropout 层，防止过拟合。虽然随着批次标准化等技术的出现，Dropout 没有那么流行了，但是它仍然是增强网络泛化能力很有效的技巧。
- 使用了 LRN 标准层，对局部神经元的活动创建竞争机制，抑制反馈较小的神经元，放大反应大的神经元，增强了模型的泛化能力。
- 使用裁剪、翻转等操作进行数据增强，增强了模型的泛化能力。预测时提取图片 4 个角加中间共 5 个位置进行左右翻转形成 10 幅图片来求取平均值，这也是比赛时常用的多模型预测结果融合技巧。
- 分块训练，当年的 GPU 计算能力不足，AlexNet 将数据分为两部分进行分组训练，最后在全连接层进行特征合并。

3. ILSVRC 分类任务的经典网络

2013 年 ILSVRC 分类任务的冠军网络是 Clarifai，不过更为熟知的是 ZFnet[6]。Hinton 的学生 Zeiler 和 Fergus 在研究中利用反卷积技术引入了神经网络的可视化，对网络的中间特征层进行了可视化，使研究人员检验不同特征激活及其与输入空间的关系成为可能。在这个指导下，他们对 AlexNet 网络进行了简单改进，包括使用更小的卷积核和步长，将 11×11 的卷积核变成 7×7 的卷积核，将 stride 从 4 变成了 2，性能超过了原始的 AlexNet 网络。

2014 年的 ILSVRC 分类任务的冠亚军网络分别是 VGGNet[7] 和 GoogLeNet[8]。其中，VGGNet 包括 16 层和 19 层两个版本，共包含参数约 550MB，图 4.6 展示了 VGG16 的配置。其全部使用 3×3 的卷积核和 2×2 的最大池化核，简化了卷积神经网络的结构。VGGNet 很好地展示了在先前的网络架构基础上通过增加网络层数和深度，就可以提高网络性能的过程，虽然简单，但是却异常有效。目前，VGGNet 仍然被很多的任务选为基准模型。

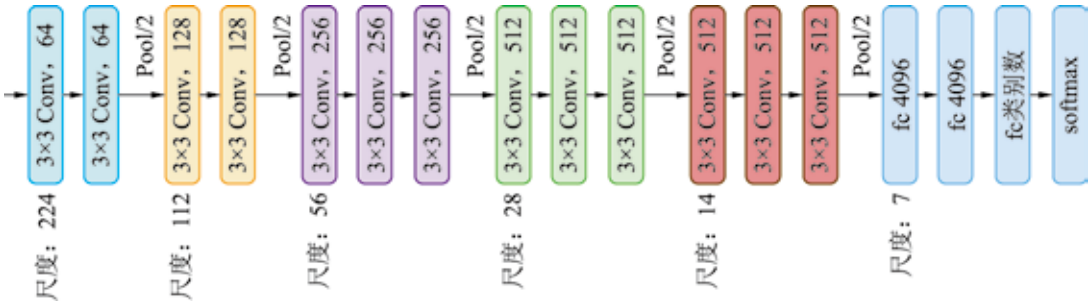


图 4.6　VGG16 网络结构

GoogLeNet 是来自 Google 的研究人员提出的 22 层的网络，其 top-5 分类错误率只有 6.7%。

GoogleNet 的核心是 Inception Module，图 4.7 展示了一个典型的结构。一个经典的 Inception 结构包括 4 个分支，即 1×1 卷积、3×3 卷积、5×5 卷积和 3×3 最大池化，最后对 4 个分支运算结果进行通道上的拼接。这就是 Inception Module 的核心思想。通过多个卷积核来提取图像不同尺度的信息然后进行融合，可以得到更好的图像表征。自此，深度学习模型的分类准确率在 ImageNet 上已经达到了人类的水平（5% ～ 10%）。

与 VGGNet 相比，GoogleNet 模型架构在精心设计的 Inception 结构下，模型更深又更小，计算效率更高。

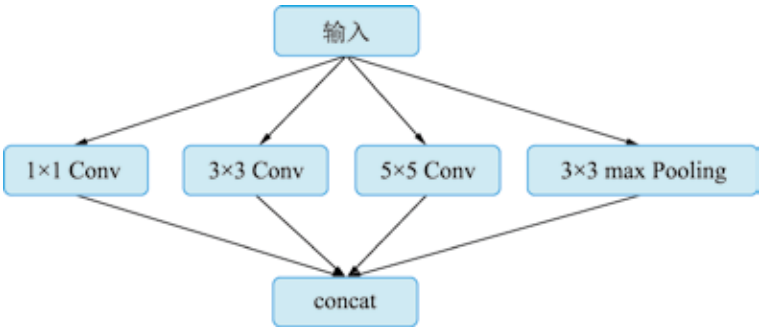


图 4.7　Inception Module 结构单元

2015 年，ResNet[9] 获得了分类任务冠军。它以 3.57% 的错误率表现超过了人类的识别水平，并以 152 层的网络架构创造了新的模型记录。ResNet 采用了跳层连接的方式，基本模块如图 4.8 所示。跳层连接成功地缓解了深层神经网络中的梯度消散问题，为上千层的网络训练提供了可能性。

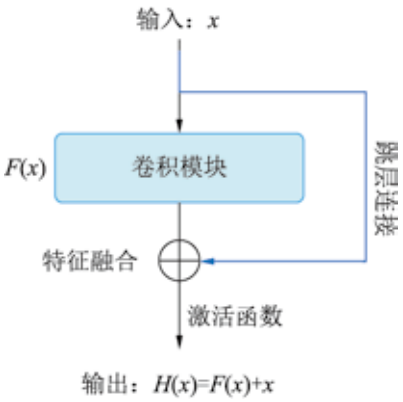


图 4.8　ResNet 跳层连接结构单元

2016 年依旧诞生了许多经典的模型，包括赢得分类比赛第二名的 ResNeXt[10]，101 层的 ResNeXt 可以达到 ResNet152 的精确度，在复杂度上只有 ResNet152 的一半，核心思想为分组卷积，即首先将输入通道进行分组，经过若干并行分支的非线性变换，最后合并。

在 ResNet 基础上，密集连接的 DenseNet[11] 的基本思想是在各个模块之间进行密集连接，如在前馈过程中将每一层与它前面的网络层都连接起来，如图 4.9 所示。对于每一层网络来说，前面所有网络层的特征图都作为输入，同时其特征图也都被后面的网络层作为输入所利用。

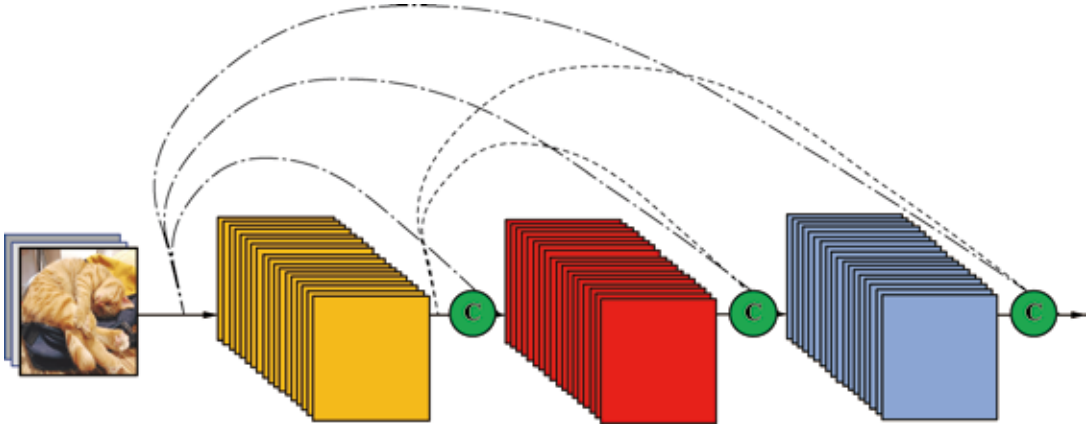


图 4.9　DenseNet 结构

DenseNet 的密集连接还可以缓解梯度消失问题，而且相比 ResNet，它更强化了特征传播和特征的复用并且减少了参数的数量。DenseNet 相较于 ResNet 所需的内存和计算资源更少，

并达到更好的性能。

2017 年是 ILSVRC 图像分类比赛的最后一年，SeNet[12] 获得了冠军，其结构如图 4.10 所示。它使用了特征通道加权的策略对特征进行处理，通过学习来获取每个特征通道的重要性，根据重要性的不同，降低或者提升相应的特征通道权重。

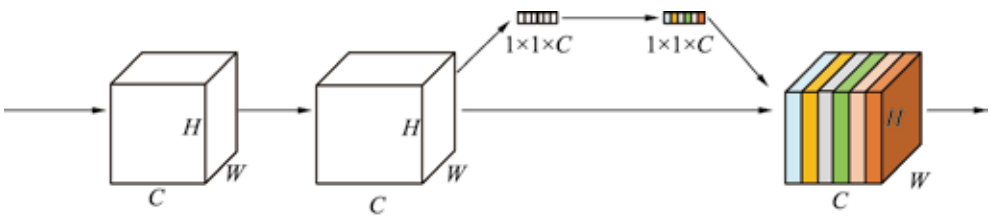


图 4.10　SeNet 结构单元

至此，图像分类的比赛基本落幕，也接近算法的极限。但是在实际应用中存在着比比赛中更加复杂的问题，需要大家不断积累经验。

4.1.3　优化目标

下面简单介绍多类别图像分类任务的优化目标，包括 0-1 损失，熵与交叉熵，Softmax 损失。

1. 0-1 损失

在机器学习中，损失函数（Loss Function）用来估量模型的预测值 $f(x)$ 与真实值 Y 的不一致程度。

对于图像分类任务，我们完全可以直接比较输出值与输入值是否相等，令 x 表示输入，y 表示标签，$f(x)$ 表示预测结果，L 表示 Loss，即损失，对于样本 i，它的 Loss 等于：

$$L(y_i, f(x_i)) = \begin{cases} 0 & \text{if } y_i = f(x_i) \\ 1 & \text{if } y_i \neq f(x_i) \end{cases} \tag{4.1}$$

当标签与预测类别相等时，损失为 0，否则为 1。但是 0-1 损失无法对 x 进行求导，这在依赖于反向传播的深度学习任务中无法被使用，0-1 损失更多的是启发新的损失函数的设计。

2. 熵与交叉熵

在物理学中有一个概念叫作熵，它表示一个热力学系统的无序程度。为了解决对信息的量化度量问题，香农在 1948 年提出了信息熵的概念，它使用对数函数表示对不确定性的测量。熵越高，则能传输的信息就越多；熵越少，则传输的信息就越少。我们可以直接将熵理解为信息量。

按照香农的理论，熵背后的原理是任何信息都存在冗余，并且冗余大小与信息中的每个符号（数字、字母或单词）出现的概率或者说不确定性有关。概率大，出现机会多，则不确定性就越小，因此在数学上就可以定义为离散随机事件的出现概率。

为什么选择对数函数而不是其他函数？首先，不确定性必须是概率 P 的单调递降函数，假设在一个系统中各个离散事件互不相关，要求其总的不确定性等于各自不确定性之和，对数函数是满足这个要求的。将不确定性 f 定义为 $\log(1/p)=-\log(p)$，其中 p 是概率。对于单个

的信息源，信源的平均不确定性就是单个符号不确定性 $-\log p_i$ 的统计平均值，信息熵的定义如下：

$$-\sum_{i=0}^{n} p_i \log p_i \tag{4.2}$$

假设有两个概率分布 $p(x)$ 和 $q(x)$，其中，p 是已知的分布，即标签，q 是预测的分布，即模型预测结果，则其交叉熵的定义是两个分布的互信息，用于反应两者的相关程度。

我们基于此就可以定义分类任务中最常用的损失，即对数损失，或者交叉熵损失（Cross Entropy Loss），我们统一称为交叉熵损失。它的定义形式如下：

$$-\sum_{i=0}^{n}\sum_{j=0}^{m} y_{ij} \log f(x_{ij}) \tag{4.3}$$

在式 4.3 中，n 对应样本数量，m 是类别数量，y_{ij} 表示第 i 个样本属于分类 j 的标签，它是 0 或者 1。对于普通的单标签多类别分类任务，只有一个分类的标签非 0。$f(x_{ij})$ 表示的是样本 i 预测为 j 分类的概率。损失的大小完全取决于分类为正确标签那一类的概率，当所有的样本分类都正确时，损失为 0，否则大于 0。随着真值标签概率的增加，损失值迅速下降，符合我们对分类任务损失特性的期望。

3. KL 散度

Kullback 和 Leibler 定义了 KL 散度用于估计两个分布 p 和 q 的相似性，定义如下：

$$D_{kl}(p \mid q) = \sum_i p_i \log\left(\frac{p_i}{q_i}\right) \tag{4.4}$$

D_{kl} 是非负的，只有当 p 与 q 处处相等时才会等于 0。式 4.4 也等价于式 4.5：

$$D_{kl}(p \mid q) = \sum_i (p_i \log p_i - p_i \log q_i) = -l(p,p) + l(p,q) \tag{4.5}$$

其中，$-l(p,p)$ 是分布 p 的熵，而 $l(p,q)$ 就是 p 和 q 的交叉熵。假如 p 是一个已知的分布，则 $-l(p,p)$ 是一个常数，此时 $D_{kl}(p|q)$ 与 $l(p,q)$，也就是交叉熵之间只有一个常数的差异，两者是等价的。

同时值得注意的是，KL 散度并不是一个对称的 Loss，即 $D_{kl}(p|q) \neq D_{kl}(q|p)$。

4. Softmax 损失

如果交叉熵损失的 $f(x_{ij})$ 的表现形式是 Softmax 函数，那么交叉熵损失就是我们熟知的 Softmax 交叉熵损失（Softmax With Cross Entropy Loss，简称 Softmax 损失），它是交叉熵函数（Cross-Entropy）的一个特例。

我们将式 4.3 重新描述如下：

令 z 是全连接层的输出，也是 Softmax 层的输入，$f(z)$ 是预测函数，C 是分类类别数，y_k 是标签。

交叉熵的计算如式 4.6 所示，$f(z_k)$ 是表示预测为第 k 类的概率，只有当样本属于第 k 类时，y_k 才为 1，否则为 0。

$$l(y,z) = -\sum_{k=0}^{C} y_k \log(f(z_k)) \tag{4.6}$$

假设 $f(z)$ 是 Softmax 函数，如式 4.7：

$$f(z_k)=\frac{e^{z_k}}{\sum_j e^{z_j}} \tag{4.7}$$

那么式 4.6 就是 Softmax 损失，但当 z_k 是非常大的正数或者非常小的负数时，式 4.6 的指数运算会非常不稳定，因此代码实现 Softmax 损失时会采取 log-sum-exp 策略，原理如式 4.8。

$$\log\sum\exp(z_1,\cdots,z_n)=\log(\sum_{i=1}^{n}e^{z_n})=\log(\sum_{i=1}^{n}e^{z_n-c}e^{c})=c+\log(\sum_{i=1}^{n}e^{z_n-c}) \tag{4.8}$$

只要保证 c 足够大，式 4.8 的指数运算就不会溢出，这样就可以稳定计算 Softmax 损失。

当我们把式 4.6 展开时，就可以得到：

$$l(y,z)=\log\sum_j e^{z_j}-\log e^{z_y} \tag{4.9}$$

$$\frac{\partial l(y,z)}{\partial z_k}=\begin{cases}f(z_y)-1, & 当y=k时\\ f(z_k), & \text{else}\end{cases} \tag{4.10}$$

从式 4.10 中可以看出 Softmax 的求导非常简单，在反向传播的数值计算中比较容易实现。Softmax 损失的特点是善于优化类间的距离，但是优化类内距离时比较弱。鉴于此，就有了很多对 Softmax 损失的改进，读者可以去参考一些相关资料。

4.1.4 评测指标

对于单标签分类问题，常用的评测指标有分类准确率和混淆矩阵等。在计算这些指标之前，我们先计算几个基本指标，这些指标是基于二分类的任务，也可以拓展到多分类。我们称标签为正样本，分类为正样本的数目为 True Positive，简称 TP；标签为正样本，分类为负样本的数目为 False Negative，简称 FN。标签为负样本，分类为正样本的数目为 False Positive，简称 FP；标签为负样本，分类为负样本的数目为 True Negative，简称 TN。

判别是否为正例只需要设一个概率阈值 T，预测概率大于阈值 T 的为正类，小于阈值 T 的为负类，默认就是 0.5。如果我们减小这个阀值 T，更多的样本会被识别为正类，这样可以提高正类的召回率，但同时也会让更多的负类被错分为正类。如果增加阈值 T，则正类的召回率降低了，精度增加了。

如果包含多类，如 ImageNet 分类比赛中的 1000 类，则预测类别就是预测概率最大的那一类。

1. 分类准确率

在单标签分类任务中，每一个样本只有一个确定的类别，如果预测到该类别，就是分类正确，没有预测到就是分类错误，因此最直观的指标就是 Accuracy，也就是准确率。

Accuracy=(TP+TN)/(TP+FP+TN+FN)，表示所有样本都正确分类的概率。

在 ImageNet 图像分类挑战赛中最常用的是 top-5 和 top-1 分类准确率。

所谓 top-1 分类准确率，即预测样本概率最大的类是否为真实标签的类别，如是则代表分类正确，反之则代表分类错误。top-5 分类准确率关注的是在预测的前 5 个类别中是否包含正确的类别，如果包含则代表分类正确。由此可知，top-5 的指标必定高于或等于 top-1。

2. 混淆矩阵

如果我们想知道类别之间相互误分的情况，查看是否有特定的类别相互混淆，则可以用

混淆矩阵画出分类的详细预测结果。对于包含多个类别的任务，混淆矩阵可以很清晰地反映各类别之间的错分概率，如图 4.11 所示。

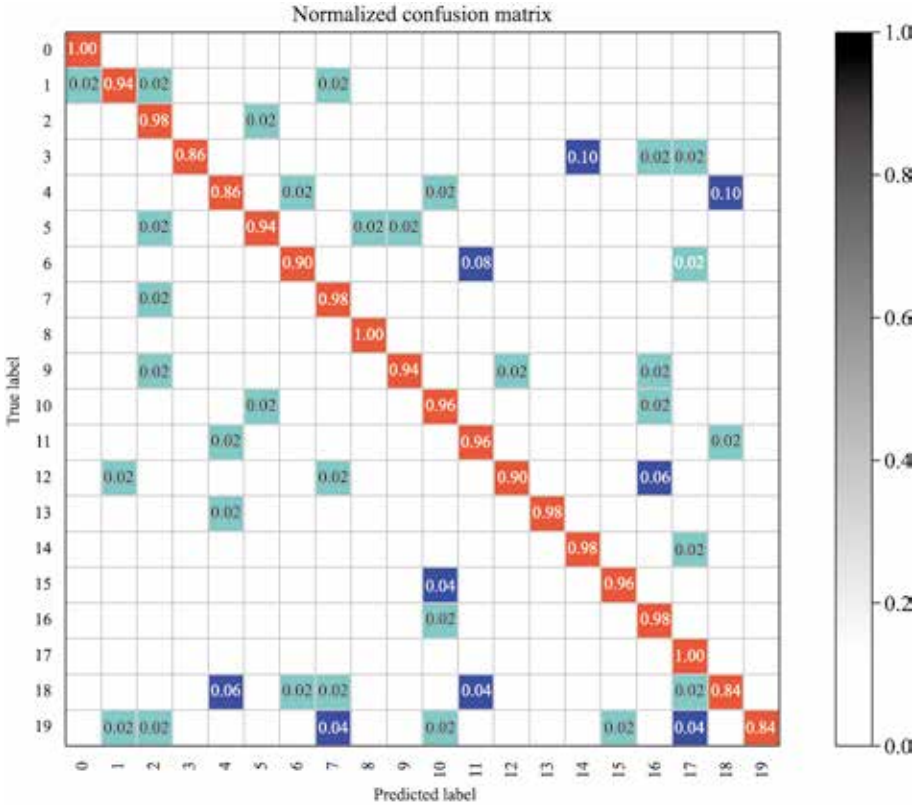


图 4.11　混淆矩阵

图 4.11 是一个包含 20 个类别的分类任务，混淆矩阵为 20 × 20 的矩阵，其中，第 i 行第 j 列表示第 i 类目标被分类为第 j 类的概率，可以看到，越好的分类器，其对角线上的值更大。

4.2　多标签图像分类

前面说的几个典型的分类问题都是单标签分类问题，即每一个图只对应一个类别，而很多的任务其实是多标签分类问题，一张图可以对应多个标签，本节我们详细介绍一下。

4.2.1　多标签图像分类问题

如图 4.12 展示了一个典型的多标签图像分类任务。

对于单标签分类问题，模型预测结果是一个整数，如 0 或 1，对于多标签分类问题，模型预测结果则是一个编码向量，如 [0 1]。

第 3 章介绍的 Pascal VOC 和 Microsoft COCO 都提供了多标签的标注，其中，Pascal

VOC2012 提供了 20 类，Microsoft COCO 则提供了 80 类。

标签1：柿子
标签2：雪
标签3：浅景深

图 4.12　多标签图像分类

4.2.2　多标签图像分类模型

多标签分类问题通常有两种解决方案，一种是将其转换为多个单标签的分类问题，对标签向量的每一维单独进行二分类预测，损失函数常使用交叉熵。这类方法的思路非常简单，但是忽略了各个类别属性的关系，而这却是非常重要的。

另一种解决方案是采用专门的多标签分类模型，这里我们介绍其中典型的 CNN-RNN 模型。为了建模各个标签之间的关系，CNN-RNN 使用 RNN 模型 [13] 对标签进行逐个预测，将多标签的预测问题变为序列预测问题，如图 4.13 所示。

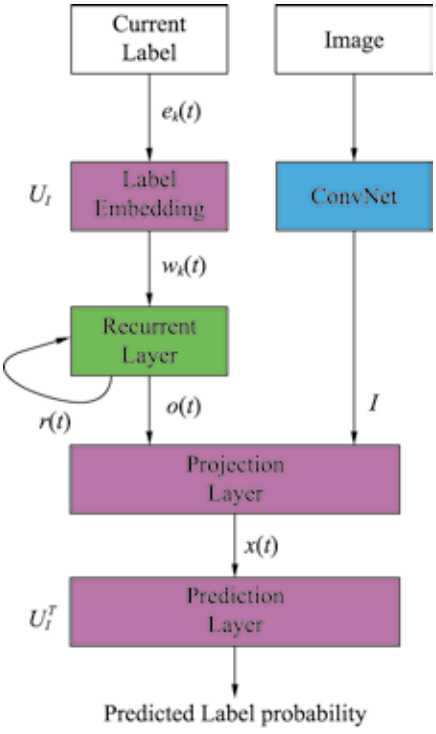


图 4.13　CNN-RNN 多标签图像分类框架

要想对 CNN-RNN 进行训练，首先需要对训练集中标签进行排序，将出现频率高的排在前面。接下来从前向后依次进行预测。

CNN-RNN 的训练流程如下：

（1）使用图像信息预测类别，得到标签向量 $e_k(t)$，与单标签图像分类一致。之后的每一次迭代，将预测的标记进行标签嵌入，得到词向量 $w_k(t)$，作为接下来 RNN 的输入。

（2）将词向量输入 RNN，更新输出层特征 $o(t)$ 和隐藏层的特征 $r(t)$。

$$o(t)=h_o(r(t-1)w_k(t)) \tag{4.11}$$

$$r(t)=h_r(r(t-1)w_k(t)) \tag{4.12}$$

$r(t-1)$ 表示上一次循环过程中的 RNN 的隐藏层特征，这里的迭代次数等于标签维度，每一次输出一个标签。

（3）将 RNN 输出特征与 CNN 特征映射到与词向量相同的特征空间，如式 4.13 所示，其中，I 表示图像对应的卷积特征，u_o^x 和 u_i^x 是两个映射矩阵。

$$x(t)=h(u_o^x o(t)+u_i^x I) \tag{4.13}$$

（4）将融合后的 $x(t)$ 与标签嵌入矩阵的转置相乘，再经过 Softmax 映射，输出类别分布概率。

$$s(t)=U_l^{\mathrm{T}}\bullet(x(t)) \tag{4.14}$$

按照以上流程进行迭代，完成模型的训练。在使用模型进行预测时，按照与训练标签中相同的顺序对各个维度的标签进行预测。

4.2.3　多标签图像分类评估指标

下面简单介绍一下多标签图像分类任务的优化目标与评测指标。

1. 优化目标

当我们将多标签图像分类任务看作多个二分类任务时，就可以完全采用上述单标签的优化目标和评测指标，即使用交叉熵函数作为损失函数，如式 4.15 所示。

$$l(y,x)=-\sum_{k=1}^{K}\sum_{c=0}^{1}y_c\log(f(x_c)) \tag{4.15}$$

其中，K 表示标签类别数，c=0,1 表示对每一个标签的维度进行二分类，y_c 表示对应类别 c 的真实标签，$f(x_c)$ 表示对应类别 c 的预测概率。

具体实现 $f(x_c)$ 的时候，我们通常使用 Sigmoid 函数而不是 Softmax 函数，因此这里的多标签分类损失为 Sigmoid 交叉熵损失，而不是 Softmax 交叉熵损失，两者在开源框架中的实现有差异。

2. 评测指标

最常用的评测指标是平均分类准确度。

令 $h(x_i)$ 是预测结果，y_i 是标签，平均分类准确度定义如式 4.16 所示。

$$\frac{1}{p}\sum_{1}^{p}h(x_i)==y_i \tag{4.16}$$

在式 4.16 中，只有当所有的类别属性都预测正确时才能等于 1，这是一个非常严格的要求。

除了上面常用的几个指标之外，还有其他的损失目标和评测指标，读者可以参考多标签图像分类研究 [14]。

4.3 细粒度图像分类

前面说过图像分类可以分为 3 个粒度，对于猫狗分类这样语义级别的问题，算法已经达到或超越人类专家水平，但是对于如何区分不同种类的猫这样的细粒度分类问题，算法只在某些数据集上勉强能突破 90%，还未超越人类专家。

细粒度分类的特点是类内方差大，类间距小，用于区分样本类别的有效信息只存在于很细小的局部区域，因此网络能否找到这些区域是成功的关键。很多方法是先找到前景对象，然后找到前景对象能辨识不同物种的局部区域。

本节我们介绍细粒度分类的经典模型，根据其是否需要对语义子区域进行定位，可以分为基于语义子区域的模型和基于高维特征的模型。

4.3.1 基于语义子区域的模型

这里所谓的语义子区域，指的是动物的头部、尾巴，植物的根部、茎叶等生物学上具有定义的部件。基于语义子区域的模型的主要思路是通过网络对前景对象的语义子区域进行定位，因为我们通常是通过一些子区域的差异来实现细粒度分类的，如动物的头部纹理和尾部颜色等。

在这类方法中，Part RCNN[15] 方法是一个典型的代表，其基本原理如图 4.14 所示。以动物分类为例，它的基本思路是首先检测出动物的头部和躯干等“部件”，对它们分别进行特征提取，从而实现姿态对齐，而且提取的都是前景区域，因此不会受到背景的严重干扰。

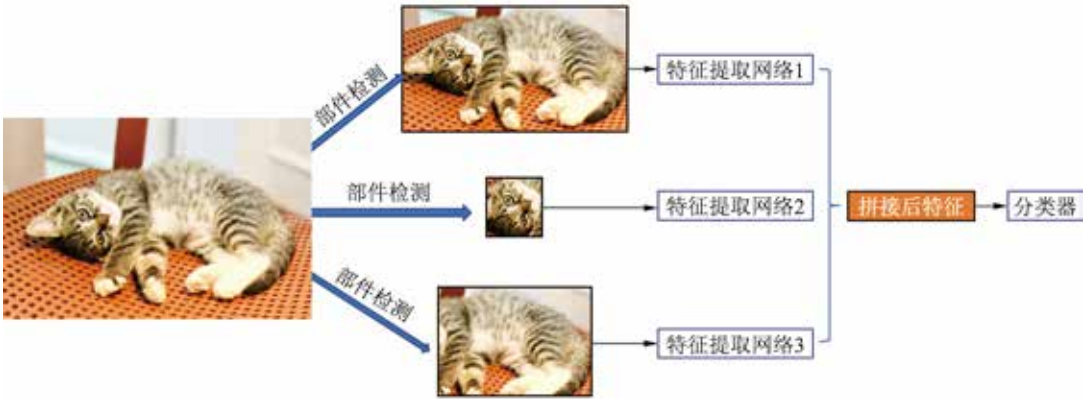


图 4.14　Part RCNN 方法的典型流程

在具体实现时，Part RCNN 方法利用经典的检测算法 RCNN 与空间约束关系进行各个部件的检测，再分别对各个部件提取卷积特征，将不同区域的特征相互串联用来训练 SVM 分类模型。

在训练的时候，数据集会提供标注框和局部信息，测试时在不提供任何信息的情况下，在 CUB-200-2011 鸟类分类数据集上能达到 73% 以上的精度。如果附加姿态对齐等操作，则有望进一步提升精度。

Part RCNN 代表了典型的基于语义子区域的模型，它有两个局限性：

- 标注代价较高，计算量较大。Part RCNN 需要训练目标检测模型和多个特征提取模型，这无疑增加了计算量，对数据集的标注也提出了额外的要求。
- 算法不够灵活，不具备通用性。对于前述鸟类分类，可以分为头部和躯干等部件，但是对于其他类别就未必合适。例如植物，就很难这样划分，这限制了算法的通用性。

针对 Part RCNN 的主要缺陷，MA-CNN 方法 [16] 进行了改进，它的主要思路是自适应地进行部件划分，通过对通道进行聚类来实现，如图 4.15 展示了 MA-CNN 方法的典型流程。

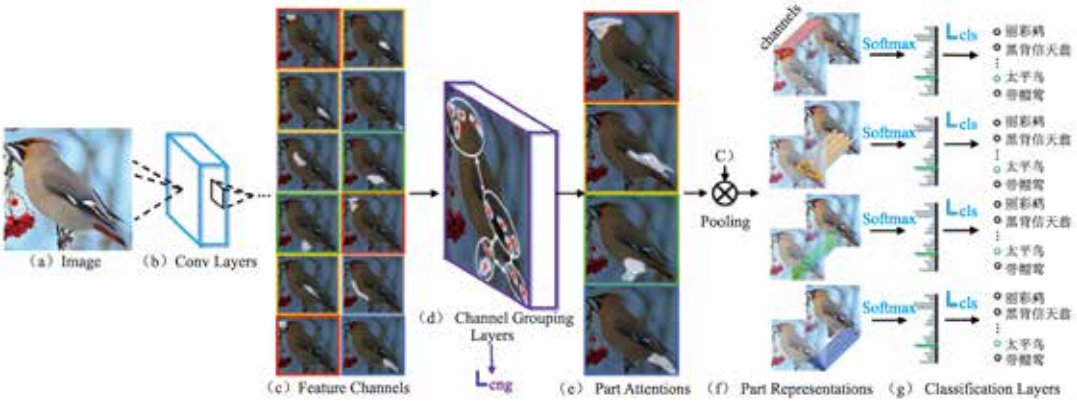


图 4.15　MA-CNN 方法的典型流程

MA-CNN 方法执行流程可以分为以下几步：

（1）通过卷积网络对图片提取特征，得到图 4.15 中的（c）图。

（2）对各个特征通道进行聚类，因为不同的特征通道会提取不同的语义特征，当模型训练得比较好之后，就会对应图像中的语义区域，有的对应头部特征，有的对应躯干特征。对应同一类部件的特征通道会具有更大的相似性，如空间激活位置。这一步使用 K-means 聚类算法来实现，具体计算每个通道对应的特征向量 $t_x^1, t_y^1 \cdots t_x^n, t_y^n$，其中，$t_x^1, t_y^1$ 表示在当前通道中第 t_x^1 行 t_y^1 列激活值最大，n 表示图像个数。假设在当前数据集中图像可以分为 4 个部件，则可以基于特征向量聚成 4 类，图 4.15 中的第 1、8、11 个特征图表示为同一类，接下来就可以将同一类的特征激活值叠加，将其作为一个部件的注意力图，从而实现该部件的定位，如图 4.15 所示。

（3）对第（2）步进行迭代，即对通道所属的部件进行学习，具体实现就是对每个部件添加全连接层进行预测，输入是第（1）步提取的特征，输出就是 d 维向量，表示每个通道有多大的概率属于当前部件，该全连接层使用第（2）步生成的聚类标签来进行预训练。

（4）使用第（3）步得到的 4 个 d 维向量对特征图的通道进行加权求和，得到 4 个部件对应的注意力图，然后与特征图各自相乘得到部件特征，池化后得到最终的特征向量，经过 Softmax 映射后对每个部件进行预测得到分数，最后把 4 个部件的分数累加起来得到最终的预测结果。

4.3.2 基于高维特征的模型

基于语义子区域的模型需要显式或者隐式的注意力机制来实现定位，总体来说流程比较复杂。

一般来说，网络最后特征层的输出是通道的一阶统计量，有的研究认为通过引入高阶统计量有助于提升特征的表达，最常见的方法就是引入不同特征通道之间的协方差矩阵，获得二阶统计量。

双线性网络[17]基于此思想，提出了一种用于解决细粒度图像分类的方法，它通过两个子网络同时进行学习，两个子网络共同配合完成任务。一个典型的双线性网络结构如图 4.16 所示，它包括两个特征提取网络。一张图像通过这两个特征提取网络分别提取特征，再进行双线性积分操作得到特征。

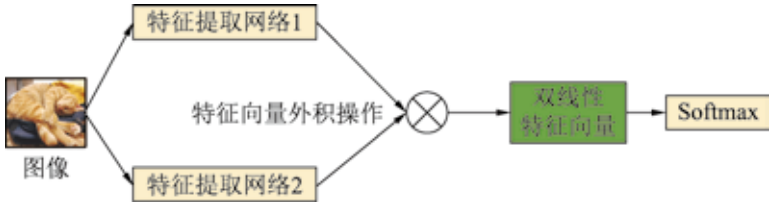


图 4.16　双线性网络

假设特征提取网络 1 和特征提取网络 2 输出的特征维度分别是 ($K1$, M, N) 和 ($K2$, M, N)，其中，$K1$、$K2$ 为通道数，M、N 分别为每一个特征图的长和宽尺寸。

双线性积分操作就是将两个网络的输出特征进行外积，即将 ($K1$, M, N) 和 ($K2$, M, N) 两个维度的特征向量按照逐个通道两两进行外积，一个尺度为 (M, N) 和另一个尺度为 (M, N) 才能进行逐个像素的乘法操作，因此二者的特征图的长、宽尺度必须相等。

完成双线性积分操作后，维度变为 ($K1 \times K2$, M, N)，最后用求和池化函数（Sum Pooling）来综合不同位置的特征。所谓求和池化函数，就是将一个特征通道所有像素值相加，即将 (M, N) 大小的特征图池化为 (1, 1) 大小，作为最后的全连接层分类器的输入。一般在输入分类器之前还会经过符号平方根变换，并增加 L2 标准化（Elementwise Normalization Layer）。由于经过了全局的池化操作，双线性积分操作计算得到的特征有位移不变性。

在计算框架中，双线性积分操作会采用矩阵相乘的形式来实现，即首先将 ($K1$, M, N) 和 ($K2$, M, N) 的特征向量各自变成 ($K1$, MN) 和 (MN, $K2$) 大小的矩阵，然后将两者相乘得到 ($K1$, $K2$) 的矩阵，再展开成长度为 ($K1 \times K2$) 的特征向量。

这里的两个特征网络可以共享部分参数，除了前面完全采用两个独立网络的方式，还有另外两种方式，分别是共享部分权重与全部权重，如图 4.17 和图 4.18 所示。

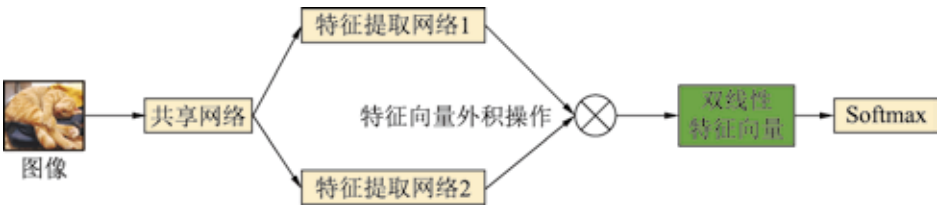


图 4.17　共享部分权重的双线性网络

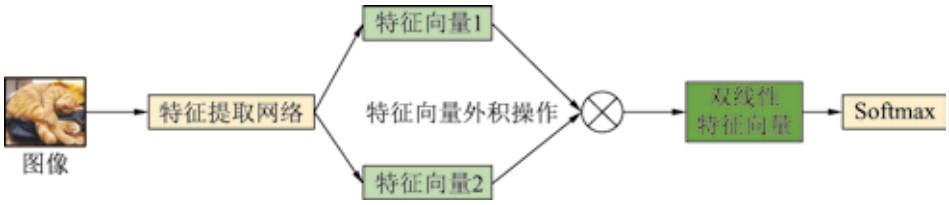


图 4.18　共享全部权重的双线性网络

研究者发现图 4.18 中的实现相比图 4.16 中的实现并没有降低性能，还可以降低参数量，因此大部分双线性网络都采用图 4.18 的形式。

原始的双线性积分操作后的输出特征通道数非常大，如果两个输入都是 512，则输出达到了 250 000 的通道数量，这个计算量超过了大部分显卡的容量，因此研究者们提出了紧凑的双线性积分方法 [18]。它利用任何多项式可以进行低维度近似的原理寻找到了一个映射，使得输出维度远小于原始的双线性积分方法，但可以取得近似的性能。

总之，双线性模型的结构比较简单，不需要特殊设计检测模块或者注意力机制模块，不过直接学习高维特征之间的相关性有较高的训练难度。

4.4　半监督与无监督图像分类

虽然图像分类任务的标签是类别，进行标注时成本较低，但是当要标注的数据量特别大时，也具有不可忽略的成本，著名的 ImageNet 数据集的标注就花费了数年时间，而且随着目标数量、可辨识难度增加，其标注成本会更大。

另一方面，在实际生产环境中，很多新的数据在源源不断地产生，如果我们想要快速利用新的数据来改进模型，则需要全自动或者半自动的数据标注方案，而这就是本节给大家介绍的半监督与无监督图像分类问题。

4.4.1　基本问题与解决思路

首先我们来看一个示意图，如图 4.19 所示。

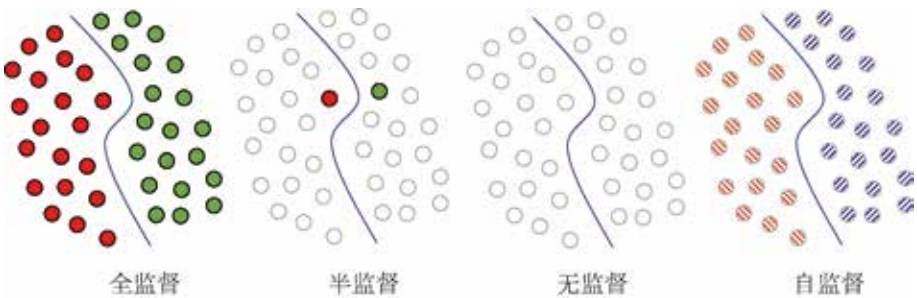


图 4.19　不同监督信息的模型分类

图 4.19 展示了 4 种不同程度监督信息的模型分类，包括全监督、半监督、无监督和自监督。全监督指所有的样本都有标注好的标签，也就是我们大家最熟悉的训练模型的方式。半监督指只有部分样本有标注的标签，因此需要同时使用有标签和无标签的数据进行训练。无监督指所有样本都没有标签，完全需要模型自行学习。自监督与无监督类似，主要区别在于，在自监督学习中，数据可以发掘出自身的统计特性，而且形成自我监督学习机制。

首先我们来介绍解决半监督分类与无监督分类问题的通用思想，这里包括 3 个概念。

1. 一致性正则化

图 4.20　数据增强样本

所谓的一致性正则化（Consistency Regularization），即对数据进行数据增强，产生的新数据输入相同的分类器，预测结果应保持自洽。也就是说，对样本本身进行增强，不应改变其标签。

如图 4.20 展示的为同一张样本生成的 64 张图片，每一张图片都应该有相同的预测结果。

基于该思想，我们可以将没有标签的数据进行随机数据增强，约束模型的预测结果一致性，这样虽然不能让模型学习能够预测出数据的标签，但是可以约束模型学习到有效的特征表达，这在具有非常丰富的无标签数据的情况下，可以充分锻炼模型的特征表达能力。

2. 最小化熵原理

所谓的最小化熵 [19]（Entropy Minimization），即分类器的分类边界不应该穿过边际分布的高密度区域，应该强迫分类器对未标记数据作出低熵预测，如图 4.21 所示。

图 4.21　最小化熵原理

对于图 4.21 所示的图像，假如预测类别是 cat 和 dog 两类，图 4.21 展示了两种预测结果。

第 1 种：熵的计算见式 4.17。

$$-0.1\times\log(0.1)-0.9\times\log(0.9)=0.141 \tag{4.17}$$

第 2 种，熵的计算见式 4.18。

$$-0.5\times\log(0.5)-0.5\times\log(0.5)=0.301 \tag{4.18}$$

我们希望模型能对两个类别有明显的概率预测差异，这说明模型可以区分这两个类，其差异越大越好，所以式 4.17 的值比式 4.18 更低，是更偏爱的结果。即使 cat 的预测概率为 0.1，dog 的预测概率为 0.9，将目标的类别完全预测错误，也比各自给出 0.5 的概率完全不能区分要好。

3. 伪标签

所谓伪标签，即通过算法生成标签，将无监督转换为监督学习问题，直到迭代至稳定，如图 4.22 所示。

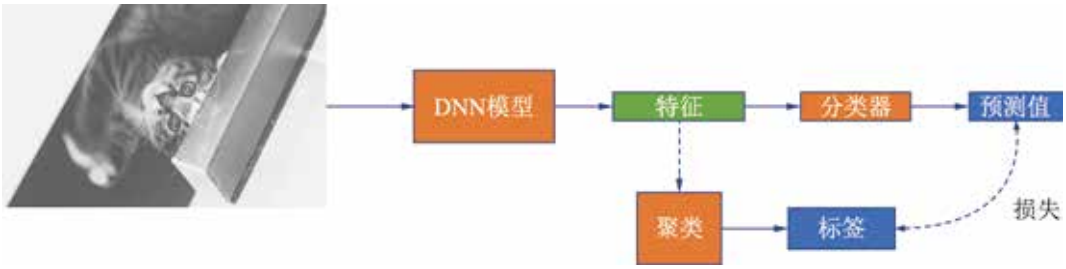


图 4.22　基于伪标签的训练原理

在图 4.22 中，数据首先通过模型提取到特征，然后使用无监督的聚类方法如 K-means 进行聚类并赋值类别标签，另一方面，特征经过分类器进行预测，通过聚类算法产生的伪标签与预测值来计算损失。

4.4.2　半监督分类模型

接下来我们介绍典型的半监督图像分类模型，基于上述思想，Google 提出了半监督分类模型 MixMatch[20]，其融入了一致性正则、伪标签思想、熵正则化及 Mixup 技术，其原理如图 4.23 所示。

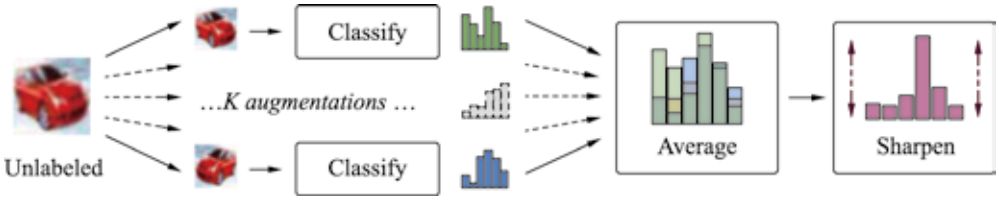


图 4.23　MixMatch 框架原理

MixMatch 的执行流程可以分为以下几步：

（1）一般数据增强：具体就是对一个批次的有标签数据和一个批次的无标签数据进行数据增广，分别得到一个批次的有标签增广数据和 K 个批次的无标签增广数据。当 K=2 时，使用的增强操作为对图像随机左右翻转和剪切。

（2）预测伪标签：将 K 个增广后的数据输入分类器，计算平均分类概率，应用温度

Sharpen 算法使伪标签熵最小化，即约束预测的各个类别的概率不能过于平均。假设 p_i 是 K 个模型的第 i 类平均预测结果，则温度 Sharpen 算法如下：

$$\text{Sharpen}(p,T)_i = p_i^{\frac{1}{T}} / \sum_{j=1}^{L} p_j^{\frac{1}{T}} \tag{4.19}$$

其中，T>0，当 T 越小，则 Sharpen 算法操作后的概率差异越大，T 接近 0 时，Sharpen 算法操作后的概率分布接近独热编码标签，MixMatch 的作者建议 T 取值为 0.5。

（3）将一个批次的有标签增广数据和 K 个批次的无标签增广数据混合，随机重排得到 W 数据集。将一个批次的有标签增广数据和 W 的前一个批次利用 Mixup 混合构成新的有标签增广数据；再将 K 个批次的无标签增广数据和 W 剩下的数据利用 Mixup 混合构成新的无标签增广数据。Mixup 数据增强算法在第 3 章中已经详细介绍过，此处不再解释。

（4）对增广后的有标签的数据，根据标签计算交叉熵损失，对增广后的无标签的数据，计算 L2 损失。假设 x 是增广的无标签数据输入，q 是猜测的标签，y 是真实的标签，无标签数据的损失形式如式 4.20 所示。

$$L_{x,q} = \| q - p_{\text{model}}(y \mid x) \|_2 \tag{4.20}$$

这里对于无标签数据的损失计算没有像有标签数据一样计算常用的交叉熵损失，是因为 L2 损失相对交叉熵对类别敏感度低，这样更能容忍猜测的标签 q 的错误，在不影响模型学习的情况下，可以提高训练的稳定性。

4.4.3 无监督分类模型

相比于半监督分类问题，无监督分类问题更进一步——彻底没有了监督信息，想要进行模型训练，不可缺少的一步就是产生伪标签。

这里我们以 Deep Clustering[21] 为例进行介绍，它通过 CNN 提取图像的特征信息，然后通过 K-means 算法对特征进行聚类，并赋予相应的伪标签，再用伪标签进行相应的分类训练，如图 4.24 所示。

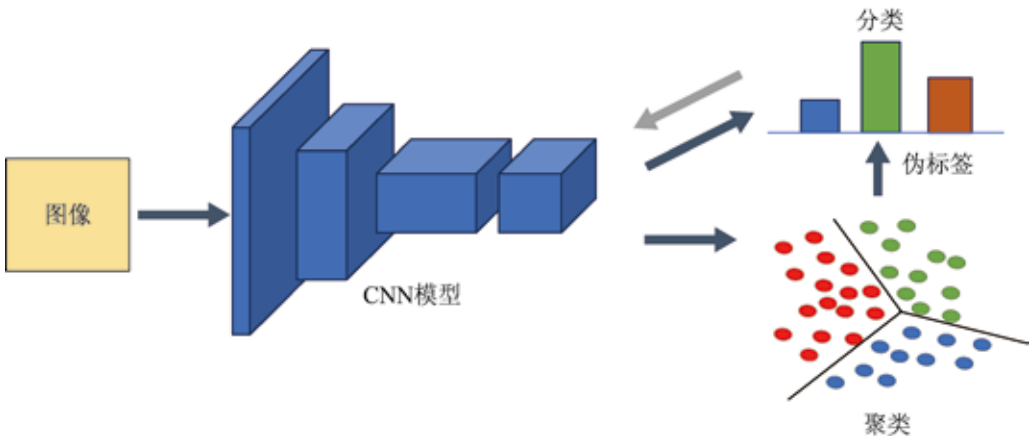


图 4.24　DeepCluster 框架

在图 4.24 中其实包含两个模型，即分类模型与聚类模型，因此优化目标也包含两部分。分类模型的优化目标就是交叉熵，而聚类模型的优化目标如下：

$$\min_{C\in\mathbb{R}^{d\times k}} \frac{1}{N}\sum_{n=1}^{N} \min_{y_n\in\{0,1\}^k} \left\| f_\theta(x_n) - Cy_n \right\|_2^2 \tag{4.21}$$

其中，x_n 表示样本，$f_\theta(x_n)$ 表示模型提取的特征，y_n 表示聚类生成的伪标签，它是一个 k 维的独热编码向量，C 表示 $d\times k$ 维的聚类中心矩阵。

DeepCluster 框架原理非常直观，但是我们需要避免网络产生的特征经过聚类都位于某个簇心周围，使得其他簇心没有样本，从而对于任意的输入都产生相同的输出，获得没有意义但正确的解。

作者们采取了两个策略：

（1）当某个簇心为空时，随机选择一个非空的簇心，在其上加一些小的扰动作为新的簇心，同时让属于非空簇心的样本也属于新的簇心。

（2）根据类别（或伪标签）对样本进行均匀采样，即根据类别数量进行加权，使得聚类后的类别分布均匀。

4.4.4　自监督分类模型

自监督与无监督分类类似，不同之处在于，自监督没有显式的聚类过程，以 Invariant Information Clustering CNN[22] 为例，其框架如图 4.25 所示。

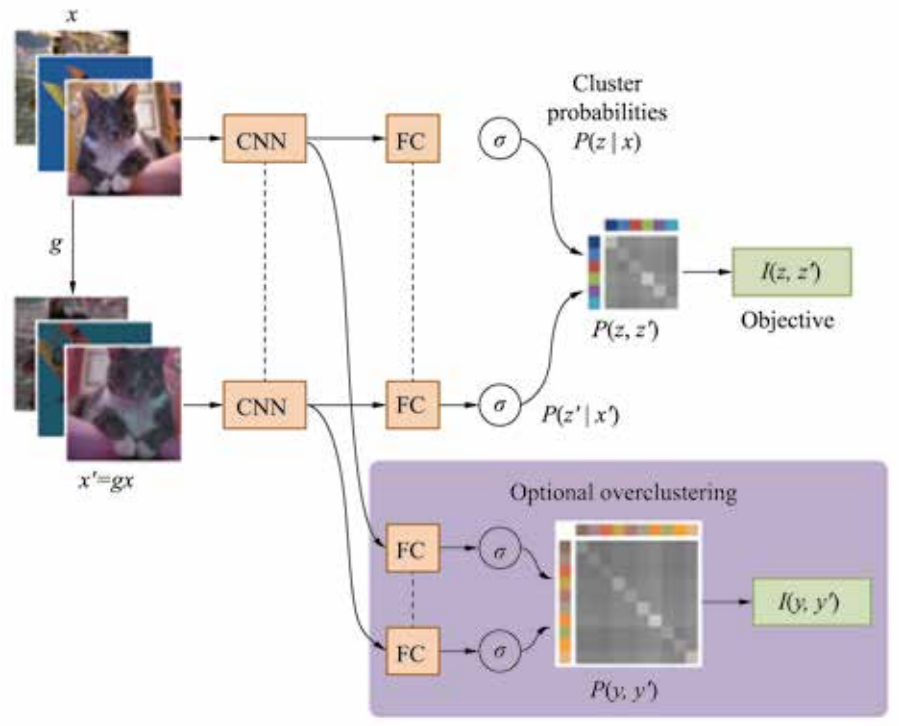


图 4.25　Invariant Information Clustering 框架

在图 4.25 中，基本思想是将原有的图像进行平移、旋转等数据增强后得到新的图像，然后将原有图像和新图像输入同一个 CNN 中，将最大化两者的互信息作为优化目标，因为互信

息表示一个随机变量包含的关于另一个随机变量的信息量，可以衡量两个随机变量之间的相互依赖程度。对于相同的样本，其互信息应该更大，不同的样本则互信息应该更小。

与无监督框架相比，虽然在自监督框架中聚类模块并不是必要的，但是它们的核心思想都是利用聚类等方法来学习无标注数据的统计规律，约束相似的样本产生相似的特征。

关于更多半监督与无监督图像分类模型，大家可以参考文献 [23]。

4.5 其他图像分类问题的典型难题

虽然图像分类是一个基本问题，但是现实中的图像分类任务有一些典型的挑战，除去我们上面介绍的细粒度图像分类，半监督与无监督图像分类，常见的还有类别不均衡问题，样本量太少问题和零样本分类问题。

4.5.1 类别不均衡问题

有时我们需要创建一个分类器，用来帮助我们识别几类图片，但是输入却可以是任意的图像，这些都是未知类别的负样本，它们的数量远远大于正样本。要保证这些负样本不会被分为正样本，就需要补充大量的负样本进行训练，而这个量级的把握往往是一个比较复杂的工程问题，因为它会造成正负样本的类别不均衡，一方面负样本相对于正样本是无穷的，另一方面某一类的负样本相对于正样本又是极少的。

另外，有很多任务其类别存在极大的不均衡问题，如边缘检测任务。图像中的边缘像素与非边缘像素通常有 3 个数量级以上的差距，如图 4.26 所示。

图 4.26 原图（左）和 Laplacian of Gaussian 边缘检测结果（右）

对于类别非常不均衡的图像分类任务，常见的解决办法包括：对少样本进行过采样，对少样本进行数据增强，或者对不同样本的损失贡献进行加权。

这里我们以加权的 Softmax 损失为例，形式如下：

$$l(y,z)=-\sum_{k=0}^{C} w_k y_c \log(f(z_k)) \tag{4.22}$$

w_k 就是权重，例如在边缘检测的分类问题中，令 k=0 代表边缘像素，k=1 代表非边缘像素，可以选值为 w_0=1，w_1=0.001，这就是通过加大边缘像素的权重来缓解类别极其不均匀的情况。当一个图像中的边缘像素数量远远少于非边缘像素时，通过加大边缘像素对损失的贡献，可以让模型更多地关注边缘像素的召回。

当然，这个权重还可以动态地计算让其自适应。例如在每一张图中，按照像素的比例进行加权。

另外，Focal Loss 也是加权对数损失的一个变种，它的定义如下：

$$L=\begin{cases}-(1-f(x))^{\gamma}\log(f(x)), & y=1\\ -(f(x))^{\gamma}\log(1-f(x)), & y=0\end{cases} \tag{4.23}$$

式 4.23 对正负样本进行了分开表达，$f(x)$ 表示属于标签为 1 的类别概率，即正样本，我们首先假设正样本是相对于负样本更容易的样本，反之亦然。

对于类别 1 的样本，当 $f(x)$ 越大时，调制项 $-(1-f(x))^{\gamma}$ 就越小，因为此时基于该样本是一个容易样本的假设，所以给予其更小的损失贡献权重。通过 γ 的设置，可以自适应地对难易样本进行学习，即给难样本更大的权重，容易样本更小的权重。

4.5.2　样本过少问题

在很多的应用场景中，想要获取足够多的样本是较难的，如某些工业产品、医疗影像和金融欺诈正样本等。深度学习模型由于有较大的容量，常需要有较大的数据集才能进行学习，因此我们需要从模型训练和数据两个方面来考虑如何解决这个问题。

从模型的训练角度来看，如果模型的训练数据足够大且与任务相匹配，那么我们可以采用预训练模型 + 微调的方式，所学到的特征具备一定的通用性。

从数据角度来看，我们可以通过第 3 章介绍的各种数据增强方法来产生更多的数据，从而降低模型的过拟合风险。

4.5.3　零样本识别问题

深度学习模型需要预先训练然后才能使用，而训练后模型能识别的类别是固定的，例如我们训练出猫狗分类模型，当给模型输入狮子的图片时，模型只能预测出该图片中的动物为狗和猫的概率有多大，而无法给出正确的类别，如图 4.27 所示，这是典型的零样本识别问题。

图 4.27　零样本图像识别问题

零样本图像识别是一个非常重要的问题，它需要模型具有“触类旁通”的能力。例如，小孩子在认识斑马这一类动物时，只需要给出它像马，并且全身是黑白条纹的认知，即使小孩子没有见过斑马的图片，在第一次见到斑马时也可以认出它，而我们前面所介绍的模型显然并没有这个能力。

当模型在线上使用时，经常会遇到新增类别识别的需求，以商品识别为典型代表。如果每一次新增类别都重新训练分类模型，则计算成本太大，无法满足落地场景，此时采用分类 + 检索的策略才是实际可行的技术方案。先训练一个类别数足够大的分类模型，将其作为通用的特征提取器，当新的图片需要预测时，将其经过特征提取后与数据集中存储的特征进行相似度匹配，使用 KNN 最近邻等算法获得最近邻样本的类别作为预测标签。这样一来，数据集中的标签数量可以大于预训练好的模型的全连接层类别数，随着商品数量越来越多，可以定期对预训练分类模型进行更新。

目前零样本图像识别的主流学术研究方法有基于直接语义预测的方法，基于嵌入模型的方法和基于生成模型的方法，读者可以参考文献 [24] 了解更多。

4.6 简单表情图像分类实战

本节将带领读者完成一个表情识别的图像分类任务，我们分别使用 PyTorch 和 Caffe 两个框架来实现。这个任务在图像分类中属于比较简单的任务，我们将从数据集的建立到模型的训练与测试详细介绍任务的整个执行过程。

4.6.1 项目背景

人脸表情识别（Facial Expression Recognition，FER）作为人脸识别技术的一个重要组成部分，近年来在人机交互、安全、机器人制造、自动化、医疗、通信和驾驶领域得到了广泛的关注，成为学术界和工业界的研究热点。

表情是我们在日常生活中经常提到的一个词语，在人际沟通中，人们通过控制自己的面部表情，可以加强沟通效果。人脸表情是传播人类情感信息与协调人际关系的重要方式，根据心理学家 A.Mehrabia 的研究表明，在人类的日常交流中，通过语言传递的信息仅占信息总量的 7%，而通过人脸表情传递的信息却达到信息总量的 55%。可以这么说，我们每天都在对外展示自己的表情也在接收别人的表情，那么表情到底是什么呢？

1. 什么是表情

所谓面部表情，是面部肌肉（也被称为 action units）的一个或多个动作或状态的结果，这些运动表达了个体对观察者的情绪状态，面部表情是非语言交际的一种形式。

人类的面部表情至少有 21 种，除了常见的高兴、吃惊、悲伤、愤怒、厌恶和恐惧 6 种表情之外，还有惊喜（高兴＋吃惊）和悲愤（悲伤＋愤怒）等 15 种可被区分的复合表情。

面部表情的研究始于 19 世纪，1872 年，达尔文在他著名的论著中就阐述了人的面部表情和动物的面部表情之间的联系和区别。1971 年，Ekman 和 Friesen 对现代人脸表情识别做了开创性的工作，他们研究了人类的 6 种基本表情（即高兴、悲伤、惊讶、恐惧、愤怒和厌恶），

确定识别对象的类别，并系统地建立了有上千幅不同表情的人脸表情图像数据库，细致地描述了每一种表情所对应的面部变化，包括眉毛、眼睛、眼睑和嘴唇等。

到了 20 世纪 90 年代，K.Mase 和 A.Pentland 使用光流来判断肌肉运动的主要方向，自此面部表情自动识别进入了新的时期，国内的研究也开始有所进展。

2. 微表情

随着对表情研究的深入，学者们将目光聚焦到了一种更加细微表情的研究，即微表情的研究，什么是微表情呢？

微表情是心理学中的名词，它是一种人类在试图隐藏某种情感时无意识做出的短暂的面部表情。它们对应 7 种世界通用的情感：厌恶、愤怒、恐惧、悲伤、快乐、惊讶和轻蔑。微表情的持续时间仅为 1/25s 至 1/5s，表达的是一个人试图压抑与隐藏的真正情感。虽然一个下意识的表情可能只持续一瞬间，但是表达的可能是相反的情绪。

微表情的研究具有巨大的商业价值和社会意义。在美国，针对微表情的研究已经应用到国家安全、司法系统、医学临床和政治选举等领域。例如，在国家安全领域，一些训练有素的恐怖分子可能轻易就可以通过测谎仪的检测，但是通过微表情，通常可以发现他们虚假表面下的真实表情。因为微表情的这种特点，它在司法系统和医学临床上也有较好的应用。电影制片人、导演或者广告制作人等，也可以通过人群抽样采集的方法对他们观看宣传片或者广告时的微表情来预测宣传片或者广告的效果。

总之，随着科技的进步和心理学的不断发展，对面部表情的研究将会越来越深入，内容也会越来越丰富，应用也将越来越广泛。

3. 项目分析

在这样的背景下，我们选择了表情分类任务来作为实践项目。

传统的对表情进行分类和识别的方法，通常采用 ASM 和 AAM 等进行面部建模，基于深度学习的方法则利用卷积神经网络等深度学习模型自身的强大建模能力，从大量的数据中学习全局和局部的表情特征，这也是现在研究的主流方向。

最简单的表情识别模型就是将完整的人脸图经过 CNN 模型提取特征，再输入分类器进行预测，只要有足够丰富的数据，就可以学习出不错的模型。

这里我们将该问题进行简化，因为表情与人脸面部的许多子区域有关系，包括眉毛、鼻子和嘴巴等，尤其是嘴巴区域。图 4.28 展示了无表情和有微笑表情的图片，它们的差异就在于嘴唇区域的不同。

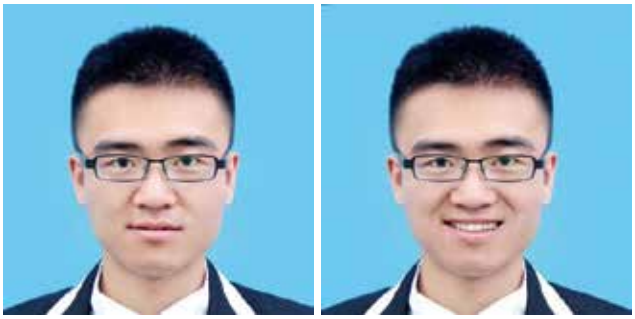


图 4.28　无表情与微笑表情的图片

这种基于人脸区域的表情识别模型不仅有更高的精度，而且可以让初学者加深对数据预处理的理解，因此我们来实现基于嘴部区域的人脸表情识别模型，识别类别分为无表情、微笑、嘟嘴和开口大笑 4 种。

4.6.2 数据预处理

表情识别的研究近几年非常广泛，因此相关的开源数据集很多，第 3 章我们已经对目前的表情数据集进行了详细介绍，这里我们选择 CelebA 数据集来获取微笑和无表情样本。

CelebA 数据集 [25] 是香港中文大学一个用于研究人脸属性的数据集，总共包含超过 200 000 张名人图像。CelebA 的图像涵盖大型姿态变化和复杂背景，多样性非常好，有约 10 万张带微笑属性的数据，我们从中随机选择了 5000 张微笑和 5000 张无表情的人脸数据。但是剩下的嘟嘴和大笑的图像在开源数据集中却很少。为此我们使用了嘟嘴、大笑等关键词，使用第 3 章介绍的可以爬取搜索引擎结果的爬虫工具各自爬取了几千张图，然后进行数据的整理与清洗。

最后，所有的人脸图像都需要通过 OpenCV 的 Cascade 人脸分类器进行人脸检测，使用 Dlib 开源库中的 68 个关键点的检测方法进行嘴唇区域的定位。由于某些人脸检测失败，最后得到了 4841 张微笑图像，4763 张无表情图像，3154 张嘟嘴图像，2348 张大笑的人脸图像，共 15 106 张图，并将图像全部统一缩放为 128 × 128 的大小。下面是核心代码：

```
# 配置 Dlib 关键点检测器路径
PREDICTOR_PATH = "shape_predictor_68_face_landmarks.dat"
predictor = dlib.shape_predictor(PREDICTOR_PATH)
# 配置人脸检测器路径
cascade_path='haarcascade_frontalface_default.xml'
# 初始化分类器
cascade = cv2.CascadeClassifier(cascade_path)
# 调用 cascade.detectMultiScale 人脸检测器和 Dlib 的关键点检测算法 predictor 获得关键点检测结果
def get_landmarks(im):
    rects = cascade.detectMultiScale(im, 1.3,5)        # 进行多尺度检测
    x,y,w,h =rects[0]
    rect=dlib.rectangle(x,y,x+w,y+h)                   # 获得检测框
    # 调用 Dlib 关键点检测
return np.matrix([[p.x, p.y] for p in predictor(im, rect).parts()]
# 打印关键点信息方便调试
def annotate_landmarks(im, landmarks):
    im = im.copy()
    for idx, point in enumerate(landmarks):
        pos = (point[0, 0], point[0, 1])
        cv2.putText(im, str(idx), pos,
                    fontFace = cv2.FONT_HERSHEY_SCRIPT_SIMPLEX,
                    fontScale = 0.4,
                    color=(0, 0, 255))                 # 添加关键点序号
        cv2.circle(im, pos, 5, color = (0, 255, 255)) # 画出关键点
    return im
# 读取并显示图像
im=cv2.imread(sys.argv[1],1)
```

```
cv2.namedWindow('Result',0)
cv2.imshow('Result',annotate_landmarks(im,get_landmarks(im)))
print "image shape = ",im.shape
# 得到 68 个关键点
landmarks = get_landmarks(im)
print "landmarks",landmarks.shape
xmin = 10000
xmax = 0
ymin = 10000
ymax = 0
# 根据最外围的关键点，获取包围嘴唇的最小矩形框
for i in range(48,67):
    x = landmarks[i,0]
    y = landmarks[i,1]
    if x < xmin:
        xmin = x
    if x > xmax:
        xmax = x
    if y < ymin:
        ymin = y
    if y > ymax:
        ymax = y
roiwidth = xmax - xmin
roiheight = ymax - ymin
roi = im[ymin:ymax,xmin:xmax,0:3]
# 将最小矩形框扩大到原来的 1.5 倍，获得最终的矩形框
if roiwidth > roiheight:
    dstlen = 1.5*roiwidth
else:
    dstlen = 1.5*roiheight
diff_xlen = dstlen - roiwidth
diff_ylen = dstlen - roiheight
newx = xmin
newy = ymin
imagerows,imagecols,channel = im.shape
if newx >= diff_xlen/2 and newx + roiwidth + diff_xlen/2 < imagecols:
    newx  = newx - diff_xlen/2;
elif newx < diff_xlen/2:
    newx = 0;
else:
    newx =  imagecols - dstlen;
if newy >= diff_ylen/2 and newy + roiheight + diff_ylen/2 < imagerows:
    newy  = newy - diff_ylen/2;
elif newy < diff_ylen/2:
    newy = 0;
else:
    newy=imagerows-dstlen

# 得到最终的样本
roi = im[int(newy):int(newy+dstlen),int(newx):int(newx+dstlen),0:3]
```

如图 4.29 所示为其中的一些样本展示。

图 4.29　数据集中的不同表情样本

接下来我们将数据均匀地划分为训练集和验证集，并将其存储到 TXT 文件中，数据集的存储格式如下：

```
# 每行信息包括图像路径、空格符号、类别标签
2smile/4353smile.jpg 2
2smile/4288smile.jpg 2
1pouting/2437pouting.jpg 1
2smile/928smile.jpg 2
2smile/4300smile.jpg 2
2smile/2484smile.jpg 2
2smile/2smile.jpg 2
0none/3252none.jpg 0
0none/2061none.jpg 0
2smile/990smile.jpg 2
2smile/4285smile.jpg 2
0none/2748none.jpg 0
0none/891none.jpg 0
2smile/3511smile.jpg 2
1pouting/2024pouting.jpg 1
0none/195none.jpg 0
1pouting/1756pouting.jpg 1
2smile/4268smile.jpg 2
1pouting/2265pouting.jpg 1
```

这是比较通用的分类任务数据格式，即按照图像和标签的格式进行存储。将前面的 15 106 个样本的数据集按照 9 ： 1 分为训练集和测试集，得到训练集 13 597 张图，测试集 1509 张图。其中：

- 无表情类共 4763 张图，训练集 4287 张图，测试集 476 张图。
- 嘟嘴类共 3154 张图，训练集 2839 张图，测试集 315 张图。
- 微笑类共 4841 张图，训练集 4357 张图，测试集 484 张图。
- 大笑类共 2348 张图，训练集 2114 张图，测试集 234 张图。

4.6.3　网络结构配置

经过统计后，我们发现大部分样本的分辨率小于 100，由于本任务相对比较简单，因此我们决定设计一个简单的网络，称为 Simpleconv3，其包含 3 个卷积层，3 个全连接层，每一个卷积层的卷积核大小为 3 × 3，步长为 2，填充大小为 1，输入图像大小设置为 48 × 48。

卷积层的配置如表 4.1 所示。

表 4.1　Simpleconv3 网络卷积层配置

网　络　层	输入特征图尺寸	输出特征图尺寸	卷积核大小	步　　长	填　　充
Conv1	3 × 48 × 48	12 × 24 × 24	3 × 3	2	1
Conv2	12 × 24 × 24	24 × 12 × 12	3 × 3	2	1
Conv3	24 × 12 × 12	48 × 6 × 6	3 × 3	2	1

3 个全连接层的输出神经元数目分别是 512、128 和 4。

使用 Netron 工具对 Caffe 格式的网络进行可视化后的训练网络结构如图 4.30 所示。

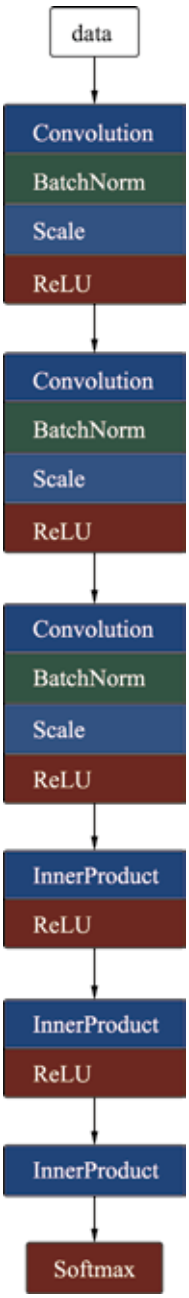


图 4.30　Simpleconv3 模型结构可视化

可以看出，Simpleconv3 模型包含 3 组卷积块，每组卷积块包含一个卷积层 Convolution、一个标准化层 BatchNorm、一个尺度缩放层 Scale 和一个激活层 ReLU，包含 3 个全连接层，前两个全连接层后接 ReLU 激活层，最后一个全连接层作为分类输出层，添加了一个 Softmax 层用于归一化概率。

完整的网络结构配置可以在本书的配套资料中下载。

4.6.4 基于 PyTorch 的项目实践

在准备好数据和模型后，接下来我们使用 PyTorch 框架进行训练，需要完成数据读取，模型训练与验证和可视化等内容。

整个 PyTorch 项目工程包括的文件如图 4.31 所示。

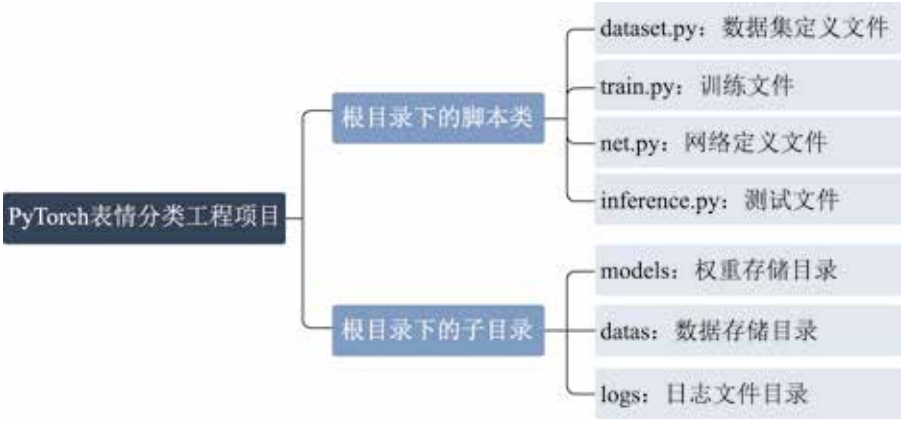


图 4.31 PyTorch 表情分类项目工程内容

1. 数据读取

PyTorch 可以使用 Torchvision 的数据集读取接口进行图像分类任务的读取，首先要将训练集图片和验证集图片分别存储到两个文件夹中，在各自的文件夹中存储若干个子文件夹，每个子文件夹对应一类图片，然后使用 Torchvision 的 Transform 接口进行数据预处理与数据增强，这部分工作在 train.py 中实现，核心代码如下：

```
import torchvision
from torchvision import datasets, models, transforms
data_dir = './data'                    # 数据目录
# 创建数据预处理函数，训练时进行随机裁剪缩放、随机翻转数据增强，随机旋转，验证预处理则使用统一缩放、中心裁剪，训练和验证时都需要进行格式转换与标准化
data_transforms = {
        'train': transforms.Compose(
            transforms.RandomSizedCrop(crop_size),
            transforms.RandomHorizontalFlip(),
            transforms.RandomRotation(20),
            transforms.ToTensor(),
            transforms.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])
```

```
        ]),
        'val' : transforms.Compose([
            transforms.Scale(crop_size),
            transforms.ToTensor(),
            transforms.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])
        ]),
    }


# 使用 torchvision 的 dataset ImageFolder 接口读取数据

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x]) for x
                                          in ['train', 'val']}


# 创建数据指针，设置 batch、shuffle 和多进程数量
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
                                              batch_size = 16,
                                              shuffle = True,
                                              num_workers = 4) for x
                                              in ['train', 'val']}
```

下面对上述代码进行简单讲解。

PyTorch 的 Torchvision 模块中提供了一个 dataset 包，它包含一些基本的数据集如 MNIST、COCO、ImageNet，以及一个通用的分类任务数据加载器 ImageFolder。当使用 ImageFolder 对分类任务的数据进行读取时，只需要将不同类别的数据放置在不同的目录下就可以实现加载。对于这个任务来说，其目录结构如下：

```
├── train
│   ├── 0
│   ├── 1
│   ├── 2
│   └── 3
└── val
    ├── 0
    ├── 1
    ├── 2
    └── 3
```

其中，train 和 val 分别表示根目录，它们都有子目录 0、1、2、3，对应 4 类表情的图像。

在 Torchvision 的 Transforms 模块中定义了一系列的数据集预处理和增强操作，这里我们分别创建了训练和验证用的两个预处理函数。

训练用的预处理操作包括随机裁剪缩放函数 RandomSizedCrop、随机翻转函数 RandomHorizontalFlip、随机旋转函数 RandomRotation、格式转换函数 ToTensor 和标准化函数 Normalize。

其中，RandomSizedCrop 函数会首先按照一定的预设比例在原图中进行裁剪，然后将裁剪后的图片统一缩放到 crop_size × crop_size 大小，crop_size=48。RandomHorizontalFlip 则以 0.5

的概率进行水平翻转。RandomRotation 是对图片随机旋转不超过 20° 。

验证集预处理操作包括尺度缩放函数 Scale，格式转换函数 ToTensor 和标准化函数 Normalize。其中，Scale 将图片统一缩放到 crop_size × crop_size 大小，与训练的输入图像大小一致。

训练集和验证集都需要包含格式转换函数 ToTensor 和标准化函数 Normalize。格式转换函数 ToTensor 将图像像素值从 0 ～ 255 归一化为 0 ～ 1，标准化函数 Normalize 则对图像减去均值向量 [0.5,0.5,0.5]，再除以方差向量 [0.5,0.5,0.5]。

在得到 ImageFolder 对象后，可以使用 torch.utils.data.DataLoader 创建数据指针。DataLoader 可以配置的接口包括批处理大小 batch_size，随机打乱标志 shuffle，多线程加载变量 num_workers。

2. 网络模型定义

接下来在 net.py 中定义一个简单的模型 Simpleconv3。完整的代码如下：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
# Simpleconv3 定义
# 包括 3 个卷积层，3 个 BN 层，3 个 ReLU 激活层，3 个全连接层

class simpleconv3(nn.Module):
    # 初始化函数
    def __init__(self,nclass):
        super(simpleconv3,self).__init__()
        # 输入图片大小为 3*48*48，输出特征图片大小为 12*24*24，卷积核大小为 3*3，
        步长为 2
        self.conv1 = nn.Conv2d(3, 12, 3, 2,1)
        self.bn1 = nn.BatchNorm2d(12)
        # 输入图片大小为 12*24*24，输出特征图片大小为 24*12*12，卷积核大小为 3*3，
        步长为 2
        self.conv2 = nn.Conv2d(12, 24, 3, 2,1)
        self.bn2 = nn.BatchNorm2d(24)
        # 输入图片大小为 24*11*11，输出特征图片大小为 48*6*6，卷积核大小为 3*3，步
        长为 2
        self.conv3 = nn.Conv2d(24, 48, 3, 2,1)
        self.bn3 = nn.BatchNorm2d(48)
        # 输入向量长为 48*6*6 = 1728，输出向量长为 512
        self.fc1 = nn.Linear(48 * 6 * 6 , 512)
        self.fc2 = nn.Linear(512 , 128) # 输入向量长为 512，输出向量长为 128
        # 输入向量长为 128，输出向量长为 nclass，等于类别数
        self.fc3 = nn.Linear(128 , nclass)

    # 前向函数
    def forward(self, x):
        # ReLU 函数不需要进行实例化，可以直接调用
        # Conv，Fc 层需要调用 nn.Module 进行实例化
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
```

```
        x = x.view(-1 , 48 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

这里定义的网络 Simpleconv3 是一个简单的 3 层网络，包括 3 个卷积层，3 个 BN 层，3 个 ReLU 激活层，3 个全连接层，要求输入的图像大小是 $3 \times 48 \times 48$，每一层特征图的大小可以使用 print 函数来查看。

每一个网络定义需要实现两个函数，即 __init__ 与 forward。在 __init__ 函数中首先调用 super(simpleconv3,self).__init__ 函数进行初始化，由于 PyTorch 的网络层包含在 nn.Module 包中，所有的网络定义都需要继承该网络层。然后进行网络的定义，其中，Conv 和 Fc 层需要调用 nn.Module 进行实例化，因此需要在 __init__ 函数中定义，而 ReLU 等没有学习参数的函数不需要进行实例化，可以直接调用，不需要声明。

网络定义在 nn 包中。完整的接口如下：

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride = 1,
padding = 0, dilation = 1, groups = 1, bias = True)
```

其中，输入参数包括输入通道数 in_channels，输出通道数 out_channels，卷积核大小 kernel_size，步长 stride，填充 padding，膨胀因子 dilation，分组因子 groups 和是否保留偏置项 bias。

在定义好网络层后，就可以在 forward 函数中实现了。

3. 优化方法与优化目标的定义

接下来看优化方法和优化目标的定义。在 train.py 中添加实现代码：

```
criterion = nn.CrossEntropyLoss()
optimizer_ft = optim.SGD(model.parameters(), lr = 0.1, momentum = 0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size = 100,
gamma = 0.1)
```

可以看出，优化目标使用了交叉熵，优化方法使用带动量项的 SGD，学习率迭代策略为 step，每隔 100 轮（即 100 个 epoch），变为原来的 1/10。

4. 添加可视化代码

为了方便监控训练过程，可以使用 TensorboardX 进行可视化。TensorboardX 的具体使用分为三步。

（1）引入包定义，创建变量。

```
from tensorboardX import SummaryWriter
writer = SummaryWriter()
```

（2）记录变量，如 train 阶段的 loss。

```
writer.add_scalar('data/trainloss', epoch_loss, epoch)
```

（3）在终端根据提示打开 TensorboardX，如打开日志目录 logs 下的文件，PyTorch 默认的日志目录为 runs。

```
tensorboard --logdir = logs
```

然后在浏览器中根据提示打开网页。这部分代码被添加在 train.py 中。

5. 模型训练

接下来实现模型的训练主函数，这部分代码在 train.py 中。核心代码如下：

```
# 训练主函数
def train_model(model, criterion, optimizer, scheduler, num_epochs = 25):
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train(True)                          # 设置为训练模式
            else:
                model.train(False)                         # 设置为验证模式

            running_loss = 0.0                             # 损失变量
            running_accs = 0.0                             # 精度变量
            num_batchs = 0                                 # batch 计数

            # 从 dataloaders 中获得数据
            for data in dataloaders[phase]:
                inputs, labels = data
                if use_gpu:
                    inputs = inputs.cuda()
                    labels = labels.cuda()

                optimizer.zero_grad()                      # 清空梯度
                outputs = model(inputs)                    # 前向运行
                # 使用 max 函数对输出值进行操作，得到预测值索引
                _, preds = torch.max(outputs.data, 1)
                loss = criterion(outputs, labels)          # 计算损失
                if phase == 'train':
                    loss.backward()                        # 误差反向传播
                    optimizer.step()                       # 参数更新

                running_loss += loss.data.item()
                running_accs += torch.sum(preds == labels).item()
                num_batchs += 1

            # 得到每轮的平均损失与精度
            epoch_loss = running_loss / num_batchs
            epoch_acc = running_accs / dataset_sizes[phase]

            # 收集精度和损失用于可视化
            if phase == 'train':
                writer.add_scalar('data/trainloss', epoch_loss, epoch)
                writer.add_scalar('data/trainacc', epoch_acc, epoch)
            else:
                writer.add_scalar('data/valloss', epoch_loss, epoch)
                writer.add_scalar('data/valacc', epoch_acc, epoch)

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
```

```
                phase, epoch_loss, epoch_acc))

        writer.close()
    return model
```

下面来分析一下上面的代码，外层循环是 epoches，然后利用 for data in dataloders[phase] 循环取一轮的数据并送入模型。需要注意的是，每次 forward 要将梯度清零，通过 optimizer.zero_grad 函数实现，因为梯度会记录前一次的状态，然后计算损失进行反向传播。

最后还需要在 train.py 中配置一些训练用的相关参数，包括 batchsize 大小，图像缩放大小，裁剪大小，以及是否使用 GPU 进行训练。代码如下：

```
crop_size = 48                          # 图像裁剪大小，即训练输入大小
nclass = 4                              # 分类类别数
model = simpleconv3(nclass)             # 创建模型
data_dir = './data'                     # 数据目录
# 模型缓存接口
if not os.path.exists('models'):
    os.mkdir('models')
# 检查 GPU 是否可用，如果可用则使用 GPU，否则使用 CPU
use_gpu = torch.cuda.is_available()
if use_gpu:
    model = model.cuda()
```

完整的训练代码请大家去本项目配套的开源项目中获取。

运行 train.py 即可进行训练，同时启动 TensorboardX 可以在浏览器中进行实时监控，可以判断模型的收敛情况。

训练了 300 轮之后，得到如图 4.32 和图 4.33 所示的结果。

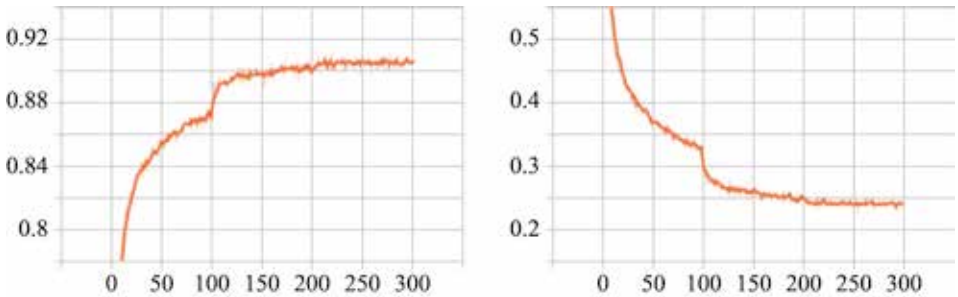


图 4.32　训练集精度（左）与损失（右）曲线

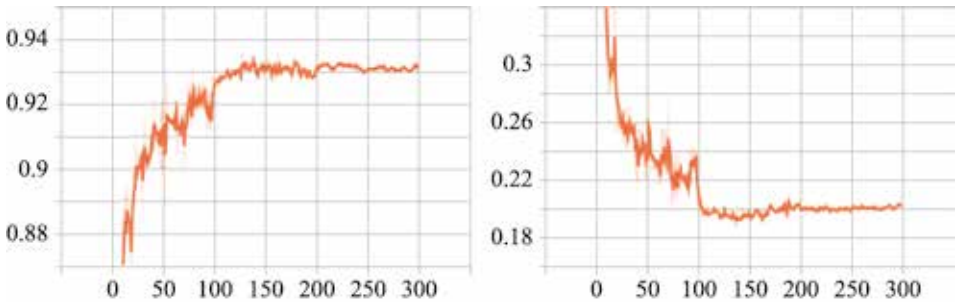


图 4.33　验证集精度（左）与损失（右）曲线

从结果图中可以看出，模型已经收敛，但是存在一定程度的过拟合，验证集的最终精度约为 0.93。

6. 模型测试

接下来的目标是将训练好的模型进行推理，真正把模型用起来。下面在 inferency.py 脚本中编写测试脚本，载入一张图片，用模型进行测试。

```
==========
import sys
import numpy as np
import cv2
import os
import dlib
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.autograd import Variable
import torchvision
from torchvision import datasets, models, transforms
import time
import os
from PIL import Image
import sys
import torch.nn.functional as F

testsize = 48                          # 测试图大小
from net import simpleconv3
net = simpleconv3(4)                   # 定义模型
# 设置推理模式，使 dropout 和 batchnorm 等网络层可以在 train 和 val 模式之间切换
net.eval()
torch.no_grad()        # 停止 autograd 模块的工作，从而加速计算，节省显存

# 载入模型权重
modelpath = 'model.pt'
net.load_state_dict(torch.load(modelpath,map_location = lambda storage,
loc: storage))

# 定义预处理函数
data_transforms = transforms.Compose([
            transforms.Resize(48),
            transforms.ToTensor(),
            transforms.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])])

# 读取 3 通道图片并扩充为 4 通道 tensor
imagepath = 'test.jpg'
image = Image.open(imagepath)
imgblob = data_transforms(image).unsqueeze(0)

# 获得预测结果 predict，得到预测的标签值 label
predict = net(imgblob)
```

```
index = np.argmax(predict.detach().numpy())
print(predict)
print(index)
```

从上面的代码可知，需要做的事情包括：

- 定义网络并使用 torch.load 和 load_state_dict 载入模型。
- 使用 net.eval 函数设置推理模式，使得 dropout 和 batchnorm 等网络层在 train 和 val 模式间切换，使用 torch.no_grad 函数停止 autograd 模块的工作，达到加速计算和节省显存的目的。
- 使用 PIL 库 Image 读取图片，将图片读取为 RGB 格式，然后使用数据预处理函数将其归一化到 0 ～ 1 并转换为 Tensor 变量。

之后就可以自己输入图片得到推理结果了，predict 就是模型的输出，index 就是预测的类别。

4.6.5　基于 Caffe 的项目实践

接下来使用 Caffe 框架进行训练，并与 PyTorch 框架进行比较，这是为了兼容笔者已经出版的《深度学习之图像识别：核心技术与案例实战》以及满足部分读者的需求，但是后面的项目都将采用 PyTorch 来完成。

我们仍然需要完成数据读取、模型训练与验证和可视化等内容，需要编写的文件包括 train.prototxt、deploy.prototxt、solver.prototxt 和 inference.py，同时还需要修改 Caffe 框架的一些源代码。如图 4.34 所示为 Caffe 表情分类项目工程文件。

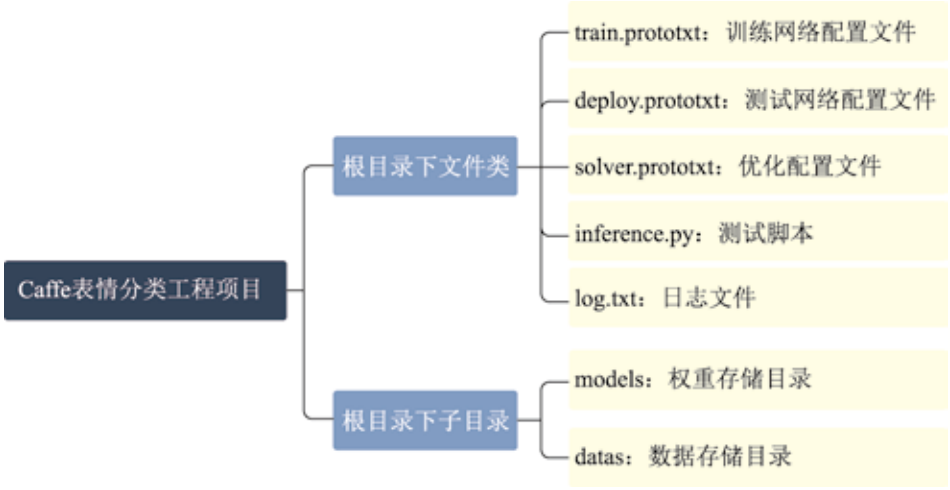


图 4.34　Caffe 表情分类项目工程文件

1. 数据读取

首先我们需要使用前面准备好的训练文本文件和验证文本文件，分别是 all_shuffle_train.txt 和 all_shuffle_test.txt，然后在 train.prototxt 中配置网络结构。

训练时的数据输入层如下：

```
# 网络数据层，包括训练数据和验证数据
name: "mouth"
layer {
    name: "data"
    type: "ImageData"
    top: "data"
    top: "label"
    image_data_param {
        source: "all_shuffle_train.txt"
        batch_size: 64
        shuffle: true
    }
    transform_param {
        mean_value: 104.008
        mean_value: 116.669
        mean_value: 122.675
        min_side_min: 48
        min_side_max: 96
        max_rotation_angle: 20
        crop_size: 48
        mirror: true
    }
    include: { phase: TRAIN}
}
layer {
    name: "data"
    type: "ImageData"
    top: "data"
    top: "clc-label"
    image_data_param {
        source: "all_shuffle_val.txt"
        batch_size: 16
        shuffle: false
    }
    transform_param {
        mean_value: 104.008
        mean_value: 116.669
        mean_value: 122.675
        crop_size: 48
        min_side_min: 48
        min_side_max: 48
        mirror: false
    }
    include: { phase: TEST}
}
```

可以看到，上面定义了一个 ImageData 数据读取层，它的输出包括 data 和 label，包含 image_data_param 和 transform_param 两个配置参数。

在 image_data_param 中配置了训练文件路径、批处理大小及是否对数据进行 shuffle。

在 transform_param 中配置了 3 个 mean_value，分别对应 RGB 图像的 BGR 通道，表示需要进行减均值预处理操作。

在 transform_param 中，配置了 min_side_min=48 和 min_side_max=96 两个参数，表示对图像进行预先缩放，然后再用于裁剪操作。缩放后的最短边的边长范围必须在 min_side_min 和 min_side_max 之间。

在 transform_param 中，配置了 crop_size=48，表示图像裁剪的输入大小，它也是真正用于模型训练的输入大小。

在 transform_param 中，配置了 mirror 属性设置为 true，表示进行随机水平翻转操作。

在 transform_param 中，配置了 max_rotation_angle=20，表示进行随机旋转的最大角度为 ±20° 。

与 PyTorch 的数据预处理进行对比，发现 Caffe 使用了类似的策略，即随机缩放裁剪、随机水平翻转与随机旋转操作，其中，随机缩放裁剪操作和随机旋转操作在 Caffe 官方框架中没有实现，我们需要自己实现。

首先需要在根目录下的 src/caffe/proto/caffe.proto 文件的 TransformationParameter 中注册相关变量：

```
message TransformationParameter {
    optional uint32 max_rotation_angle = 10 [default = 0];
    optional uint32 min_side_min = 13 [default = 0];
    optional uint32 min_side_max = 21 [default = 0];
       }
```

以上代码注册了 3 个整型变量并将其默认值赋值为 0，重新编译 Caffe 项目后就可以在 transform_param 中配置参数了。而操作的实现则在 data_transformer.cpp 中，我们从中找到下面的函数：

```
void DataTransformer<Dtype>::Transform(const cv::Mat& img, Blob<Dtype>*
transformed_blob)
```

然后在其中添加随机缩放和随机旋转相关的代码，核心的 C++ 代码如下：

```
const int crop_size = param_.crop_size();        // 获得裁剪尺寸大小
const int min_side_min = param_.min_side_min(); // 获得最小边的最小值配置
CHECK(min_side_min >= crop_size);                // 必须大于或等于裁剪尺寸
const int min_side_max = param_.min_side_max(); // 获得最小边的最大值配置
CHECK(min_side_max >= min_side_min);      // 最小边的最大值配置必须大于最小值
// 是否使用变形的缩放
const bool use_deformed_resize = param_.use_deformed_resize();
float current_apply_resize_prob;           // 应用缩放操作的概率
// 生成 0 ~ 1 的概率
caffe_rng_uniform(1, 0.f, 1.f, &current_apply_resize_prob);
// 使用变形缩放的阈值概率
const float deformed_resize_th = param_.deformed_resize_th();
const int rotation_angle = param_.max_rotation_angle();    // 随机旋转角度
// 是否使用随机旋转
const bool do_rotation = rotation_angle > 0 && phase_ == TRAIN;

// 随机缩放操作模块
int newsize = 0;
if(phase_ == TRAIN)
```

```
        newsize = min_side_min + Rand(min_side_max - min_side_min + 1);
    else
        newsize = crop_size;
    if(use_deformed_resize && (current_apply_resize_prob >= deformed_
    resize_th))
        adaptedresize(cv_img, newsize,true);
    else
        adaptedresize(cv_img, newsize,false);

    // 随机旋转操作模块
    int current_angle = 0;
    if (do_rotation) {
        current_angle = Rand(rotation_angle*2 + 1) - rotation_angle;
        rotate(cv_img, current_angle);
    }
```

仔细阅读上面的数据缩放逻辑可知，缩放函数都是 adaptedresize，它通过最后的参数标志来实现变形缩放或者保持短边的缩放操作。

对于验证阶段，均将图像短边统一缩放到 crop__size 大小。对于训练阶段，则会生成一个在 min_size_min 和 min_side_max 之间的值作为最短边尺寸 new_size。

当使用变形缩放标志 use_deformed_resize 为 True，并且生成的随机概率 current_apply_resize_prob 大于配置的阈值 deformed_resize_th 时，使用变形缩放，否则使用保持图像长宽比的缩放。use_deformed_resize 的默认值为 false，deformed_resize_th 的默认值是 0.5，可见即使配置了 use_deformed_resize 为 true，也只有 0.5 的概率使用变形缩放。

adaptedresize 函数的定义如下：

```
    void adaptedresize(cv::Mat& cv_img, int smallest_side, const bool &use_
    deformed_resize) {
             int cur_width = cv_img.cols;
             int cur_height = cv_img.rows;
             cv::Size dsize;
             if(use_deformed_resize)
                    cv::resize(cv_img, cv_img, cv::Size(smallest_
    side,smallest_side),cv::INTER_NEAREST);
             else{
                    if (cur_height <= cur_width) {
                           double k = ((double)cur_height) / smallest_side;
                           int new_size = (int) ceil(cur_width / k);
                           dsize = cv::Size(new_size, smallest_side);
                    }else {
                           double k = ((double)cur_width) / smallest_side;
                           int new_size = (int) ceil(cur_height / k);
                           dsize = cv::Size(smallest_side, new_size);
                    }
                    cv::resize(cv_img, cv_img, dsize, cv::INTER_NEAREST);
             }
       }
```

通过上面的数据裁剪与缩放操作，可以控制在训练时对图像进行不同尺度的缩放操作，实现多尺度的裁剪。

rotate 的定义如下：

```
   void rotate(cv::Mat& src, int angle) {
         cv::Point2f center(src.cols / 2.0, src.rows / 2.0);
         cv::Mat rot = cv::getRotationMatrix2D(center, angle, 1.0);
         cv::Rect bbox = cv::RotatedRect(center, src.size(), angle).
boundingRect();
         rot.at<double>(0, 2) += bbox.width / 2.0 - center.x;
         rot.at<double>(1, 2) += bbox.height / 2.0 - center.y;
         cv::warpAffine(src, src, rot, bbox.size());
   }
```

通过上面的旋转函数，可以控制图像旋转某一个角度。

验证时的数据层格式相同，不同的是取消了 shuffle、mirror 和 rotate 操作，去掉了随机裁剪缩放操作，采用了固定的裁剪操作，即 min_size_min=min_side_max=crop_size=48。

2. 网络层定义

接下来我们需要实现网络的定义，主要包括卷积层与全连接层，由于完整的模型配置代码过长，这里只展示第 1 个卷积层和第 1 个全连接层的配置文件。

第 1 个卷积层的定义如下：

```
layer {
    name: "conv1"
    type: "Convolution"
    bottom: "data"
    top: "conv1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 12
        pad: 1
        kernel_size: 3
        stride: 2
        weight_filler {
            type: "xavier"
            std: 0.01
        }
        bias_filler {
            type: "constant"
```

```
            value: 0.2
        }
    }
}
```

可以看出类型是 Convolution，输入是 data。在卷积层的参数 convolution_param 变量中配置了相关参数，包括输出通道数 num_output 为 12，卷积核大小 kernel_size 为 3，步长 stride 为 2，填充 pad 为 1，权重初始化策略 weight_filler 选择 xavier 初始化，方差为 0.01，偏置初始化选择固定值 0.2。

第 1 个全连接层的定义如下：

```
layer {
    name: "ip1"
    type: "InnerProduct"
    bottom: "conv3"
    top: "ip1"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 0
    }
    inner_product_param {
        num_output: 512
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}
```

完整的网络结构，可以去本书配置的 GitHub 资料中获取。

3. 优化参数定义

在 solver.prototxt 的配置中，采用 SGD 优化方法进行训练，初始的 lr=0.1，momentum= 0.9，batchsize=96，使用的优化目标为 Softmax 损失，solver.prototxt 的完整配置如下：

```
net: "mobilenet_train.prototxt"                    # 网络名称
base_lr: 0.1                                       # 初始学习率
momentum: 0.9                                      # 动量项
type: "SGD"                                        # 优化方法
lr_policy:  "step"                                 # 学习率变更策略
```

```
stepvalue:14163                                    # 学习率变更的 batch 步长
display: 100                                       # 显示间隔
max_iter: 100000                                   # 最大迭代次数
snapshot: 1000                                     # 缓存间隔
snapshot_prefix: "./models/simpleconv3"            # 缓存前缀
solver_mode: GPU                                   # 使用 GPU 进行训练
```

Softmax 损失层的定义如下：

```
layer {
    bottom: "ip3"
    bottom: "label"
    name: "loss"
    type: "SoftmaxWithLoss"
    top: "loss"
}
```

其中，stepvalue=100 × 13597/96，13597 是训练数据集图片数量，96 是每次训练输入的图片数量，这可以实现每隔 100 轮学习率就下降，与 PyTorch 训练的策略保持一致。

4. 模型训练

训练时使用 C++ 接口与预训练模型，代码如下：

```
SOLVER=./mobilenet_solver.prototxt                 # 优化文件
/build/tools/caffe train -solver $SOLVER -weights $WEIGHTS -gpu 0 2>&1
 | tee log.txt
```

在训练时记得要将 log 存储到日志文件中，因为后面需要进行结果分析与查看。

训练样本总数为 13 597 个，选择的 batchsize=96，下面是训练了 100 000 次迭代，也就是 100 000 × 96/13 597，约等于 700 轮之后的结果。

训练集与验证集的损失和精度变化情况如图 4.35 和图 4.36 所示。

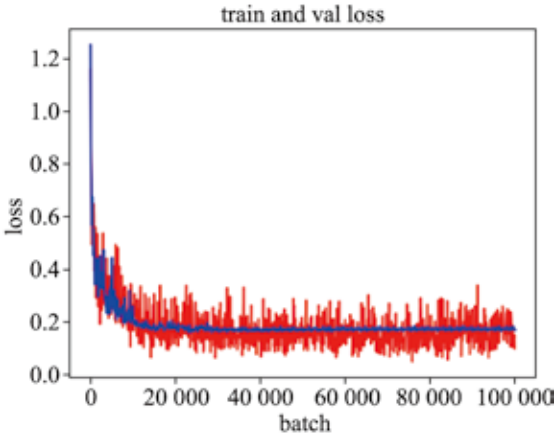


图 4.35　损失曲线

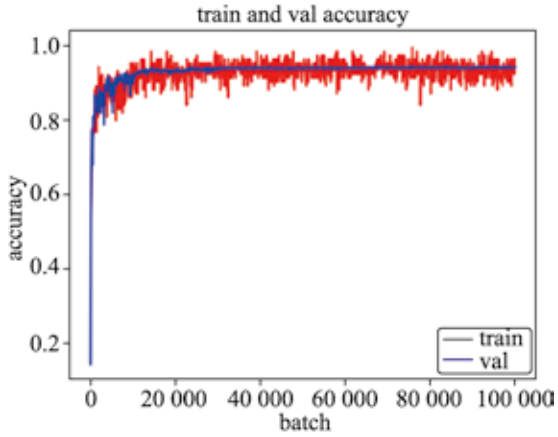


图 4.36　精度曲线

验证集损失最终稳定在 0.18 左右，验证集精度稳定在 94.5% 左右。通过图 4.35 和图 4.36 可以看出模型没有过拟合，训练集和验证集的收敛情况相当，在 40 000 个批次左右时已经基本收敛完毕。

5. 测试结果

接下来我们对模型进行测试，在测试的时候也需要输入网络结构，定义在 deploy.prototxt 中，它与 train.prototxt 的主要不同之处在于输入数据层格式不同并且没有精度层与损失层。

测试时的数据输入层如下：

```
# 网络数据层
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 1 dim: 3 dim: 48 dim: 48 } }
```

与训练数据输入层相比，训练时该层的数据类型是 ImageData，且需要配置一系列与训练相关的参数，包括数据标签、文件路径、批处理大小和数据增强策略等。而测试数据输入层只需要配置数据的输入尺寸即可，即上述代码中的 shape 属性，表示输入需要的单张图片尺寸大小是 $1 \times 3 \times 48 \times 48$。

测试时的最后一个全连接层与输出层如下：

```
layer {
    bottom: "ip2"
    top: "ip3"
    name: "ip3"
    type: "InnerProduct"
    param {
        lr_mult: 1
        decay_mult: 1
    }
    param {
        lr_mult: 2
        decay_mult: 1
    }
    inner_product_param {
        num_output: 4
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
            value: 0
        }
    }
}

layer {
    bottom: "ip3"
    name: "prob"
```

```
    type: "Softmax"
    top: "prob"
}
```

ip3 为最后一个全连接层的名称，其类型是 InnerProduct，它包含了一个全连接层参数配置。其中，输出通道数 num_output 为 4，说明模型是一个 4 分类模型。

最后一个网络层名称为 prob，类型为 Softmax，表示这是一个使用 Softmax 函数进行映射后的张量，它的通道维度为 4，每一个维度的数值大小在 0 ～ 1 之间。其中，数值最大的通道索引就是对应的预测类别标签。

测试核心 Python 代码如下：

```
def start_test(model_proto,model_weight,imgtxt,testsize,enable_crop):
    # 初始化网络
    caffe.set_device(0)                                  # 使用第一个 GPU
    net = caffe.Net(model_proto, model_weight, caffe.TEST)
    imgs = open(imgtxt,'r').readlines()                  # 获取图像
    count = 0
    acc = 0
    # 遍历图像
    for imgname in imgs:
        imgname, label = imgname.strip().split(' ')
        imgtype = imgname.split('.')[-1]
        if imgtype != 'png' and imgtype != 'jpg' and imgtype != 'JPG' and
        imgtype != 'jpeg' and imgtype != 'tif' and imgtype != 'bmp':
            print imgtype," error"
            continue
        imgpath = imgname
        img = cv2.imread(imgpath)
        if img is None:
            print "---------img is empty---------",imgpath
            continue
        imgheight, imgwidth, channel = img.shape
        # 选择使用裁剪或者缩放的方案
        if enable_crop == 1:
            print "use crop"
            cropx = (imgwidth - testsize) / 2
            cropy = (imgheight - testsize) / 2
            img = img[cropy:cropy+testsize,cropx:cropx+testsize,0:channel]
        else:
            img = cv2.resize(img,(testsize,testsize),interpolation=cv2.
INTER_NEAREST)

        # 减去均值等预处理
        transformer = caffe.io.Transformer({'data': net.blobs['data'].
data.shape})
        transformer.set_mean('data', np.array([104.008,116.669,122.675]))
        transformer.set_transpose('data', (2,0,1))
```

```
        # forward 获取结果
        out = net.forward_all(data=np.asarray([transformer.preprocess
('data', img)]))

        # 等到最终分类的结果
        result = out['classifier'][0]
        print "result=", result
        predict = np.argmax(result)
        if str(label) == str(predict):
            acc = acc + 1
        count = count + 1
    print "acc=", float(acc) / float(count)
```

注意，在测试时可以选择将所有图片缩放到 60 × 60 的尺寸，然后裁剪中间的 48 × 48 的中心裁剪方案，也可以直接将所有图缩放到 48 × 48 的尺寸。前者的测试精度往往更高一些，因为嘴唇以外的背景区域更小，有助于提高模型的测试精度。

4.6.6 项目总结

本节中我们选择了表情分类项目，从比较简单的任务开始，直观地体验了一个分类任务的完整训练流程。考虑到实际应用需要，我们自定义了一个非常小的模型。

因为是本书的第一个项目，所以本项目的讲解非常详细，从数据的整理到模型的搭建和可视化，并且对比了 PyTorch 和 Caffe 框架的实验结果，后面的项目不会再进行如此详细的讲解。

我们对本任务的训练做一个总结，有几个地方是需要注意的。

1. 模型输入的大小

嘴唇表情识别，这个任务本身并不难，为了在移动端部署，我们应该考虑较小的分辨率，这里使用了 48 这个尺寸，这样只需要 3 个步长为 2 的卷积层就可以提取特征，这是一个非常小的模型，模型体积小于 4MB，能取得 94% 左右的精度。读者可以去实践使用更大的模型，更大的输入尺寸，获得更高的精度。

2. 数据增强

前面采用了 48 作为网络的输入尺度，但是实际上我们准备的数据尺度大多在 100 × 100 左右，我们应用了随机缩放后裁剪、随机旋转与随机翻转数据增强操作，这是在分类任务中经常使用的几个数据增强操作。在 ImageNet 分类任务中，研究人员通常采用将图像缩放到短边为 256 再裁剪 224 × 224 这个尺度作为网络的输入，这样可以实现最小 33 × 33 倍的数据扩增。

表情识别目前的关注点已经从实验室环境转移到具有挑战性的真实场景条件下，研究者开始利用深度学习技术来解决如光照变化、遮挡、非正面头部姿势等问题，仍然有很多的问题需要解决。

另一方面，尽管目前已有学者或机构在研究表情识别技术，但是我们所定义的表情只涵盖特定种类的一小部分，尤其是面部表情，而实际上人类还有很多其他未被我们定义的表情。本节研究的表情是基于嘴唇的表情，而实际上，一个表情不仅仅依靠嘴唇来表达，因为人的表情有非常多的种类，如眉毛、鼻子等都会反映出人的表情。

表情的研究相对于颜值、年龄等要难得多，应用也广泛得多，相信这几年会不断出现有意思的应用，值得大家去研究。

4.7　鸟类细粒度图像分类实战

上一节完成了一个简单的表情图像分类任务，类间方差较大，类内方差很小，属于图像分类中难度较低的项目。本节需要完成一个较难的细粒度图像分类任务，类内方差和类间方差都很大，我们将从项目背景、方案选择、模型训练与参数实验等几个方向进行阐述。

4.7.1　项目背景

现实世界很多的分类问题都是粒度较小的，例如对不同种类的植物进行识别，对不同种类的车型进行识别，对不同种类的野生动物进行识别，其中，动物和植物的识别对于生态保护工作有很大的意义。以动物来说，野生动物种类多，活动范围广，很多都是濒临灭绝的物种，人类专家识别有很大的难度。例如不同种类的燕鸥，它们之间的差异很小，如果只依赖于领域专家的知识，则成本太高，如果能够自动地发现并识别它们，则可以减少相关工作者的负担，本节选择鸟类的分类作为项目任务。

由于细粒度数据集的标注成本比较高，我们选择公开的数据集 Caltech-UCSD Birds-200-2011 来完成本次实践的内容。数据集可以通过网址 http://www.vision.caltech.edu/visipedia/CUB-200-2011.html 来获取。

Caltech-U 数据集包括 200 种鸟类，共 11 788 张图片，每一个种类约 60 个样本，图片的分辨率大小不等，长度都在 400 分辨率左右。除了类别信息之外，数据集还标注了鸟类的 15 个身体部位和 312 个分割属性及回归框等，如图 4.37 所示为一些样本和标注案例，在接下来的模型训练中，我们只采用类别标注信息。

图 4.37　数据集的一些样本

在接下来的训练中，我们按照 9∶1 的比例将每一个类别进行均匀拆分，把数据集分成测试集和训练集。测试集中的每一类约为 6 张图像，共 200 类，1191 个样本。在整个训练过程中，不从该数据集以外增加数据。

下面是一些数据和标签案例：

```
# 每行信息包括图像路径、空格符号、类别标签
```

```
176.Prairie_Warbler/Prairie_Warbler_0063_172682.jpg 175
076.Dark_eyed_Junco/Dark_Eyed_Junco_0102_67402.jpg 75
009.Brewer_Blackbird/Brewer_Blackbird_0133_2324.jpg 8
074.Florida_Jay/Florida_Jay_0097_64906.jpg 73
085.Horned_Lark/Horned_Lark_0126_74354.jpg 84
100.Brown_Pelican/Brown_Pelican_0074_93692.jpg 99
048.European_Goldfinch/European_Goldfinch_0094_794673.jpg 47
048.European_Goldfinch/European_Goldfinch_0080_33322.jpg 47
096.Hooded_Oriole/Hooded_Oriole_0105_90875.jpg 95
```

数据集类的定义与实现如下：

```
class BirdDataset(Dataset):
    def __init__(self, data_dir, filelist="train.txt", transform=None):
        """
        data_dir: 数据集所在路径
        filelist: 图片路径与标签文本文件
        transform: 数据预处理函数
        """
        self.filelist = filelist
        self.data_dir = data_dir
        self.image_labels = self._get_img_label() # 存储所有图片路径和标签
        self.transform = transform

    def __getitem__(self, index):
        imgpath, label = self.image_labels[index]
        img = Image.open(imgpath).convert('RGB')
        if self.transform is not None:
            img = self.transform(img)
        return img, label

    def __len__(self):
        if len(self.image_labels) == 0:
            raise Exception("\ndata_dir:{} is a empty dir!".format
(self.data_dir))
        return len(self.image_labels)

    def _get_img_label(self):
        image_labels = []
        datas = open(self.filelist).readlines()
        for data in datas:
            imgpath,label = data.strip().split(' ')
            image_labels.append((os.path.join(self.datadir,imgpath),
int(label)))
        return image_labels
```

4.7.2 模型搭建

选择的模型是以 VGG 系列网络为主干的双线性分类模型，并与 VGG 系列网络进行比较，

下面我们来搭建双线性模型，核心代码如下：

```
class BCNN(torch.nn.Module):
    def __init__(self):
        torch.nn.Module.__init__(self)
        # 提取 VGG 的特征层，去掉最后一个池化层
        self.features = torchvision.models.vgg11(pretrained=True).features
        self.features = torch.nn.Sequential(*list(self.features.
children())[:-1])
        self.fc = torch.nn.Linear(512**2, 200)
        for param in self.features.parameters():
            param.requires_grad = True
        torch.nn.init.kaiming_normal_(self.fc.weight.data)
        if self.fc.bias is not None:
            torch.nn.init.constant_(self.fc.bias.data, val = 0)

    def forward(self, X):
        N = X.size()[0]
        assert X.size() == (N, 3, 224, 224)
        X = self.features(X)
        assert X.size() == (N, 512, 14, 14)
        X = X.view(N, 512, 14**2)

        # 使用矩阵相乘实现双线性操作
        X = torch.bmm(X, torch.transpose(X, 1, 2)) / (14**2)
        assert X.size() == (N, 512, 512)
        X = X.view(N, 512**2)
        X = torch.sqrt(X + 1e-5)
        X = torch.nn.functional.normalize(X)
        X = self.fc(X)
        assert X.size() == (N, 200)
        return X

if __name__ == '__main__':
    model = BCNN()
    model.load_state_dict(torch.load('models/model.pt', map_location =
lambda storage, loc: storage))
    model.train(False)
    model.eval()
    input = torch.randn((1, 3, 224, 224))
    torch.onnx.export(model, input, "model.onnx", verbose = False)
    output = model(input)
    print(output.shape)
```

在上述代码中首先提取 VGG 的特征层并去掉最后一个步长为 2 的池化层作为主干模型，当输入图像大小为 224×224 时，提取到的图像特征大小为 $512 \times 14 \times 14$，接下来该特征变形为 512×196 的矩阵，再与转置后的 196×512 矩阵相乘，得到 512×512 的矩阵，再展开为 262 144 长度的向量。

如图 4.38 展示了模型转换为 ONNX 格式后的可视化结果。

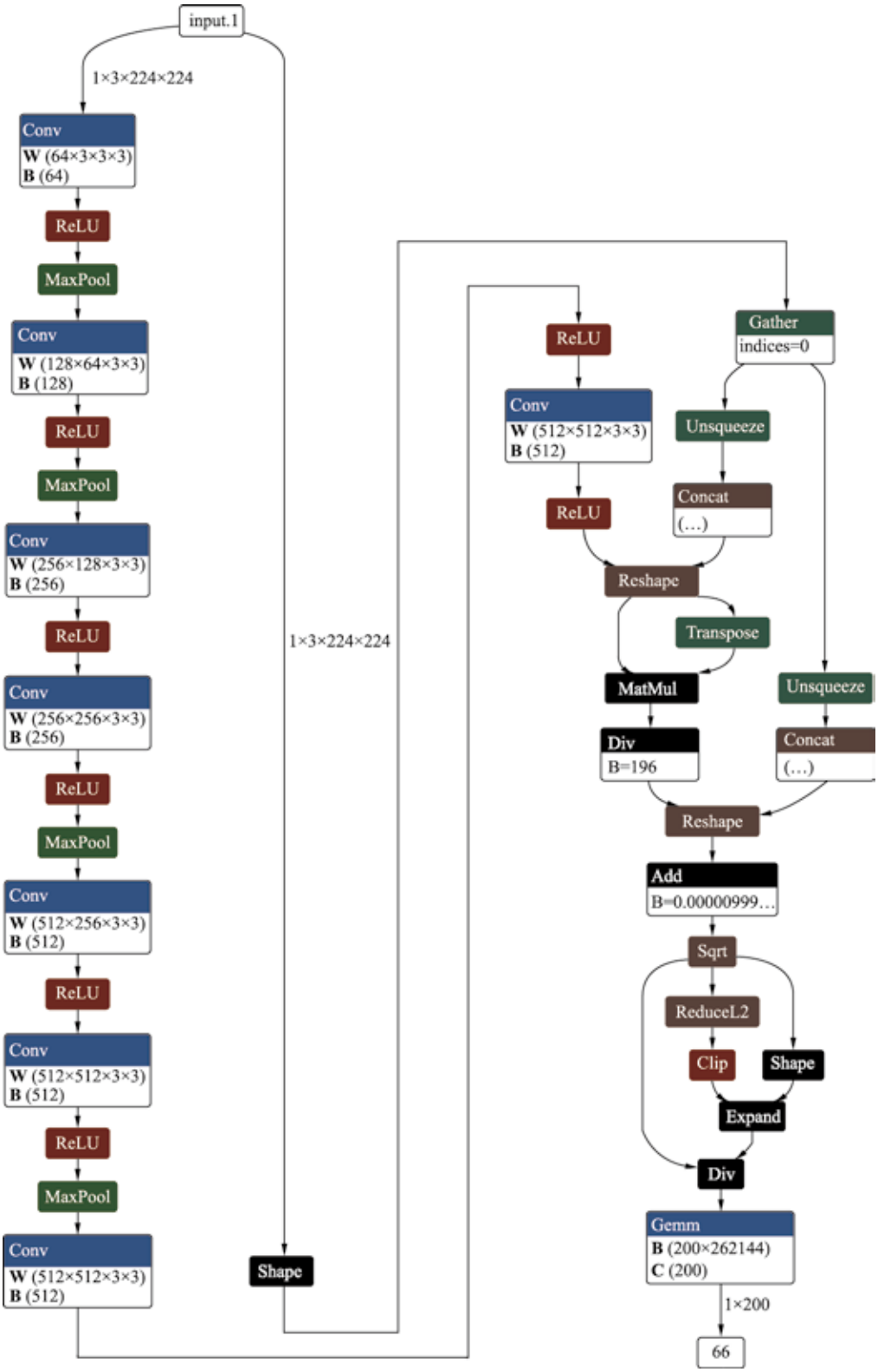


图 4.38　双线性网络可视化结果

4.7.3　模型训练

接下来我们对模型进行训练，一些预处理相关的参数如下：

```
image_size = 256                    # 缩放图像大小
crop_size = 224                     # 图像裁剪大小，即训练输入大小
nclass = 200                        # 分类类别数

# 创建数据预处理函数，训练预处理包括随机裁剪缩放、随机翻转、随机旋转和归一化，验证预处理包括缩放和标准化
data_transforms = {
        'train': transforms.Compose([
            transforms.Resize([image_size, image_size]),
            transforms.RandomSizedCrop(crop_size),
            transforms.RandomHorizontalFlip(),
            transforms.RandomRotation(20),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
        ]),
        'val': transforms.Compose([
            transforms.Resize([crop_size, crop_size]),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
        ]),
    }
```

优化目标使用交叉熵，优化方法使用带动量项的 SGD，学习率迭代策略为 step。

对于双线性模型的训练，每隔 100 轮，学习率变为原来的 1/10，训练批处理大小为 32，使用单卡进行训练。

对于 VGG 模型的训练，只将最后一个全连接层输出维度更改为 200，其他结构不变，每隔 20 轮，学习率变为原来的 1/10，训练批处理大小为 32，使用单卡进行训练。

如图 4.39 和图 4.40 展示了 VGG 基准模型和双线性模型的验证集精度结果。

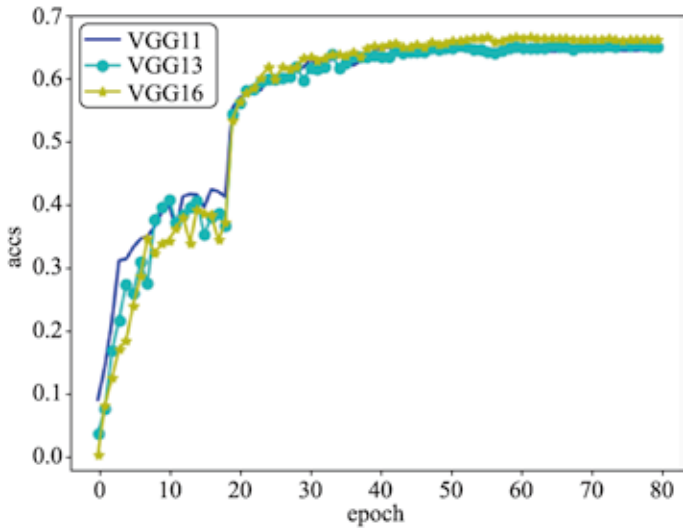


图 4.39　VGG 模型验证集精度

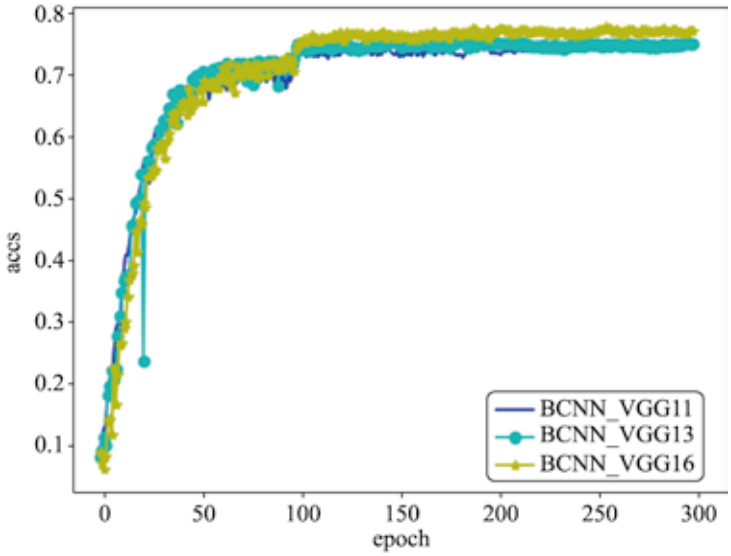


图 4.40　双线性模型验证集精度

从结果中可以看出，VGG11 模型、VGG13 模型和 VGG16 模型都只能达到 0.7 以下的精度，VGG16 精度约为 0.65。而 VGG11 双线性模型、VGG13 双线性模型和 VGG16 双线性模型精度都在 0.7 以上，其中，VGG11 双线性模型精度约为 0.74，VGG13 双线性模型精度约为 0.75，VGG16 双线性模型精度约为 0.77，模型的训练都经过了一些参数调优，争取在当前训练方法中实现最佳精度。

当然，一般来说双线性模型可以使用更大的分辨率训练来提升性能，当我们将分辨率提升到 448 时，VGG16 双线性模型精度可以提升到 79%。

以上实验表明，双线性模型相比于基准模型，的确有明显的性能优势，是一个更合适的细粒度图像分类模型，而且其参数量相比于基准模型更少。当然，这里的双线性模型在基准架构、训练图像分辨率和训练方式等方面还存在较大的改进空间，读者可以自行改进。

4.7.4　项目总结

本章完成了一个细粒度的图像分类任务，这在图像分类任务中是属于难度较高的任务，因为网络需要学习更精细的局部特征。基于 VGG 的双线性模型最终取得了接近 80% 的测试集精度，但是距离人类专家的水平还有很大的差距，因此还有较大的研究空间。

4.8　总结

不论是在基于图像处理算法和经典机器学习模型的图像识别问题中，还是在基于深度卷积神经网络模型的图像识别问题中，图像分类都是最基础的内容，其中，最重要的步骤是将高维图像压缩为线性可分的低维特征。

当然，图像分类本身也存在一些细分领域，本章介绍了基本的多类别图像分类，以及更

复杂的细粒度图像分类、多标签图像分类、无监督与半监督图像分类、零样本图像分类，比较完备地讲解了其中的核心算法。最后基于两个实践案例，为初学图像识别的读者理解并掌握图像分类问题提供了非常合适的练习素材。

参考文献

[1] Wah C，Branson S，Welinder P，et al. The caltech-ucsd birds-200-2011 dataset[J]. Technical Report CNS-TR-2011-001, California Institute of Technology，2011.

[2] Khosla A，Jayadevaprakash N，Yao B，et al. Novel dataset for fine-grained image categorization:Stanford dogs[C]//Proc. CVPR workshop on fine-grained visual categorization（FGVC）. Citeseer，2011：1-2.

[3] Krause J，Stark M，Jia D，et al. 3D Object Representations for Fine-Grained Categorization[C]// IEEE International Conference on Computer Vision Workshops. IEEE Computer Society，2013：554-561.

[4] LeCun Y，Bottou L，Bengio Y，et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE，1998，86（11）：2278-2324.

[5] Krizhevsky A，Sutskever I，Hinton G E. Imagenet classification with deep convolutional neural networks[J]. Communications of the ACM，2017，60（6）：84-90.

[6] Zeiler M D，Fergus R. Visualizing and understanding convolutional networks[C]//European Conference on Computer Vision，2014：818-833.

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

[8] Szegedy C，Liu W，Jia Y，et al. Going deeper with convolutions[C]//IEEE conference on computer vision and pattern recognition. IEEE，2015：1-9.

[9] He K，Zhang X，Ren S，et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition，2016：770-778.

[10] Xie S，Girshick R，Dollár P，et al. Aggregated residual transformations for deep neural networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition，2017：1492-1500.

[11] Huang G，Liu Z，Van Der Maaten L，et al. Densely connected convolutional networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition，2017：4700-4708.

[12] Hu J，Shen L，Sun G. Squeeze-and-excitation networks[C]//Proceedings of the IEEE conference on computer vision and pattern recognition，2018：7132-7141.

[13] Wang J，Yang Y，Mao J，et al. CNN-RNN：A unified framework for multi-label image classification[C]//Proceedings of the IEEE conference on computer vision and pattern recognition，2016：2285-2294.

[14] Zhang M L，Zhou Z H. A review on multi-label learning algorithms[J]. IEEE transactions on

knowledge and data engineering，2013，26（8）：1819-1837.

[15] Zhang N，Donahue J，Girshick R，et al. Part-based R-CNNs for fine-grained category detection[C]//Computer Vision–ECCV 2014:13th European Conference，Zurich，Switzerland，September 6-12，2014，Proceedings，Part I 13. Springer International Publishing，2014：834-849.

[16] Zheng H，Fu J，Mei T，et al. Learning multi-attention convolutional neural network for fine-grained image recognition[C]//Proceedings of the IEEE international conference on computer vision，2017：5209-5217.

[17] Lin T Y，RoyChowdhury A，Maji S. Bilinear CNN models for fine-grained visual recognition[C]//Proceedings of the IEEE international conference on computer vision，2015：1449-1457.

[18] Gao Y，Beijbom O，Zhang N，et al. Compact bilinear pooling[C]//Proceedings of the IEEE conference on computer vision and pattern recognition，2016：317-326.

[19] GRANDVALET Y. Semi-supervised learning by entropy minimization[C]//Advances in Neural Information Processing Systems，2005，17：529-536.

[20] Berthelot D，Carlini N，Goodfellow I，et al. Mixmatch：A holistic approach to semi-supervised learning[C]//Advances in Neural Information Processing Systems，2019：5049-5059.

[21] Caron M，Bojanowski P，Joulin A，et al. Deep clustering for unsupervised learning of visual features[C]//Proceedings of the European conference on computer vision（ECCV），2018：132-149.

[22] Ji X，Henriques J F，Vedaldi A. Invariant information clustering for unsupervised image classification and segmentation[C]//Proceedings of the IEEE/CVF International Conference on Computer Vision，2019：9865-9874.

[23] Schmarje L，Santarossa M，Schröder S M，et al. A survey on semi-，self-and unsupervised learning for image classification[J]. IEEE Access，2021，9：82146-82168.

[24] 冀中，汪浩然，于云龙，等 . 零样本图像分类综述：十年进展 [J]. 中国科学：信息科学，2019，49（10）：1299-1320.

[25] Liu Z，Luo P，Wang X，et al. Large-scale celebfaces attributes（celeba）dataset[J]. Retrieved August，2018，15（2018）：11.