第3章

深度学习基础

知识目标	• 了解深度学习的基本概念以及与机器学习之间的区别 • 了解卷积神经网络的基本架构和各部分的功能 • 了解生成对抗网络的基本原理及网络架构 • 了解注意力机制的基本原理和在计算机视觉任务中的作用 • 了解视觉注意力网络的基本构成与网络架构 • 了解多层感知机的基本构成与网络架构 • 了解扩散模型的基本原理以及不同分类 • 了解图神经网络的基本原理与不同分类 • 了解神经架构搜索的基本流程与各流程所使用的方法
技能目标	• 掌握卷积神经网络架构，包括确定卷积层、池化层、全连接层等的基于 Python 代码实现 • 掌握空间注意力、通道注意力、像素注意力等不同类型注意力的适用场景与基于 Python 代码实现
素质目标	• 通过案例分析、实验操作和项目开发等实践环节，提高学生的实际操作能力和解决问题的能力 • 鼓励学生在掌握基础知识的基础上，勇于探索新的技术、方法和应用，培养创新精神和创新能力 • 通过本章的学习，掌握深度学习的基本原理、方法和应用，为后续的学习和实践奠定坚实的基础
思政目标	• 通过深度学习在各个领域的应用案例，引导学生认识到科技创新对国家发展的重要性，激发他们的爱国情怀和为国家发展贡献力量的决心，激发读者的学习兴趣，养成良好的学习习惯，勤于反思，培养读者的信息意识 • 鼓励学生关注社会热点问题，运用深度学习技术进行分析和解决，传播正能量，推动社会进步

本章主要介绍深度学习的相关概念及目前主流的 7 种深度学习框架。通过本章的学习，读者可以加深对深度学习的认识与理解，熟悉本章讲解的深度学习框架并积极运用到后面章节的计算机视觉任务中，为以后深入学习与应用深度学习打下坚实的基础。

3.1 深度学习简介

深度学习是机器学习领域中的一个重要分支，其核心在于构建和训练深度神经网络模型。深度神经网络是一种模拟人脑神经网络结构的计算模型，通过组合低层特征形成更抽象的高

层表示属性类别或特征，以发现数据的分布式特征表示。

深度学习的研究最早可追溯到20世纪40年代提出的神经网络。然而，直到2006年，深度学习的概念才被正式提出，并在随后几年中随着计算能力的提升和大数据的涌现而得到快速发展。如今，深度学习已经在许多领域取得了显著成果，包括计算机视觉、语音识别、自然语言处理、数据挖掘等。与传统的机器学习算法相比，深度学习具有更强的特征学习能力。它可以自动地从原始数据中提取有用的特征，而无须进行烦琐的特征工程。这使得深度学习在处理复杂和高维数据时具有更大的优势。同时，深度学习还具有强大的泛化能力。通过在大规模数据集上进行训练，深度学习模型可以学习到数据的内在规律和模式，从而对新数据进行准确的预测和分类，使深度学习在解决实际问题中有广泛的应用前景。

目前，深度学习按照学习过程的不同可大致分为5类，分别为有监督学习、弱监督学习、无监督学习、自监督学习以及强化学习。这些分类方式主要基于模型学习过程中是否使用了标签数据，以及标签数据的来源和使用方式。下面对上述五类学习方式进行具体阐述。

1. 有监督学习

有监督学习(Supervised Learning)是机器学习中的一种重要方法，它利用一组带有已知标签或结果的样本来训练模型，使其能够学会如何根据输入数据预测正确的输出或结果。有监督学习的目标是基于已知标签的数据来训练模型，以便对未知标签的数据进行准确预测。在有监督学习中，模型利用一组带有已知标签的样本数据进行训练。标签通常是人工标注的，用于指示每个样本的正确输出或类别。通过比较模型的预测输出与真实标签之间的差异，并使用如梯度下降等优化算法来最小化这种差异，模型逐渐学会从输入数据中提取有用特征，并预测新样本的标签。

然而，有监督学习也存在一些挑战和限制。首先，它需要大量的带有标签的数据进行训练，而在某些情况下，获取这样的数据可能非常困难或成本高昂。其次，如果训练数据中的标签存在噪声或错误，可能会对模型的性能产生负面影响。此外，有监督学习通常假设训练数据和测试数据具有相同的分布，如果这种假设不成立，可能会导致模型的泛化能力下降。

2. 半/弱监督学习

弱监督学习(Weakly-Supervised Learning)的目标是利用不完全、不确切或不精确的标签信息，通过构建预测模型来提升学习性能。弱监督学习通常可以分为以下三类。

(1) 不完全监督：在这种情境下，只有训练集的一个(通常很小的)子集是有标签的，其他数据则没有标签。这常见于各类任务，特别是在图像分类任务中，考虑到标记的人工成本，只有小部分图像能够被标注。

(2) 不确切监督：在该情况下，图像或数据只有粗粒度的标签，而不是具体的、精确的分类。

(3) 不准确的监督：指的是模型给出的标签不总是真值。这种情况可能由于图片标注者不小心或比较疲倦，或者某些图片本身就是难以分类等原因造成的。

弱监督学习的应用非常广泛，例如，在图像分类任务中，许多图像都没有明确的标签，但通过调整网络结构和学习算法，弱监督学习仍然可以实现较高的准确率。此外，弱监督学习还可以与一些生成模型相结合，通过建立概率模型来描述输入和输出之间的关系，以提高模型的性能和泛化能力。

3. 无监督学习

与有监督学习不同，无监督学习(Unsupervised Learning)使用的数据没有明确的标签。在无监督学习中，模型利用无标签的数据进行学习，通过寻找数据内在的结构和规律来发现数

据中的模式或特征。无监督学习更像是一种让机器“自学”的过程。模型通过对数据进行内在结构和关系的探索，自动地找出数据的隐藏模式或特征。

无监督学习具有一些显著的特点。首先，由于没有人为的标签或目标，数据本身成为唯一的信息来源，包含学习模型所需要的所有信息。其次，无监督学习具有一定的自我学习能力，能够根据数据的规律和内在特征进行精准建模，无须外部指导和参数限制。此外，无监督学习强调数据驱动，通过吸取数据相关信息和发现隐藏在数据内部的知识和规律，推导出规则和信息，从而提高算法和模型的效果。总的来说，无监督学习是一种灵活的机器学习技术，它能够在无标签数据的情况下发现数据中的模式和结构，为数据分析和决策提供有力支持。

4. 自监督学习

自监督学习（Self-Supervised Learning）是一种介于有监督学习和无监督学习之间的方法，其核心思想在于利用输入数据本身的上下文信息或结构特性作为监督信号，从而进行特征学习和模型训练。与传统的无监督学习相比，自监督学习更注重于通过设计辅助任务来挖掘数据自身的表征特征，以此提升模型的特征提取能力。

自监督学习的最大特点在于不依赖人工标注的数据标签，而是直接从原始数据中自动学习有区分度的特征表示。这使得自监督学习能够在大量无标签数据上有效地进行训练，从而避免了监督学习中需要大量人工标注数据的局限。在自监督学习的过程中，核心在于合理构造有利于模型学习的任务。一旦模型通过这些自监督任务进行了训练，它就可以用于其他相关任务，有时甚至可以超越传统的有监督学习方法。

5. 强化学习

强化学习（Reinforcement Learning）是机器学习的一个范式和方法论，专注于描述和解决智能模型在与环境的交互过程中通过学习策略以达成回报最大化或实现特定目标的问题。强化学习具有以下显著特点。

（1）延迟奖励：与有监督学习和无监督学习不同，强化学习不是立即给予奖励或惩罚，而是根据整个序列的累积奖励来决定学习效果。这种延迟奖励机制使得强化学习更加适应于具有长期依赖性的复杂任务。

（2）序列决策：强化学习通常需要解决的是序列决策问题，即在面对一系列决策时，如何选择每个决策以达到最终的目标。这需要考虑到未来的影响和结果，而不仅仅是单个决策的奖励或惩罚。

（3）与环境交互：强化学习通过与环境交互来学习，即通过尝试不同的行动来观察结果并更新知识。这种交互性使强化学习更加灵活和适应各种不同的环境。

强化学习在基因组学、游戏 AI 以及自动驾驶等多个领域都有广泛的应用。它可以让计算机系统在没有人类指导的情况下自己学习如何学习知识，并取得了超越人类的成绩。然而，强化学习也面临着一些挑战，如数据稀疏性、过拟合等问题。为了解决这些问题，研究人员提出了多种改进算法和技术，如分布式强化学习、模仿学习等，以提高强化学习的稳定性和收敛性。

总之，深度学习是一种强大的机器学习方法，它通过构建深度神经网络模型来自动地学习数据的特征和规律，从而实现对数据的准确处理和分析。随着技术的不断进步和应用场景的不断拓展，深度学习将在未来发挥更加重要的作用。

3.2 卷积神经网络

1980 年，计算机科学家 Kunihiko Fukushima 在论文“Neocognitron：A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”[1]

中提出了一个包含卷积层、池化层的神经网络结构，这也是卷积神经网络的雏形。1989 年，Yang LeCun 等提出了 LeNet，这一举措极大地推动了深度学习的发展，Yang LeCun 也被称为“卷积网络之父”。卷积神经网络(Convolutional Neural Network，CNN)是一种最广泛使用的人工神经网络结构，能够自动提取输入数据的特征，从而实现对输入数据的高效分类和识别，是深度学习中的一种重要网络结构，它主要包括卷积层(Convolutional Layer)、池化层(Pooling Layer)、全连接层(Fully Connected Layer)和激活函数(Activation Function)，由于其网络结构能够从自然图像中捕捉到图像的细节信息，在处理不同尺寸的图像上也表现比较出色，因此在图像分类、目标检测等计算机视觉研究领域被广泛应用。CNN 的基本特点如下。

1. 局部连接与权值共享

CNN 使用局部连接和权值共享，将每个神经元仅与输入的一小块区域连接，且在它们之间共享相同的权值。这种结构有利于减少网络参数数量，提高网络的稳定性和泛化能力。

2. 多层次的特征提取

卷积神经网络通过堆叠多个卷积层和池化层，逐层学习和提取输入数据的特征表示，从而实现高级别的抽象和分类任务。

3. 平移不变性和位置信息的保留

由于卷积层和池化层具有平移不变性，因此卷积神经网络能够处理不同位置的输入数据，同时还能保留输入数据的位置信息，在图像领域有很好的应用效果。

4. 预训练和微调

CNN 可以用于预训练(Pre-training)和微调(Fine Tune)，预训练可以提高网络的鲁棒性和泛化能力，微调操作可以在已有的模型上进行重新训练，进而使模型更好地适应新的数据集。

5. 数据增强的可用性

卷积神经网络(CNN)通过数据增强(Data Enhancement)，如旋转(Rotating)、翻转(Flipping)、裁剪(Cropping)等来扩展数据集的大小，从而提高网络模型的泛化能力。

3.2.1 卷积算子

卷积层(Convolutional Layer)是卷积神经网络的核心组件，其单位是卷积核，主要负责在预处理后的原始输入图像上提取有用的特征。早期图像处理中，就有学者发现可以选择各种类型的卷积核来对输入图像进行不同特征的处理，如纹理、边缘、形状等特征。由于图像是由一个又一个的像素组成的，所以这些特征也是由图像中的每个像素通过组合或其他方式所体现。本节将讨论卷积操作的计算、卷积操作的特点以及部分常用的卷积操作。

1. 卷积操作的计算

在卷积操作中，通常涉及泛函分析的使用。泛函分析是 20 世纪 30 年代形成的数学分科，是从变分问题、积分方程和理论物理的研究中发展起来的。它综合运用函数论、几何学、现代数学的观点来研究无限维向量空间上的泛函、算子和极限理论。它可以看作无限维向量空间的解析几何及数学分析。泛函分析中的卷积 $\boldsymbol{f} * \boldsymbol{g}$ 被定义为

$$(\boldsymbol{f} * \boldsymbol{g})(t) = \int_{-\infty}^{\infty} \boldsymbol{f}(\tau)\boldsymbol{g}(t-\tau)\mathrm{d}\tau \tag{3-1}$$

其中，$\boldsymbol{f}$ 与 $\boldsymbol{g}$ 分别表示一个从向量下标到向量元素值的映射，令 $\boldsymbol{f}$ 表示输入向量，即原始图像，$\boldsymbol{g}$ 表示的向量称为卷积核(convolution kernel)或滤波器。卷积核施加于输入向量上的操作类似于一个权值向量在输入向量上滑动(滑动的方向遵循从左往右、从上至下)，每滑动一次

就进行一次对应元素的相乘求和操作，每次滑动的距离称为步长（stride），步长代表了提取特征的精度，通常设置为 1。也可以用大于或等于 2 的步长对图像进行下采样，替代池化操作（池化操作在后文有详细讲解）。如图 3-1 所示为当输入同一图像时，设置不同的卷积核、步长提取图像特征得到的结果。

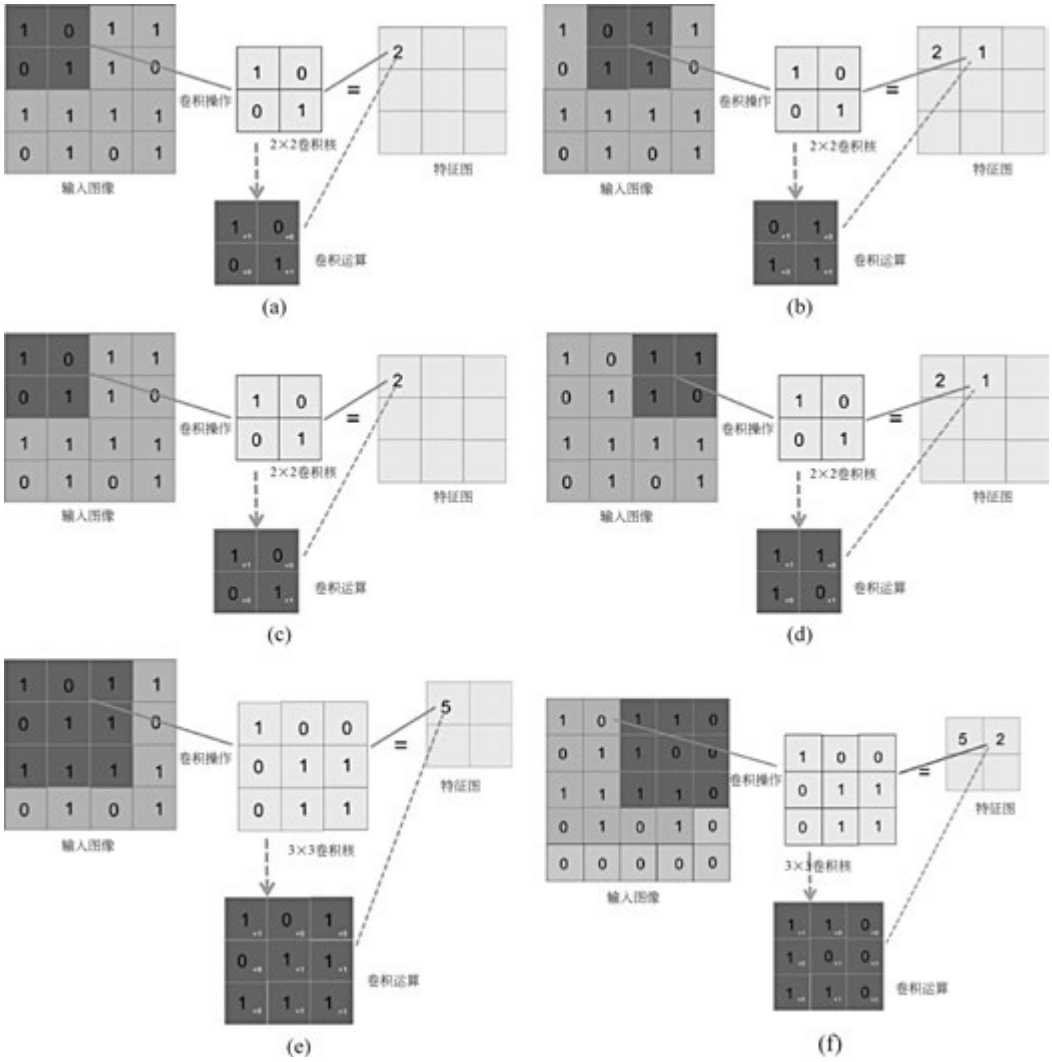


图 3-1　卷积操作

如图 3-1(f)所示，可以发现，当输入图像尺寸过小而卷积核过大时，会对输入图像进行填充（padding）操作，这样可以有效避免图像中的信息丢失或者过拟合等问题发生。对于较小的图像使用填充操作，对于 CNN 来说是很有效的，因为增加的像素数量相对较小，不会对模型的计算效力产生过多的影响。填充的方式除了使用上述“零填充”进行边界填充处理外，还可以使用均值来进行填充。

此外，为了避免卷积后的图像变小，也会使用填充操作来弥补该损失，如输入尺寸为 5×5，卷积核的大小为 3×3，若不对图片进行填充，步长为 1 时，卷积核沿着图片滑动后只能生成一个 3×3 的图片出来，卷积后的图片会越变越小，且输入图片边缘像素只被计算一次，而中间像素会被卷积计算多次，意味着图像角落信息被丢失，对图片进行边界填充处理后，图像大小变为 7×7，此时可控制卷积后特征图的大小，并加强了对图像周围像素信息的利用。

2. 卷积操作的特点

卷积操作的特点通常表现为权值共享和局部连接，如图 3-1(a)与图 3-1(b)所示。权值共享在卷积神经网络中主要是指在空间上的权值共享，即使用同一组参数(filter)去遍历整个图像，由于其在减少网络参数的同时还可以保留良好的网络质量的优质性能，被广泛应用于提取整个图像中具有共性的特征信息，如颜色特征、纹理特征等。

而局部连接是指在卷积层中，每个输出神经元在空间上只和邻域的一小部分输入神经元进行相连，连接的空间大小叫作神经元的感受野(Receptive Field)，它表示滤波器的空间尺寸。局部连接保留了图像局部结构，并减少了网络的权值个数。例如，对于一个 10×10 的输入图像而言，如果下一个隐藏层的神经元数目为 10^3 个，采用全连接就有 $10\times10\times10^3$ 个权值参数，这样的计算量看起来似乎还可以接受。但对于一个 $10\,000\times10\,000$ 的输入图像而言，如果下一个隐藏层的神经元数目为 10^8 个，则采用全连接就有 $10\,000\times10\,000\times10^8$ 个权值参数，这样的参数计算量在训练过程中是很难进行的。而采用局部连接，隐藏层的每个神经元仅与图像中 10×10 的局部图像相连接，那么此时的权值参数数量为 $10\times10\times10^8=10^{10}$，直接减少 6 个数量级，大大减少了网络训练时所用的参数量，加快了模型训练速度。

3. 常用的卷积

表 3-1 中总结了部分常用的卷积操作及其特点，下文详细介绍转置卷积和深度可分离卷积。

表 3-1 常用卷积

名　　称	特　　点
空洞卷积 (Dilated Convolution)	也称为扩张卷积，与普通卷积操作不同的是，它通过在卷积核中引入间隔(空洞)来扩大卷积核的感受野，同时也能在不增加计算负担的情况下，学习更加丰富的空间信息
转置卷积 (Transposed Convolution)	也称为反卷积或上采样卷积，是一种在卷积神经网络中用于实现特征图空间尺寸增大的卷积操作。通常用于将低分辨率的图像上采样到高分辨率图像或是将网络中间层的特征图上采样到与原始输入图像相同的尺寸，便于后续像素级任务的进行
组卷积 (Group Convolution)	是指将输入输出通道分组来执行卷积运算，实现分组卷积，其优势是可以更好地进行结构化学习，克服过拟合，减少参数，产生类似正则化效果，从而提升整个网络的精度与效率
深度可分离卷积 (Depthwise Separable Convolution)	深度可分离卷积可以大幅减少卷积神经网络的参数，因此它在模型层数不变的前提下可以使模型总参数量大幅下降而精度只略有损失

1) 转置卷积

在神经网络中，经常需要采用上采样操作来提高低分辨率图片的分辨率，转置卷积可以作为一种通过卷积学习参数从而获得最优上采样的方法。图 3-2 是一个 4×4 的输入矩阵，用 3×3 的卷积核进行步长为 1 的卷积操作，最终输出一个 2×2 的矩阵。

现在将 3×3 的卷积核重排为 4×16 的形式，如图 3-3 所示(以普通卷积的前两行为例)，同时将 4×4 的输入矩阵展开为 16×1 的列向量。通过矩阵乘法得到 4×1 的列向量，可以看出这个列项量正是由图 3-2 中的 2×2 的卷积输出矩阵展开得到的，即将卷积操作写成矩阵乘法运算。

通过上述操作，就可以将 16(4×4 的矩阵)个值映射为 4(2×2 的矩阵)个值。虽然转置卷积能够通过学习参数进行最优的上采样，但在实际应用中，研究人员往往更加倾向于使用线性

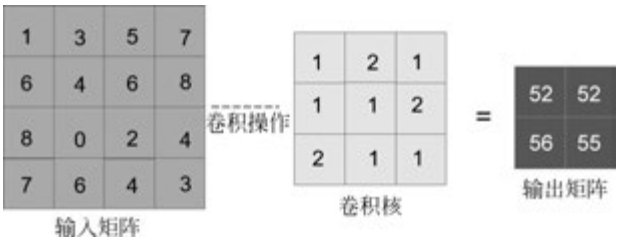


图 3-2　普通卷积操作

图 3-3　转置卷积操作

插值的方式做上采样，效率会更高。

2）深度可分离卷积

深度可分离卷积涉及空间维度和深度维度的信息处理，主要分为两个过程：逐通道卷积和逐点卷积。其中，逐通道卷积的卷积核与通道是一一对应的，所以输出的特征图片的深度和输入的深度完全一样，如图 3-4(a)所示。

逐点卷积的运算与常规卷积的运算相似，它的卷积核的尺寸为 $1\times1\times M$，M 为上一层的深度，它是将上一步的特征图在深度方向上进行加权相加(由图 3-4 的深度可分离卷积的输出作为逐点卷积的输入)，生成新的特征图。在该操作过程中，特征图输出的个数与卷积核的个数是一一对应的，如图 3-4(b)所示。

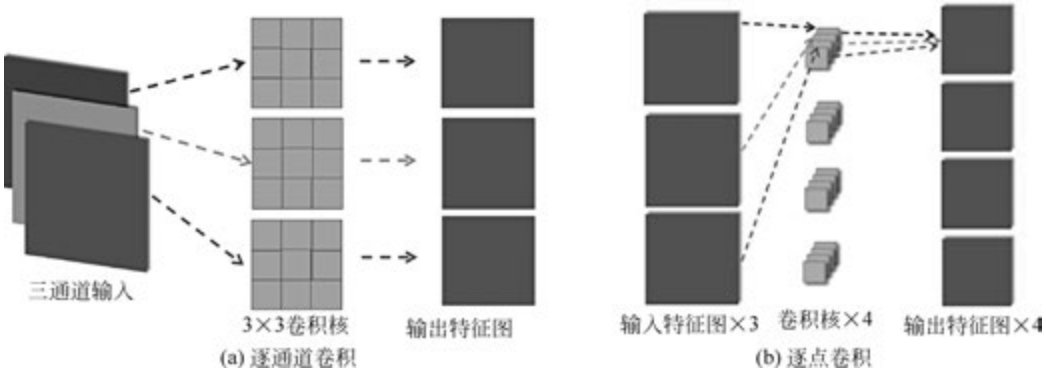


图 3-4　深度可分离卷积

深度可分离卷积的优势在于需要提取的属性越多，就能节省越多的参数，减少计算量。最早出现在 MobileNet 中，也是由于轻量化网络的特征提取部分，使嵌入式部署的神经网络推理速度更快速。

除上述转置卷积和深度可分离卷积外，在计算机视觉的相关应用领域中还会用到扩张卷积（用来增大感受野）、分组卷积（解决显存不足，将输入特征图进行分组处理，减少训练参数，避免过拟合现象的出现）、可变形卷积等。

3.2.2　池化操作

一般地，一个卷积层后会有一个池化层，后者几乎汇总了接收到领域所输出特征图的激活情况。所谓池化操作，也称为下采样操作，是对特征图进行特征压缩，降低输入图片的尺寸，减少数据量，同时保留重要的特征。池化操作的作用使得检测到的特征进一步增强同时加速运算。其输入是卷积层的输出，输出为下采样操作后的特征图。池化操作可以分为最大池化（Max Pooling）、平均池化（Average Pooling）、L2 池化（L2 Pooling）、随机池化（Stochastic Pooling）等不同种类，其操作过程中也会用到 kernel 和步长。其中，最大池化是指在每次移动池化窗口所框选出的区域中找到最大值将其输出（如图 3-5(a)的最大池化操作）；而平均池化操作（如图 3-5(b)所示）则是将池化窗口内的所有参数求平均值作为输出。

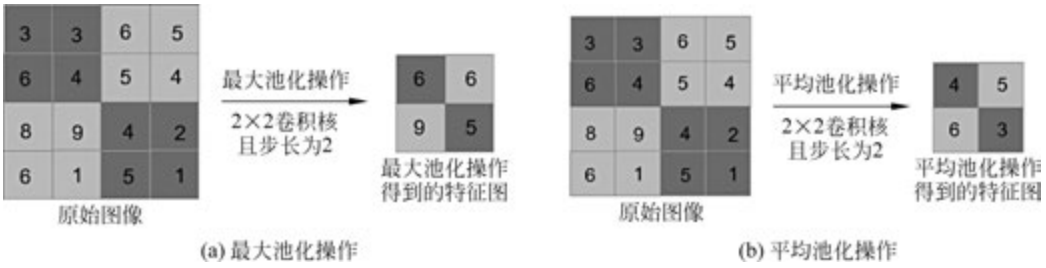


图 3-5　最大池化操作与平均池化操作

表 3-2 中介绍了部分池化操作的类型及特点，在具体的应用场景中要选择合适的池化操作来适应模型的训练。下面将详细介绍最大池化与平均池化操作的内容。

表 3-2　部分池化操作类型及特点

池化操作名称	特　　点
最大池化（Max Pooling）	即对领域内特征点求最大值，其优点是能够保留图像的细节特征
平均池化（Average Pooling）	对领域内特征点求平均值，在保留背景信息方面表现优异，但容易使图像变模糊，对图像的全局信息保留不完整
L2 池化（L2 Pooling）	即对每个小区域内的数值求 L2 范数（即向量长度），然后将 L2 范数作为输出值，能够有效地抑制噪声和过拟合，并提高模型的泛化能力
随机池化（Stochastic Pooling）	指对像素点按照数值大小赋予概率，数值越大则概率越大。注意，这里与最大池化操作有所不同，最大池化操作永远只选择最大的值，而随机池化是由概率确定的

如图 3-5 中最大池化的值通过下列计算得到。

```
max(3,3,6,4) = 6; max(6,5,5,4) = 6; max(8,9,6,1) = 9; max(4,2,5,1) = 5
```

平均池化的值只需要将取最大值的计算替换成平均值的计算即可。

PyTorch 中的最大池化：

```
torch.nn.MaxPool2d(kernel_size, stride = None, padding = 0, dilation = 1, return_indices = False,
ceil_mode = False)
```

其中，kernel_size 用来定义最大池化的窗口大小；stride 表示窗口移动步长，默认为 kernel_size；padding 是指要输入的每一条边添加 0 进行填充；dilation 是控制窗口中元素步幅；ceil_mode 表示向上取整模式，默认值为 False，如果为 True，会返回输出最大值的序号，对于上采样操作有很大的帮助，如果为 False，则表示当剩余数据不足 kernel_size 大小时，直接舍弃。

在具体使用过程中，卷积操作与池化操作会对输入的图像多次进行。随着网络层数的增加，特征图的个数也在不断增加，同时图像的空间分辨率不断减小。在多次卷积-池化层的最后，特征图会被输送到全连接网络中，最后是输出层。输出单元依赖于具体的任务，若是回归问题，则输出单元的激活函数是线性的。如果是二分类问题，则输出单元是 sigmoid；对于多分类问题，则输出层是 softmax 单元。

卷积操作与池化操作之间有哪些异同呢？通常有以下 4 点。①卷积对应有卷积核，池化对应有池化核。卷积核内有参数，而池化层只是一个框架，里面没有参数。②卷积里面一般用 Padding same，而池化层通常用 Padding valid。③在通道方面，使用池化不会造成数据矩阵深度的改变，只会在高度和宽度上降低，达到降维的目的，池化前是几个通道，池化后仍为几个通道。但在卷积中，上一层的 feature map 的个数（即图层个数）与下一层的卷积核通道数一致。④两者在使用过程中均需要定义其大小（size）、步长（stride）、padding 类型。在使用这两种操作的过程中，要注意两者之间的区别和联系。

3.2.3 归一化和正则化

CNN 中的归一化操作和正则化操作在模型训练过程中能够提高模型的性能和稳定性。

1. 归一化

归一化（Normalization）是数据预处理中常用的方法之一，用于将数据特征缩放到一个统一的范围内，通常是[0,1]或[−1,1]。该操作能够：①防止数值过小的特征被淹没；②保证数据的有效性，稳定处理的数据符合标准正态分布。在 CNN 中，输入数据的每个通道（如 RGB 三通道）可以独立进行归一化处理，这使不同尺度的特征具有可比性，同时也加快了模型的收敛速度。

常见的归一化的方法有最小-最大归一化（Min-Max Normalization），用于对原始数据进行线性变换，其语法格式为

```
x_normalized = (x - min_value)/(max_value - min_value)
```

其中，x 是原始数据，min_value 和 max_value 是数据的最小值和最大值。此外，还有批归一化（Batch Normalization，BN）、层归一化（Layer Normalization，LN）等，具体内容详见表 3-3。

表 3-3 归一化操作类型

归一化操作名称	特　点
批归一化	批归一化（BN）通过对神经网络的每一层的输入数据进行规范化处理，使其符合标准的正态分布 $N(0,1)$，有效避免前一层参数更新所导致的当前层数据分布发生变化的现象，进而保证网络的训练有效进行

续表

归一化操作名称	特　点
层归一化	层归一化(LN)对每个样本的所有特征都进行均值和方差计算,并利用得到的值对该样本的所有特征进行归一化处理。由于不同样本之间像素存在较大的相关性,LN无法很好地处理这些信息,故LN在CNN中处理图像数据的效果不如BN
分组归一化	分组归一化(Group Normalization,GN)将输入数据划分为多个组,分别对每个组进行归一化操作。GN一个组内的通道之间共享归一化参数,考虑了通道之间的关系,同时该操作步骤(batch)大小和网络深度的影响,具有较好的稳定性。但GN对组的数量较为敏感,若分组不恰当可能会导致模型性能下降
实例归一化	实例归一化(Instance Normalization,IN)常用于图像生成任务(如风格迁移),其基本思想是将图像的每个通道单独进行归一化处理,能够保留每个通道的特异性

2. 正则化

正则化(Regularization)[2]是一种避免模型过拟合、欠拟合或减少泛化误差的方法。所谓过拟合(over-fitting)是指当模型过度学习训练样本中的细节和噪声,把训练样本自身的一些特点当作所有潜在的样本都会具有的一般性质,导致训练出的模型在测试数据集上表现不佳,模型自身的泛化性能力下降。通常,导致模型出现过拟合的原因有以下几点。

(1) 数据量太小。即训练过程中使用一个较大的数据集来进行训练,该数据较好地与某一函数相吻合,但当从数据中摘出部分数据出来进行训练时,此时该部分数据很有可能变为一个线性函数,而该函数在测试集上的表现自然也会变得不好。

(2) 训练集与测试集分布不一致。有时在该数据集上训练出来的数据可能只能适应训练集的数据分布形式,当将其用在分布不一的数据上,效果可能不如训练阶段。

(3) 模型复杂度太大。若选择一个复杂度较高的算法来对一个相较简单的数据继续训练,该模型可能就不那么适用。

(4) 数据质量很差。当训练的数据存在过多噪声时,模型在训练过程中也会将这部分噪声学习到,从而减少了具有一般性的规律。

(5) 过度训练。这一点与第4点是相联系的,只要训练时间足够长,模型就会把一些噪声隐含的规律学习到,此时会降低模型的性能。

正则化作为一种回归的形式,可以将系数估计(coefficient estimate)朝零的方向进行约束、调整或缩小。即正则化可以在学习过程中降低模型复杂度和不稳定程度,从而避免过拟合现象的发生。在CNN中,常用的正则化方式如表3-4所示。

表3-4　正则化操作类型

正则化操作名称	类别及特点	
参数正则化	Dropout正则化	在CNN中约束模型参数,减少模型训练过程中出现过拟合现象,提高模型的泛化能力
	L1正则化(L1 norm)	
	L2正则化(L2 norm)	
数据正则化	数据增广	通过调整数据集的分布和尺度,改善模型的训练效率
	提前停止	
标签正则化	标签平滑	通过对标签信息进行优化,提升模型的泛化能力

在CNN中引入正则化和归一化操作,可以有效提高模型的性能和泛化能力。在实际任务中,通常会将两者结合使用,以此构建高效的卷积神经网络模型来适应不同的计算机视觉任务。

3.2.4 激活函数

激活函数(Activate Function)又称为非线性映射层，它对于人工神经网络模型去学习、理解非常复杂和非线性的函数来说具有十分重要的作用。激活函数将非线性特性引入网络中，增加整个网络的表达能力，使网络更好地学习和理解复杂的输入输出关系，避免了线性层操作的堆叠而导致的线性映射的干扰。在计算机视觉领域，常用的激活函数有 Sigmoid 型函数和 ReLU 型函数。其他激活函数如表 3-5 所示。

表 3-5 其他激活函数

激活函数名称	特　　点
Tanh 函数	Tanh 函数的输出范围为[−1,1]，常用于输出层，适用于二分类问题。该函数存在梯度消失问题
Leaky ReLU 函数	该函数是 ReLU 的改进版，其输入小于 0 的部分，输出不为−∞，而是一个较小的正值，这样可以避免在训练初期由于梯度为 0 而导致无法学习到有效信息的问题
PReLU 函数	PReLU 函数与 Leaky ReLU 函数一样，用来解决 ReLU 带来的神经元坏死的问题。与 Leaky ReLU 函数不同的是，PReLU 函数负半轴的斜率参数是通过学习得到的，而非手动设置的恒定值
Softmax 函数	Softmax 函数常用于多分类问题的输出层，它将一组输入映射到一个概率分布。其输出范围为[0,1]，且总和为 1，可以看作以概率分布。其函数映射值越大，则真实类别可能性越大
Swish 函数	Swish 函数通常被用来引入非线性因素，有助于改善模型的学习能力和性能。一般地，Swish 函数的取值范围受到 Sigmoid 函数的限制

1. Sigmoid 函数

Sigmoid 函数又称为 Logistic 函数，常用于输出层，其表达式为

$$\sigma(x)=\frac{1}{1+\exp(-x)} \tag{3-2}$$

函数形状如图 3-6(a)所示，经过 Sigmoid 函数的作用后，输出的值域被压缩到[0,1]，而 0 和 1 分别对应输出层的“休眠状态”和“激活状态”，观察 Sigmoid 函数的两端，对于大于 5(或小于−5)的值无论多大(或多小)都会压制 1(或 0)，这样会促使梯度饱和，对照图 3-6(b)——Sigmoid 函数的梯度图，大于 5(或小于−5)部分的梯度接近于 0(从数学的角度思考，即该处的导数为 0)，这会导致在误差反向传播的过程中，该区域的误差很难传递到前层，进而使整个网络无法训练。此外，在网络训练的过程中要避免参数初始化过大，原因是当初始化参数过大时，将其输出值域直接带入这个区域，将会直接引发梯度饱和现象，造成整个网络的训练受阻。

2. ReLU 函数

2010 年，Nair 和 Hinton 为避免梯度饱和现象的发生，将 ReLU(Rectified Linear Unit，修正线性单元)引入神经网络中，是目前 CNN 中最常用的激活函数之一。其公式定义为

$$\mathrm{ReLU}(x)=\max\{0,x\}=\begin{cases}0, x<0\\ x, x\geqslant 0\end{cases} \tag{3-3}$$

根据其公式定义可见，ReLU 函数的梯度在 $x\geqslant 0$ 时为 1，反之为 0，如图 3-7 所示。在 $x\geqslant 0$ 的部分就可以消除 Sigmoid 函数的梯度饱和效应。此外，Sigmoid 函数还助于随机梯度下降方法的收敛，收敛速度快了 6 倍左右。

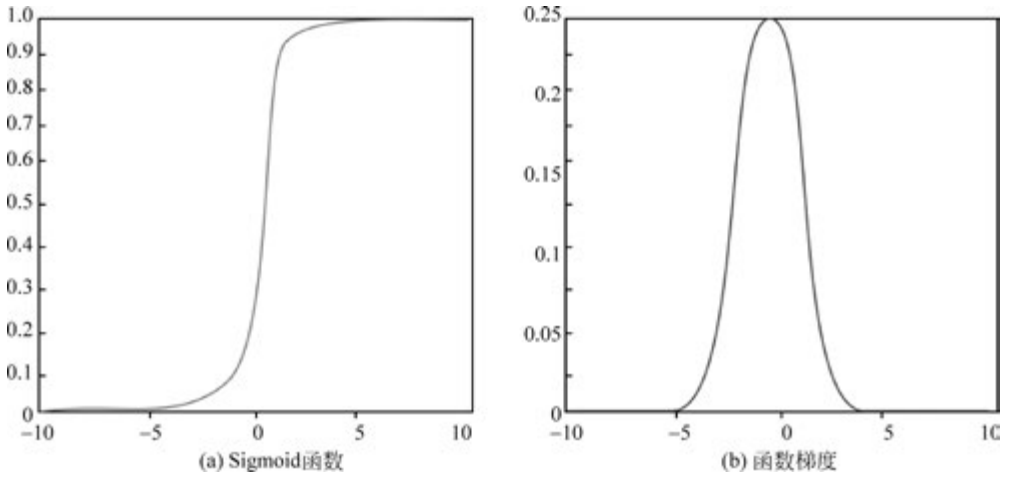


图 3-6 Sigmoid 函数及其函数梯度

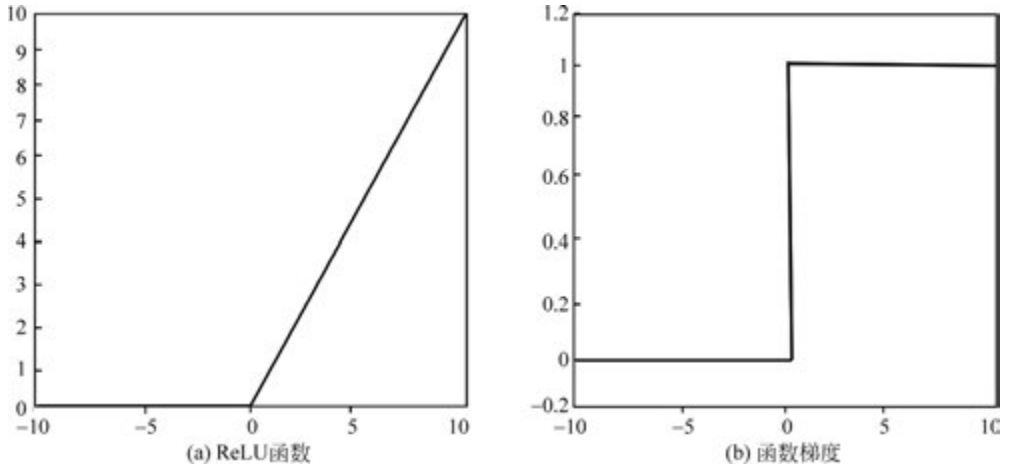


图 3-7 ReLU 函数及其函数梯度

3.2.5 损失函数

损失函数(loss function),又称为代价函数(cost function),是将随机事件或其有关随机变量的取值映射为非负实数以表示该随机事件的"风险"或"损失"的函数。

在实际应用中,损失函数通常作为学习准则与优化问题相联系,即通过最小化损失函数求解和评估模型。常见的损失函数如表 3-6 所示。

表 3-6 常见损失函数

名 称	表 达 式
铰链损失函数(Hinge Loss Function)	$L=(\hat{y},y)=\max(0,1-\hat{y}y)$
交叉熵损失函数(Cross-entropy Loss Function)	$L(\hat{y},y)=y\log(\hat{y})-(1-y)\log(1-\hat{y})$
指数损失函数(Exponential Loss Function)	$L(\hat{y},y)=\exp(-\hat{y}y)$
IOU 损失函数(Intersection over Union)	$L_{\text{IOU}}=1-\dfrac{\lvert B\cap B_{\text{gt}}\rvert}{\lvert B\cup B_{\text{gt}}\rvert}$
欧几里得损失函数(Euclidean Loss Function),也称为 L2 Loss 或均方差损失(Mean Squared Error Loss,MSE)函数	$L=\dfrac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$

续表

名　称	表　达　式
L1 范数损失函数(L1 Norm Loss Function)	$L=\frac{1}{n}\sum_{i=1}^{n}\mid y_i-\hat{y}_i\mid$
对比损失函数	$L(p,y)=\frac{1}{2N}\sum_{n}yd^2+(1-y)\max(0,m-d)^2,n\in[1,N]$
期望损失函数(Expected Loss Function)	$L(p,y)=\sum_{n}\left\vert y_n-\frac{\exp(p_n)}{\sum_{n}\exp(p_k)}\right\vert,n\in[1,N]$
结构相似性度量(Structural Similarity Index Measure Loss)函数	$\mathrm{SSIM(n)}=\frac{2\mu_{p_n}\mu_{y_n}+C_1}{\mu_{p_n}^2+\mu_{y_n}^2+C_1}\cdot\frac{2\sigma_{p_ny_n}+C_2}{\sigma_{p_n}^2+\sigma_{y_n}^2+C_2}$

表 3-6 中总结了部分 CNN 模型中常会使用到的损失函数类型，在针对某一具体任务时，要选用合适的损失函数对模型进行度量。

3.2.6　经典 CNN

一般来说，卷积神经网络由卷积层、池化层、全连接层以及激活函数组成，其基本结构如图 3-8 所示。在计算机视觉网络中，最常见的 CNN 模型有 LeNet、ResNet(残差网络)、VGG 卷积神经网络、DenseNet 和 AlexNet、EfficientNet。

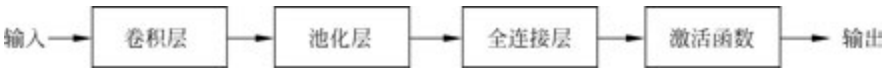


图 3-8　卷积神经网络

1. LeNet

LeNet[3]模型是 Yan LeCun 等于 1998 年提出的一种卷积神经网络模型。主要用于 MNIST 数据集的手写字符的识别与分类。如图 3-9 所示，LeNet 结构包括一个输入层、两个卷积层、两个池化层、三个全连接层(其中最后一个全连接层为输出层)。LeNet 的实现确立了 CNN 的结构，现在神经网络中的许多内容在 LeNet 的网络结构中都能看到。虽然 LeNet 早在 20 世纪 90 年代就已经提出，但由于当时缺乏大规模的训练数据和计算机硬件的性能较低，因此 LeNet 在处理复杂问题时效果并不理想。

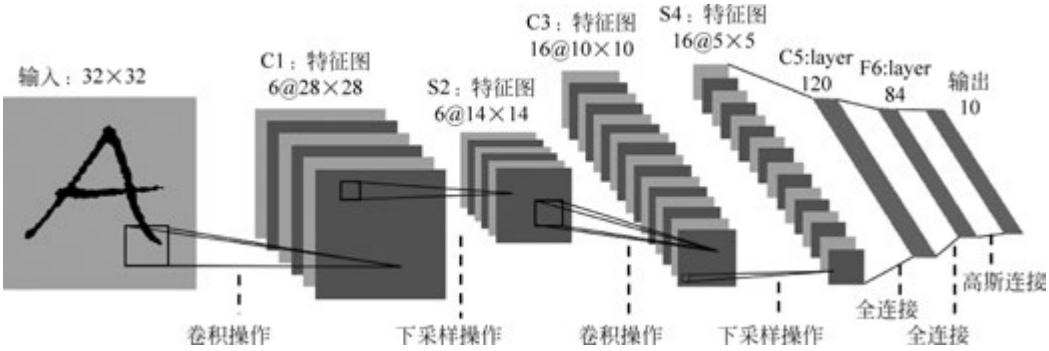


图 3-9　LeNet 模型结构图

2. VGG

VGG[4]卷积神经网络是由牛津大学 Visual Geometry Group(视觉几何组)于 2014 年研究提出的，如表 3-7 所示。VGG 卷积神经网络作为一种常用的图像分类网络，使用了大量的

小卷积核和池化层，并且也采用了深度网络结构。

表 3-7 VGG 网络结构

ConvNet 配置					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
输入图像(224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
最大池化					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
最大池化					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
最大池化					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
最大池化					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
最大池化					
全连接(Full Connected,FC)-4096					
全连接(Full Connected,FC)-4096					
全连接(Full Connected,FC)-1000					
Softmax 激活函数					

如表 3-7 所示，根据卷积核大小和卷积层数，VGG 共有 6 种类型，分别为 A、A-LRN、B、C、D、E。其中，D、E 是目前视觉神经网络中常用的两种结构，即 VGG-16 和 VGG-19。表 3-7 中所列数字从上往下表示通道数的数量，分别为 64、128、512、512、4096、4096、1000。随着通道数的增加，卷积层通道数翻倍，直到 512 时不再增加，而更多的特征信息也被提取出来。全连接的 4096 属于经验值，该值也可以更改，但不应该小于最后的类别(1000 表示分类的类别数)。VGG 网络的“层”(在 VGG 中称为 stage)是由多个 3×3 的卷积层叠加起来的，可以保留更多的特征信息，其网络层数可达 19 层。

以 VGG-16 模型为例，代码展示如下。

```
class VGG16(nn.Module):
    def __init__(self, num_classes = 10):
        super(VGG16, self).__init__()
```

```
        self.features = nn.Sequential(
            nn.Conv2d(3,64,kernel_size = 3,padding = 1),      #第 1 层
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.Conv2d(64,64,kernel_size = 3,padding = 1),     #第 2 层
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(64,128,kernel_size = 3,padding = 1),    #第 3 层
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.Conv2d(128,128,kernel_size = 3,padding = 1),   #第 4 层
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(128,256,kernel_size = 3,padding = 1),   #第 5 层
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.Conv2d(256,256,kernel_size = 3,padding = 1),   #第 6 层
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.Conv2d(256,256,kernel_size = 3,padding = 1),   #第 7 层
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(256,512,kernel_size = 3,padding = 1),   #第 8 层
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.Conv2d(512,512,kernel_size = 3,padding = 1),   #第 9 层
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.Conv2d(512,512,kernel_size = 3,padding = 1),   #第 10 层
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(512,512,kernel_size = 3,padding = 1),   #第 11 层
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.Conv2d(512,512,kernel_size = 3,padding = 1),   #第 12 层
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.Conv2d(512,512,kernel_size = 3,padding = 1),   #第 13 层
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.AvgPool2d(kernel_size = 1,stride = 1),
            )
        self.classifier = nn.Sequential(
            nn.Linear(512,4096),                              #第 14 层
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),                            #第 15 层
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096,num_classes),                      #第 16 层
            )
```

```
            #self.classifier = nn.Linear(512, 10)
        def forward(self, x):
            out = self.features(x)
            print(out.shape)
            out = out.view(out.size(0), -1)
            print(out.shape)
            out = self.classifier(out)
            print(out.shape)
            return out
```

3. ResNet

ResNet(Residual Neural Network,残差神经网络)[5]是由何恺明、孙剑等于2015年提出的最具影响力且使用最为广泛的神经网络结构。它通过使用ResNet Unit成功训练出了152层的神经网络,并在ILSVRC2015比赛中取得冠军,在Top5上的错误率为3.57%。相比其他网络结构,ResNet的参数量更低,但其效果却非常突出。ResNet在传统卷积神经网络中加入残差学习的思想,解决了随着网络层数的增加模型性能下降(由于网络模型层的增加导致出现梯度消失和梯度爆炸)的问题,使网络能够越来越深,同时既保证了精度,也控制了速度。

ResNet结构如表3-8所示。ResNet一共包括5个卷积组,每个卷积组中包含1个或多个基本的卷积计算过程(卷积—批处理—ReLU激活函数)。每个卷积组中包含1次下采样操作,使特征图大小减半,如表3-8中展示的输出特征尺寸大小,从第一组卷积块到第二组卷积块,输出特征尺寸由112×112变为56×56。在整个ResNet中,第一个卷积组只包含1次卷积计算操作,在该卷积组中,使用kernel 7×7,步长为2,padding为3的操作,之后进行BN、ReLU和最大池化操作。5种典型ResNet结构中的第一个卷积组完全相同,其卷积核大小均为7×7,步长为2;第2~5个卷积组都包含相同的残差单元,在很多实现代码上,通常把第2~5个卷积组分别叫作stage1、stage2、stage3、stage4。4个stage中,均使用make-layer( )来生成stage,每个stage中有多个模块,每个模块叫作building block,如ResNet-18=[2,2,2,2],就有8个building block。

表3-8 ResNet结构

(注:18-layer即ResNet-18,34-layer即ResNet-34,50-layer即ResNet-50;101-layer即ResNet-101,152-layer即ResNet-152)

层	输出尺寸	18-layer	34-layer	50-layer	101-layer	152-layer
conv1_x	112×112	7×7,64,步长为2				
conv2_x	56×56	3×3最大池化,步长为2				
		$\begin{bmatrix}3\times3,64\\3\times3,64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3,64\\3\times3,64\end{bmatrix}\times3$	$\begin{bmatrix}3\times3,64\\3\times3,64\\1\times1,256\end{bmatrix}\times3$	$\begin{bmatrix}3\times3,64\\3\times3,6\\1\times1,256\end{bmatrix}\times3$	$\begin{bmatrix}3\times3,64\\3\times3,64\\1\times1,256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3,128\\3\times3,128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3,128\\3\times3,128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1,128\\3\times3,128\\1\times1,512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1,128\\3\times3,128\\1\times1,512\end{bmatrix}\times4$	$\begin{bmatrix}3\times3,128\\3\times3,128\\1\times1,512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3,256\\3\times3,256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3,256\\3\times3,256\end{bmatrix}\times6$	$\begin{bmatrix}3\times3,256\\3\times3,256\\1\times1,1024\end{bmatrix}\times6$	$\begin{bmatrix}3\times3,256\\3\times3,256\\1\times1,1024\end{bmatrix}\times23$	$\begin{bmatrix}3\times3,256\\3\times3,256\\1\times1,1024\end{bmatrix}\times36$

续表

层	输出尺寸	18-layer	34-layer	50-layer	101-layer	152-layer
conv5_x	7×7	$\begin{bmatrix}3\times3,512\\3\times3,512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3,512\\3\times3,512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1,512\\3\times3,512\\1\times1,2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1,512\\3\times3,512\\1\times1,2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1,512\\3\times3,512\\1\times1,2048\end{bmatrix}\times3$
	1×1	平均池化，1000-d 全连接（Full Connected，FC），Softmax 激活函数				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ResNet 结构可分为两种，即 BasicBlock（如图 3-10(a)所示）和 BottleNeck（如图 3-10(b)）所示。前者应用于 ResNet-18、ResNet-34 模型，后者应用于 ResNet-50、ResNet-101、ResNet-152 模型。两种结构均是通过跳跃短连接（Shortcut Connections）和残差路径缓解深层网络梯度消失和梯度爆炸的问题。

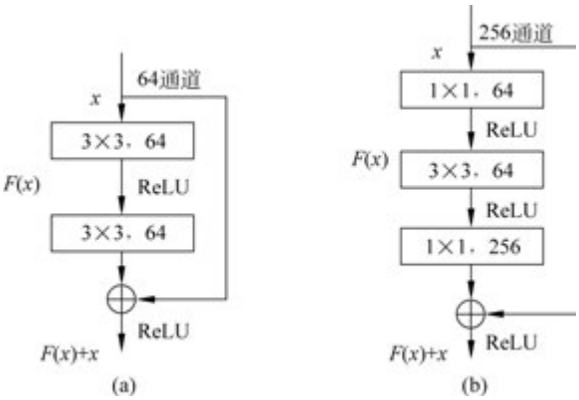


图 3-10　ResNet 的残差学习单元结构图

ResNet 中的 BasicBlock 示例代码如下。

```
class BasicBlock(nn.Module):
    """搭建 BasicBlock 模块"""
    expansion = 1
    def __init__(self, in_channel, out_channel, stride = 1, downsample = None):
        super(BasicBlock, self).__init__()
        # 使用 BN 层是不需要使用 bias 的，bias 最后会抵消掉
        self.conv1 = nn.Conv2d(in_channel, out_channel, kernel_size = 3, padding = 1, stride =
stride, bias = False)
        # BN 层，BN 层放在 Conv 层和 ReLU 层中间使用
        self.bn1 = nn.BatchNorm2d(out_channel)
        self.conv2 = nn.Conv2d(out_channel, out_channel, kernel_size = 3, padding = 1, bias =
False)
        self.bn2 = nn.BatchNorm2d(out_channel)
        self.downsample = downsample
        self.relu = nn.ReLU(inplace = True)
    # 前向传播
    def forward(self, X):
        identity = X
        Y = self.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
    # 保证原始输入 X 的 size 与主分支卷积后的输出 size 叠加时维度相同
        if self.downsample is not None:
          identity = self.downsample(X)
        return self.relu(Y + identity)
```

此外，如图 3-10(b)中 BottleNeck 结构所示，第一个函数层将对特征矩阵进行维度下降，即将其深度由原来的 256 降为 64，第三个函数层用于对特征矩阵进行维度提升，即将其深度由 64 还原为 256。这样既提高了模型在高维度下的表示能力，也减小了计算量。BottleNeck 示例代码如下。

```
class BottleNeck(nn.Module):
    """搭建 BottleNeck 模块"""
#BottleNeck 模块最终输出 out_channel 是 Residual 模块输入 in_channel 的 size 的 4 倍(Residual
#模块输入为 64),shortcut 分支 in_channel 为 Residual 的输入 64,因此需要在 shortcut 分支上将
#Residual 模块的 in_channel 扩张 4 倍,使之与原始输入图片 X 的 size 一致
    expansion = 4
    def __init__(self, in_channel, out_channel, stride = 1, downsample = None):
        super(BottleNeck, self).__init__()
    #默认原始输入为 256,经过 7×7 层和 3×3 层之后 BottleNeck 的输入降至 64
        self.conv1 = nn.Conv2d(in_channel, out_channel, kernel_size = 1, bias = False)
        #BN 层, BN 层放在 Conv 层和 ReLU 层中间使用
        self.bn1 = nn.BatchNorm2d(out_channel)
self.conv2 = nn.Conv2d(out_channel,out_channel,kernel_size = 3,stride = stride,padding = 1,bias =
False)
        self.bn2 = nn.BatchNorm2d(out_channel)
        self.conv3 = nn.Conv2d(out_channel, out_channel * self.expansion, kernel_size = 1,
bias = False)
        #Residual 中第三层 out_channel 扩张到 in_channel 的 4 倍
        self.bn3 = nn.BatchNorm2d(out_channel * self.expansion)
        self.downsample = downsample
        self.relu = nn.ReLU(inplace = True)
    #前向传播
    def forward(self, X):
        identity = X
        Y = self.relu(self.bn1(self.conv1(X)))
        Y = self.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
    #保证原始输入 X 的 size 与主分支卷积后的输出 size 叠加时维度相同
        if self.downsample is not None:
            identity = self.downsample(X)
        return self.relu(Y + identity)
```

综上所述，ResNet 的整体结构由若干个残差单元和瓶颈残差单元组成，其中每个残差单元包含若干个残差基础块。整个网络的输入是一个图片，经过多个卷积操作后输出一个具有对应概率的向量，用于后续任务操作。

3.3 生成式对抗网络

生成式对抗网络(Generative Adversarial Networks，GAN)[6]是由 Goodfellow 等于 2014 年的博弈论中提出的一种深度学习模型。GAN 是一种无监督的深度学习技术，通过让两个神经网络(生成网络和判别网络)相互博弈，以对抗训练的方式进行学习，使得 GAN 能够在不使用标注数据的情况下生成与真实数据相近的人工合成数据，对半监督或弱监督学习、图像生成等起着极大促进作用。目前，GAN 已被广泛应用于自然语言处理、计算机视觉、语义分割、自动驾驶、时间序列以及医疗等各种领域，尤其在计算机视觉的图像生成、图像分割、视频预测、风格迁移、图像超分辨率、数据增强领域影响深刻。例如，在图像生成方面，GAN 能够模拟真实图像进而生成与真实图像相似的合成图像，一定程度上缓解了计算机视觉中显著目标

检测中实验数据集不足问题；在风格迁移方面，GAN 通过将源图像输入生成器用于生成与源图像相同风格和特征的目标图像。

3.3.1 GAN 架构

GAN 提供了一个强大的网络架构来合成与真实数据相近的人工合成数据，该网络由生成器 G 和判别器 D 组成。G 旨在尽可能地生成无限接近真实数据的“假”数据，D 旨在尽可能地判别输入数据是真实数据还是由 G 生成的“假”数据。GAN 的基本架构如图 3-11 所示，G 以随机噪声 z 作为输入，通过一系列复杂的神经网络处理后，最终生成无限接近真实数据的“假”数据 $G(z)$。D 以真实数据 x 和“假”数据 $G(z)$作为输入，然后判别输入数据是 x 还是 $G(z)$。最后反向传播来微调训练 G 和 D，通过 G 与 D 不断进行对抗训练，在理想情况下，D 达到了难以判别其输入数据真假的程度，此时模型达到最优状态。

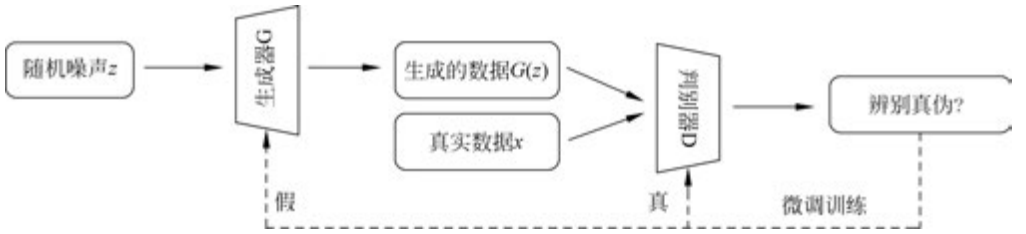


图 3-11　GAN 的基本架构图

GAN 设计了目标函数来交替优化 G 和 D，其公式如下。

$$\min_G \max_D V(D,G) = E_{x\sim P_{\text{data}}(x)}[\log D(x)] + E_{z\sim P_z(z)}[\log(1-D(G(z)))] \tag{3-4}$$

其中，E 为期望，$P_{\text{data}}(x)$为真实数据的分布，$P_z(z)$为输入噪声的分布。判别器处理的是一个二分类问题，$V(D,G)$为二分类问题中常见的二元交叉熵损失函数。$D(x)$表示判别器判断来自真实数据 x 的概率，$D(x)$越大，$V(D,G)$就越大，其判别能力越强。$D(G(z))$表示判别器判断“假”数据 $G(z)$的概率，G 生成的 $G(z)$越真，$D(G(z))$就越大，则 $V(D,G)$就越小。我们训练是为了使 D 能正确区分 $P_{\text{data}}(x)$和 $P_z(z)$，增强其判别能力，即增加 $D(x)$，减小 $D(G(z))$，使得 $V(D,G)$变大；同时提高 G 的生成能力以减少被 D 判别出来的概率，增大 $D(G(z))$，即最小化 $\log(1-D(G(z)))$。生成器 G 与判别器 D 两者之间的优化相互交替，以函数 $V(D,G)$做最大最小化博弈，进而提高 G 的生成能力与 D 的判别能力。

生成器和判别器进行交替训练是一个迭代优化的过程，由于训练数据有限，如果先将 D 优化完成会导致过拟合，最终导致模型不能收敛。先对判别器进行训练，然后对生成器进行训练，不断地进行重复，通常会训练迭代 k 次判别器（实践中一般 k 为 1），然后再迭代 1 次生成器。首先最大化判别器，即 $\max_D V(D,G)$，使 $V(D,G)$取得最大值；其次再最小化生成器，即最小化该值。固定生成器 G 不变，最大化 $V(D,G)$，公式如下。

$$\begin{aligned} V(D,G) &= \int_x P_{\text{data}}(x)\log(D(x))\mathrm{d}x + \int_z P_z(z)\log(1-D(g(z))) \\ &= \int_x P_{\text{data}}(x)\log(D(x)) + P_g(x)\log(1-D(x))\mathrm{d}x \end{aligned} \tag{3-5}$$

对 $V(D,G)$进行求导操作，将 $P_{\text{data}}(x)$和 $P_z(z)$视为常数，对 $D(x)$进行求导，得出 $D(x)$的极值，即最优的判别器的情况如下。

$$D_G^*(x) = \frac{P_{\text{data}}(x)}{P_{\text{data}}(x) + P_g(x)} \tag{3-6}$$

将其代入 $V(D,G)$ 的求导公式中，得出

$$\min_G V(G,D_G^*)=E_{x\sim P_{\text{data}}(x)}\left[\log\frac{P_{\text{data}}(x)}{P_{\text{data}}(x)+P_g(x)}\right]+E_{x\sim P_g(x)}\left[\log\frac{P_g(x)}{P_{\text{data}}(x)+P_g(x)}\right]$$
$$=-2\log(2)+2\text{JSD}(P_{\text{data}}(x)\,\|\,P_g(x)) \tag{3-7}$$

其中，JSD 表示 JS 散度(Jenson Shannon Divergence)，用于表示真实数据分布 $P_{\text{data}}(x)$ 和生成数据分布 $P_g(x)$ 之间的差异，其结果越小，证明两者越接近，反之，差异越大。JSD($p_{\text{data}}\,\|\,p_g$) 即真实数据与生成数据之间的差异。当 G 和 D 能力足够时，模型收敛，两者达到纳什均衡状态，即 $P_{\text{data}}(x)=P_g(x)$，判别器对真实数据和生成数据的判别概率各占 0.5，达到了理想情况下 D 难以辨别真伪的程度。

3.3.2 GAN 的训练策略

GAN 的训练策略是将生成器和判别器同时进行交替训练，逐步迭代优化，最终达到两者之间的平衡。图 3-12 展示了 GAN 的训练过程。粗虚线表示真实数据分布，细虚线表示判别器的判别分布，实线表示生成器生成的生成数据，下面的水平线 z 表示输入生成器的噪声分布，x 表示经过生成器映射后学到的真实数据分布，z 到 x 的箭头(即 $x=G(z)$)表示生成器将非均匀分布 p_g 映射到转换样本上。GAN 在训练时，不断更新判别器，使其能够判别是否为真实数据，生成器则生成与真实数据非常逼近的伪数据。图 3-12 (a)是原始状态，G 和 D 处于对抗状态，此时 D 还不能很好地判别数据真伪，G 生成的数据真实性也比较差。经过训练后，如图 3-12 (b)所示训练判别器，D 能正确区分数据真伪。如图 3-12 (c)所示训练生成器，D 能够更好地指导 G 生成更真实的数据，使 G 生成的数据更逼近真实数据。经过图 3-12(b)→图 3-12(c)→图 3-12(b)→图 3-12(c)这样的迭代交替后，预期结果如图 3-12(d)所示，当 G 和 D 的能力足够时，模型收敛。此时 D 达到了难以辨别数据真伪的程度，判别生成数据和真实数据的概率各占一半，模型达到最优状态。

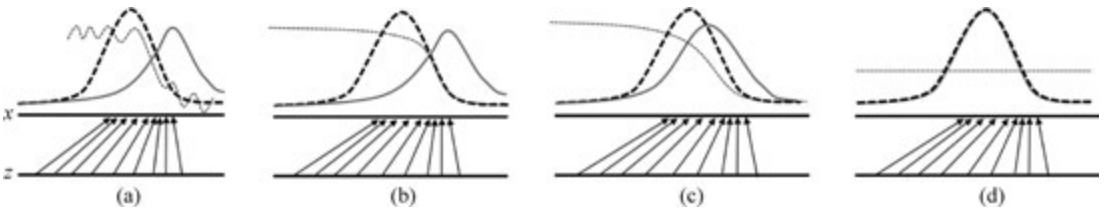


图 3-12 GAN 训练过程示意图

3.3.3 经典 GAN

虽然 GAN 在各个领域已取得了突破性的成就，但在实际应用中仍然存在一些问题。一是训练过程不稳定，易出现偏差，导致模型崩溃或震荡；二是无法充分合成高质量、真实多样的图像。为缓解以上两个问题，研究人员提出了许多改进方法，如网络结构、生成器、判别器、损失函数、优化技术、训练评估措施等方面的改进，基于这些改进方法，产生了许多 GAN 的变体，如 Conditional GAN[7]、MAD-GAN[8] 和 Dropout-GAN[9]，这些变体促进了生成模型的发展。

1. 基于网络结构改进的 GAN

原始 GAN 是一种无条件约束的模型，它无法控制生成特定类的数据。Mehdi Mirza 等通

过为 GAN 添加一些额外信息作为条件后就产生了条件生成对抗网络(Conditional Generative Adversarial Networks,CGAN)。CGAN 通过引入了额外的条件信息来控制模型的生成过程，对 G 和 D 添加类别标签作为条件，使 G 和 D 能够根据类别标签条件产生对应的输出，其结构如图 3-13 所示。CGAN 在原有 GAN 结构的基础上分别为 G 和 D 添加额外的条件信息 y(类别标签或特定的输入)作为输入。G 的输入有随机噪声 z 和条件向量 $\boldsymbol{y}$，它可以根据给定的输入来生成特定类别的图像。D 的输入有真实数据 x 和条件向量 $\boldsymbol{y}$，它可以根据给定的输入来判别生成的图像是否是真实的。其中，$\boldsymbol{y}$ 一般都是先独自进行编码，再与 z 和 x 进行连接的。例如，给出了“鸟”“狗”两个类别标签，CGAN 就可以生成鸟和狗的图像。

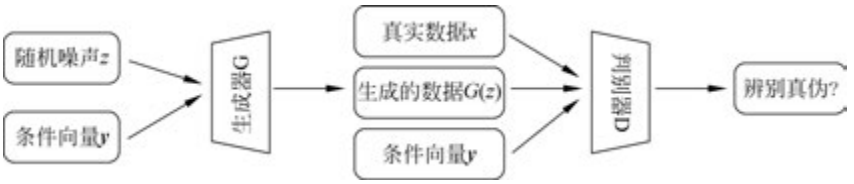


图 3-13　CGAN 结构图

CCGAN 主要应用于图像生成、图像转换、图像修复、图像编辑和视频生成等多个领域。其中最典型的一个应用是图像生成，根据给定的额外条件信息来生成特定类别的逼真图像。通过给网络提供额外的条件信息，如类别标签或文本描述，可以控制生成的图像的特征和属性，生成各种不同的图像样式和特征，使其成为一个有趣和强大的图像生成工具。例如，生成特定风格的人脸、动物、作品等。

除了上述 CGAN，还有许多其他经典的基于网络结构改进的 GAN(如 DCGAN[10]、InfoGAN[11]和 CycleGAN[12])，感兴趣的读者可查阅相关资料进行学习。

2. 基于生成器改进的 GAN

基于生成器改进的 GAN 通过改进生成器的网络构架来提高 GAN 的整体效率。为解决模式崩塌问题，MADGAN(Multi-Agent Diverse GAN)中使用了多个生成器和一个判别器的 GAN 结构，使生成器能够捕获到不同的高概率区域，产生多样化的输出。为了强制不同的生成器执行多样性效果，MADGAN 对标准 GAN 目标函数进行了两个扩展，从而避免所有的生成器学会生成多个类似的样本的问题。第一个扩展为多样性强制项，其被添加到生成器的目标函数中，鼓励不同的生成器使用用户定义的基于相似性的函数生成不同的样本。第二个扩展是修改了鉴别器的目标函数，在找到真实和虚假样本的同时，鉴别器必须预测产生给定虚假样本的发生器。

MADGAN 结构如图 3-14 所示，该模型由多个生成器和一个判别器构成，判别器判别输入数据是真实数据还是生成数据，每个生成器学习不同的模式，生成不同的生成数据，显式地保证了生成数据的多样性，缓解了模式崩塌问题。由于生成器的初始层捕获的高频结构对于特定类型的数据集(例如人脸)几乎是相同的，因此生成器之间的大多数初始层是共享权值的，这样可以避免冗余计算。

MADGAN 可以产生多样化的样本，已被应用到多样化图像生成、跨域风格迁移、图像编辑和重构、图像到图像的翻译等具有挑战性的任务中。在跨域风格迁移中，MADGAN 通过调整输入条件，可以将一个域中的图像转换成另一个域的图像，同时保留多个属性。例如，将黑白图像转换为多种不同的彩色图像。但是，由于多个生成器的引入，导致模型的复杂度变高，增加了计算资源和时间。因此，需要充分考虑其优缺点，根据具体应用场景选择使用该模型。

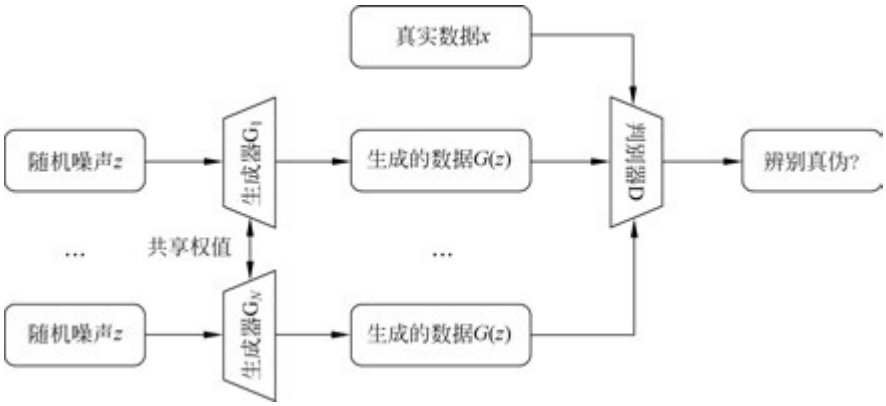


图 3-14　MADGAN 结构图

除了本节列举的模型，还有许多经典的基于生成器改进的 GAN 模型（如 MGAN[13]、MPMGAN[14]），感兴趣的读者可查阅相关资料进行学习。

3. 基于判别器改进的 GAN

基于判别器改进的 GAN 与基于生成器改进的 GAN 类似，主要通过改进判别器的网络结构来加速模型。Dropout-GAN（GAN with Dropout）中提出了使用一种 Dropout 技术来动态集成判别器的框架，该框架使单个生成器能够从一组判别器中学习。Dropout 技术可以根据一定的概率选择性地丢弃判别器中的一些神经元，动态地改变判别器的神经组合，促进了生成样本的多样性，避免了 GAN 常见的模式崩塌问题，进而提高模型的生成效率。

Dropout-GAN 结构如图 3-15 所示，该模型由多个判别器和一个生成器组成，通过在每个判别器的反馈上应用 Dropout 技术，以一定的概率丢弃一些结果，然后将剩余的结果聚合后反馈到生成器，迫使生成器不依赖于某个特定的判别器，而是依赖不同判别器给出的反馈，从而使生成器学习多种样本。

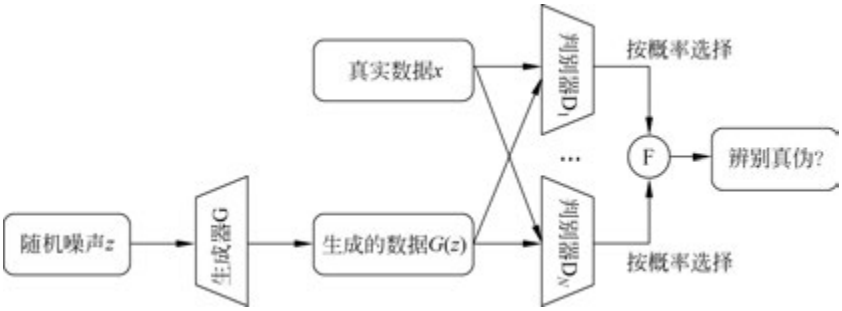


图 3-15　Dropout-GAN 结构图

Dropout-GAN 充分结合了 GAN 和 Dropout 的优势，被用来生成多样的高质量假数据，可以应用于图像生成、图像修复、数据增强、异常检测和文本生成等任务。例如，在图像修复中，通过在生成器中使用 Dropout 技术，可以生成不完整或损坏图像的修复版本，从而实现图像修复。但是，其也不可避免地占据较大的计算成本，未来可以根据判别器集的大小来调整生成器的学习，促使模型变得更加稳定。

还有许多经典的基于判别器改进的 GAN 模型（如 EBGAN[15]、GMAN[16]），感兴趣的读者可查阅相关资料进行学习。

3.3.4 GAN 案例

Zhang 等[17]提出了一个用于解决单个图像去雨问题的网络 ID-CGAN，通过引入一个强大的生成建模能力的 CGAN，并加上一个强制约束，即去雨后的图像必须与相应的背景图像不可区分。ID-CGAN 主要由两个部分组成：生成器 G 和判别器 D。图 3-16 给出了该网络的整体构架图，G 是一个对称的密集连接网络（DenseNet），具有适当的跳跃连接，其主要目标是从一个真实图像（被雨水污染的图像）中合成“假”图像（被雨水污染的图像）。D 利用全局和局部信息来判别图像真伪，通过多尺度池化来捕获上下文信息，其主要用于将“假”图像与对应的真实图像区分开。G 和 D 根据去雨的条件来生成或判别图像。

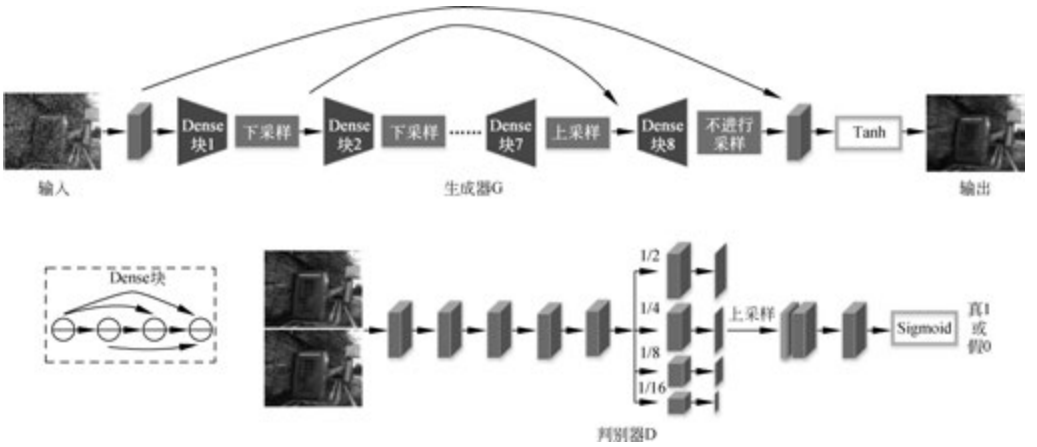


图 3-16　用于单个图像去雨的 ID-CGAN

1. 生成器

生成器的网络构架的内部模块构建如表 3-9 所示，其代码实现如下。

表 3-9　ID-CGAN 的网络构架

生成器的网络构架	判别器的网络构架
输入，通道数＝3	输入，通道数＝6
3×3 卷积，BN，ReLU，MaxP，通道数＝64	
密集块（4 层）＋下采样，通道数＝128	3×3 卷积，BN，ReLU，MaxP，通道数＝64
密集块（6 层）＋下采样，通道数＝256	3×3 卷积，BN，ReLU，MaxP，通道数＝256
密集块（8 层）＋不进行采样，通道数＝512	3×3 卷积，BN，ReLU，MaxP，通道数＝512
密集块（8 层）＋不进行采样，通道数＝128	3×3 卷积，BN，ReLU，MaxP，通道数＝64
密集块（6 层）＋上采样，通道数＝120	四级池化模块，通道数＝72
密集块（4 层）＋上采样，通道数＝64	
密集块（4 层）＋上采样，通道数＝64	Sigmoid 激活函数
密集块（4 层）＋不进行采样，通道数＝16	
3×3 卷积，Tanh 激活函数，通道数＝3	输出，通道数＝72
输出，通道数＝3	

```
def build_generator(self):
    #定义生成器的卷积层
    def Conv2d(layer_input,no_filters,kernel,stride,bn = False,padding = 'valid'):
        x = Conv2D(filters = no_filters,kernel_size = kernel,strides = stride,padding = padding)
(layer_input)
```

```
        x = BatchNormalization(momentum = 0.8)(x)
        x = Activation('relu')(x)
        return x
    #使用跳跃连接的密集网络模型中的密集块
    def dense_block(layer_input,num_layers):
        x_list = [layer_input]
        for i in range(num_layers):
            x = Conv2D(filters = 32,kernel_size = (3,3),padding = 'same')(layer_input)
            x = BatchNormalization()(x)
            x = LeakyReLU()(x)
            x_list.append(x)
            x = Concatenate(axis = -1)(x_list)          #级联所有跳跃连接
        return x
      #上采样块
      def Deconv2d(layer_input, filters, kernel = 4, dropout_rate = 0):
          x = UpSampling2D(size = 2)(layer_input)
          x = Conv2D(filters, kernel_size = kernel, strides = 1, padding = 'same', activation = 'relu')(x)
          if dropout_rate:
              x = Dropout(dropout_rate)(x)
          x = BatchNormalization(momentum = 0.8)(x)
          return x

    inp = Input(shape = self.img_shape)                 #输入
    #下采样
    x0 = Conv2d(inp,64,(3,3),(1,1),bn = True)
    x0 = MaxPooling2D()(x0)
    x1 = dense_block(x0,4)
    x1 = Conv2d(x1,128,(3,3),(2,2),bn = True)
    x2 = dense_block(x1,6)
    x2 = Conv2d(x2,256,(3,3),(2,2),bn = True)
    x3 = dense_block(x2,8)
    x3 = Conv2d(x3,512,(3,3),(1,1),bn = True,padding = 'same')
    x4 = dense_block(x3,8)
    x4 = Conv2d(x4,128,(3,3),(1,1),bn = True,padding = 'same')
    #上采样
    x5 = dense_block(x4,6)
    x5 = Deconv2d(x5,120)
    x6 = dense_block(x5,4)
    x6 = Deconv2d(x6,64)
    x7 = dense_block(x6,4)
    x7 = Deconv2d(x7,64)
    x8 = dense_block(x7,4)
    x8 = Conv2d(x8,16,(3,3),(1,1),bn = True,padding = 'same')
    x9 = ZeroPadding2D(padding = (5,5))(x8)
    x10 = Conv2D(filters = 3,kernel_size = (3,3))(x9)
    out = Activation('tanh')(x10)
    return Model(inp,out)
```

2. 判别器

判别器的网络构架的内部模块构建如表 3-9 所示，其代码实现如下。

```
def build_discriminator(self):
#定义判别器层
def d_layer(layer_input, filters, f_size = 4, bn = True):
    x = Conv2D(filters, kernel_size = f_size, strides = 1)(layer_input)
```

```
    x = PReLU()(x)
    if bn:
      x = BatchNormalization(momentum = 0.8)(x)
      x = MaxPooling2D()(x)
      return x
#反卷积层
def Deconv2d(layer_input, filters, kernel = 4, dropout_rate = 0):
    x = UpSampling2D(size = 2)(layer_input)
    x = Conv2D(filters, kernel_size = kernel, strides = 1, padding = 'same', activation = 'relu')(x)
    if dropout_rate:
      x = Dropout(dropout_rate)(x)
      x = BatchNormalization(momentum = 0.8)(x)
    return x
def Pyramid_Pool(layer_input):                              #空间金字塔
    x_list = [layer_input]
def Pool(size):
    x = MaxPooling2D(pool_size = (size * 2,size * 2))(layer_input)
    for i in range(size):
        x = Deconv2d(x,2)
    return x

  x_list.append(Pool(1))                                    #金字塔的第一层
  x2 = MaxPooling2D(pool_size = (4,4))(layer_input)         #金字塔的第二层
  x2 = Deconv2d(x2,2)
  x2 = Deconv2d(x2,2)
  x2 = ZeroPadding2D(padding = (1,1))(x2)
  x_list.append(x2)
  x3 = MaxPooling2D(pool_size = (8,8))(layer_input)         #金字塔的最后一层
  x3 = Deconv2d(x3,4)
  x3 = Deconv2d(x3,4)
  x3 = Deconv2d(x3,4)
  x3 = ZeroPadding2D(padding = (3,3))(x3)
  x_list.append(x3)
  x = Concatenate(axis = -1)(x_list)
  return x
```

3.4 视觉注意力网络

Transformer是Google公司在2017年于文章*Attention is All You Need*[18]中提出的一种基于自注意力机制(Self-Attention Mechanism)的神经网络模型，最早应用于自然语言处理(Natural Language Processing，NLP)领域，如机器翻译、文本分类任务等，通过使用自注意力机制来提高模型对于长序列数据的建模能力。Transformer采用多层编码器-解码器结构，如图3-17所示，每一层都由多个注意力模块和前馈神经网络(Feed-Forward Neural Network，FFN)模块组成。编码器用于将输入序列编码成一个高维特征向量表示，解码器则用于将该特征向量表示解码为目标序列。其中，模型中还使用了残差连接和层归一化处理技术来加速模型收敛和提高模型性能。

随着技术的不断发展，Google在2020年于文章*An Image Is Worth* 16×16 *Words*：*Transformers For Image Recognition At Scale*[19]中提出了Transformer网络模型的变种模型——Vision Transformer (ViT)，将Transformer迁移到计算机视觉领域应用于图像分类任务。ViT的模型相对于原始Transformer更加简单，仅借鉴了Transformer中的编码器结构，其整体网络结构如图3-18所示。由于ViT在图像分类任务上取得了优异性能，逐渐拓展到其他视觉任务上，如目标检测、语义分割、图像修复等。

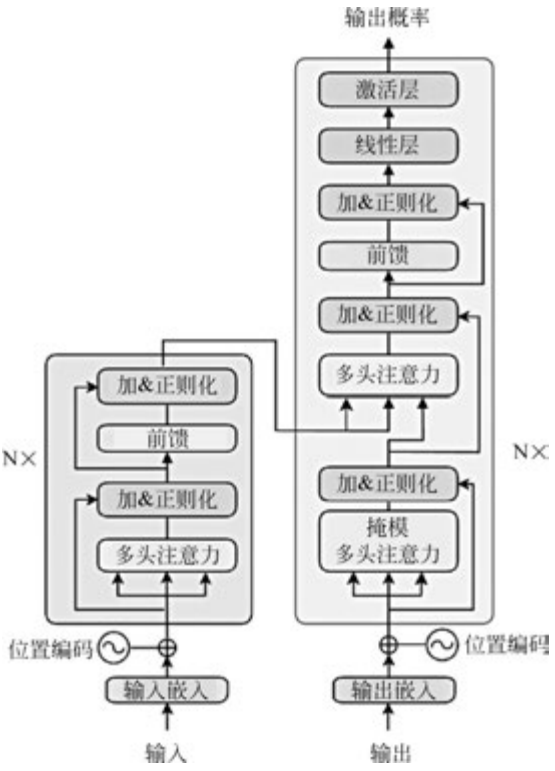


图 3-17　Transformer 网络模型

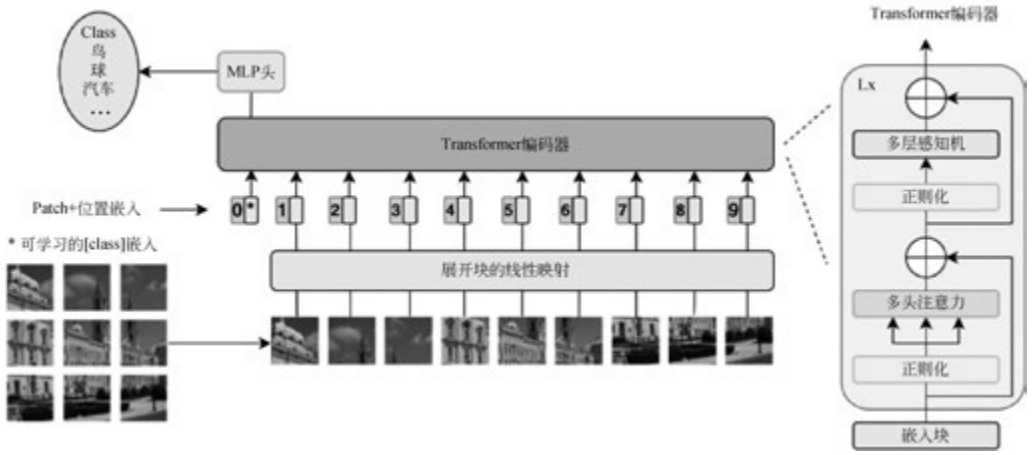


图 3-18　Vision Transformer 网络模型

由图 3-18 可知，ViT 模型从整体上可以分为三部分，分别是线性嵌入层（Linear Projection of Flattened Patches）、编码器层（Transformer Encoder）和分类层（MLP Head）。下面将对 ViT 模型每部分的核心模块进行详细介绍。

3.4.1　线性嵌入层

1. 图像块划分

标准的 Transformer 编码器接收的输入序列是一组二维矩阵，如图 3-18 所示，0～9 对应的都是向量数据。但是图像数据是一组三维矩阵，格式为$[H,W,C]$，不满足 Transformer 编码器的输入要求，所以需要通过一个 PE（Patch Embedding，块嵌入）操作来变换输入数据的维度。

给定一张尺寸大小为 224×224 的输入图像，其形状为[16,3,224,224]，其中，16 为 batch_size，3 为通道数，224 为像素数（$H=224, W=224$）。首先将该输入图像按照 16×16 大小的 patch 划分成一组固定大小的图像块，即一系列局部的特征块，划分后会得到 $(224/16)^2=196$ 个 patches，每个 patch 的形状为[16,16,3]。其次，通过线性映射操作，将每个 patch 映射到一维向量中，得到一个长度为 768 的向量（即 token）。PE 操作具体可以通过一个卷积层来实现，如图 3-19 所示。

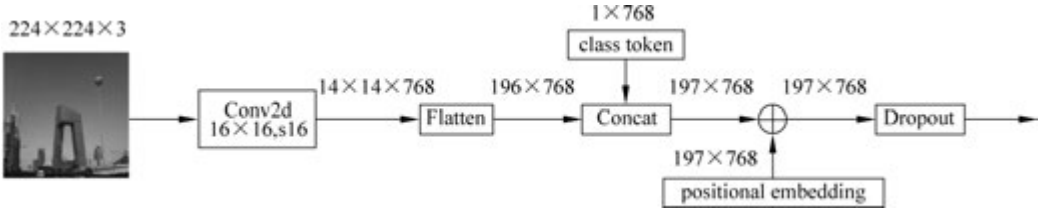


图 3-19　线性嵌入层核心步骤

其各项操作的代码如下。

（1）2D 卷积操作：卷积核大小为 16×16，步长为 16，卷积核数为 768。

```
img_size = 224, patch_size = 16, in_c = 3, embed_dim = 768, norm_layer = None
self.patch_embeddings = Conv2d(in_channels = in_channels, out_channels = config.hidden_size,
                               kernel_size = patch_size, stride = patch_size)
x = self.patch_embeddings(x)          #卷积操作，把图像分成指定的 patch
print(x.shape)                        #torch.Size([16, 768, 14, 14])
```

通过卷积操作，图像尺寸由原来的[224,224,3]变为[14,14,768]。

（2）Flatten：将图像块的 H 和 W 两个维度展平。

```
x = x.flatten(2)
print(x.shape)                        #torch.Size([16, 768, 196])
x = x.transpose(-1, -2)
print(x.shape)                        #torch.Size([16, 196, 768])
```

经过卷积层的处理，最后得到一个二维矩阵[196,768]，即 Transformer 编码器的输入数据。

2. 位置编码

图像经过位置编码（Positional Encoding）处理之后被分成一组二维矩阵，传入 Transformer 编码器中进行下一步处理。如图 3-19 所示，在将特征输入 Transformer 编码器之前，需要在得到的 tokens 中插入一个专门用于分类的[class]token，其作用是整合所有序列的特征信息，用于图像分类。这个[class]token 是一个可训练参数，其数据格式和其他 token 相同，都是一个长度为 768 的向量，即[1,768]。最后，将[class]token 在维度为 1 上与之前从图片中生成的 tokens 进行拼接，进而得到 shape=[197,768]的特征。由于图像切分后获得的 patches 不具有位置信息，为了使模型能够处理序列信息，需要在输入的图像块表示中引入位置编码。位置编码是一个与位置相关的向量，与[class]token 相同，positional embedding 采用的也是一个可训练参数，其数据格式与其他 token 相同，通过将 positional embedding 与图像块的表示相加，但维度保持不变，使模型能够区分不同位置的信息。其代码如下。

```
#引入用于分类的[class] token
self.cls_token = nn.Parameter(torch.zeros(1, 1, config.hidden_size))
```

```
cls_tokens = self.cls_token.expand(B, -1, -1)
x = torch.cat((cls_token, x), dim=1)          #[197, 768]

#引入位置编码
self.position_embeddings = nn.Parameter(torch.zeros(1, n_patches+1, config.hidden_size))
embeddings = x + self.position_embeddings
print(embeddings.shape)                       # torch.Size([16, 197, 768])
embeddings = self.dropout(embeddings)
print(embeddings.shape)                       # torch.Size([16, 197, 768])
return embeddings
```

3. Dropout

Dropout 的作用是防止模型在训练过程中出现过拟合问题。其代码如下。

```
self.dropout = Dropout(config.transformer["dropout_rate"])
embeddings = self.dropout(embeddings)
```

尺寸为 224×224 的输入图像经过线性嵌入层的处理后，每个图像块都有了相应的位置信息。接下来，特征会被传入 Transformer 编码器中进行特征提取。

3.4.2 Transformer 编码器

如图 3-18 所示，Transformer 编码器主要由正则化＋多头注意力和正则化＋多层感知机两部分组成，通过堆叠 L 次构成 Transformer 编码器。其中，层正则化的作用是在图像样本内做归一化操作，不考虑类间差异，维持值域稳定。多头自注意力是由多组自注意力结合而成，让模型关注不同方面的信息以捕捉更加丰富的特征，而自注意力是一种特殊的注意力机制。下文将对注意力机制做详细阐述。

注意力机制(Attention Mechanism)是一种模仿人类视觉或听觉注意力过程的计算机科学方法，主要用于机器学习和深度学习领域。从本质上来讲，类似于人类在感知信息时的选择性注意力机制，即关注重要细节而忽略不重要的部分，其核心目标是从众多信息中选择出符合当前任务目标的信息。注意力机制的引入使模型更灵活地处理复杂的输入数据，对数据中的不同位置或特征分配不同的权重，以便在处理过程中更加关注重要部分，提高模型对输入数据的理解和表达能力。

自注意力机制(Self-Attention Mechanism)是注意力机制的一种特殊情况，通常用于处理序列数据，例如，自然语言处理中的句子或时间处理数据。在视觉任务中，自注意力机制处理的是图像序列。自注意力的核心思想就是序列中的每个元素都可以和序列中的其他元素建立关联，而不仅依赖于相邻位置的元素。通过计算元素之间的相对重要性来自适应地捕捉元素之间的长程依赖关系。其计算过程主要包括以下三个步骤。

(1) 计算注意力权重：计算每个位置与其他位置之间的注意力权重，即每个位置对其他位置的重要性。

(2) 计算加权和：将每个位置向量与注意力权重相乘，然后将它们相加，得到加权和向量。

(3) 线性变换：对加权和向量进行线性变换，得到最终的输出向量。

假设图 3-20 中的 x^1、x^2、x^3、x^4 表示自注意力模块接收的输入序列数据，其中，x^1 可以是图像序列中的第一个 patch 对应的向量。首先，将每个输入数据 x 进行一个线性变换，变为 $a^i=\boldsymbol{W}x^i$；其次，将每个 a^i 分别乘上三个不同的转移矩阵 $\boldsymbol{W}^q$、$\boldsymbol{W}^k$、$\boldsymbol{W}^v$，得到三个子向量 q^i、k^i、v^i，分别代表着 Q(Query)、K(Key)、V(Value)。得到 Q、K、V 后，将每个 q^i 对每个 k^i 做

attention 操作来比较两个向量的接近程度，进而得到 $a_{i,j}$，计算公式如式(3-8)所示。

$$a_{i,j} = q^i \cdot k^i / \sqrt{d} \tag{3-8}$$

其中，d 为 q 和 k 的维度，除以$\sqrt{d}$ 以达到归一化的效果。将得到的 $a_{i,j}$ 进行 Softmax 操作得到 $\hat{a}_{i,j}$，并将其与所有的 v^i 相乘，然后将结果相加得到 b^1，计算公式如下，具体过程如图 3-21 所示。

$$b^1 = \sum_i \alpha_{1,i} v^i \tag{3-9}$$

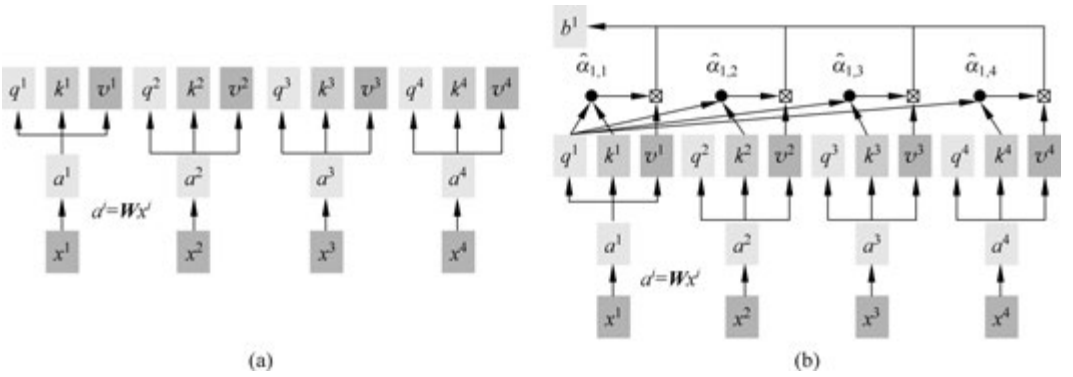


图 3-20　自注意力机制计算原理

多头自注意力(Multi-head self-attention)是由多组自注意力结合而成，通过将 q^i，k^i，v^i 拆成多个 head 来提升信息的利用效果，进而得到丰富的上下文信息。以两个 head 为例，如图 3-21 所示。首先，将 q^i 拆分成 $q^{i,1}$ 和 $q^{i,2}$（即再乘以两个转移矩阵 $\boldsymbol{W}^{q,1}$ 和 $\boldsymbol{W}^{q,2}$）、k^i 拆分成 $k^{i,1}$ 和 $k^{i,2}$（即再乘以两个转移矩阵 $\boldsymbol{W}^{k,1}$ 和 $\boldsymbol{W}^{k,2}$），v^i 拆分成 $v^{i,1}$ 和 $v^{i,2}$（即再乘以两个转移矩阵 $\boldsymbol{W}^{v,1}$ 和 $\boldsymbol{W}^{v,2}$）。其次，$q^{i,1}$ 和 $k^{i,1}$ 做 attention 操作得到 α，再通过 Softmax 计算及与 $v^{i,1}$ 计算得到最终的 $b^{i,1}$，同理可得到 $b^{i,2}$，将其合并即可得到 self-attention 中的 b^i。

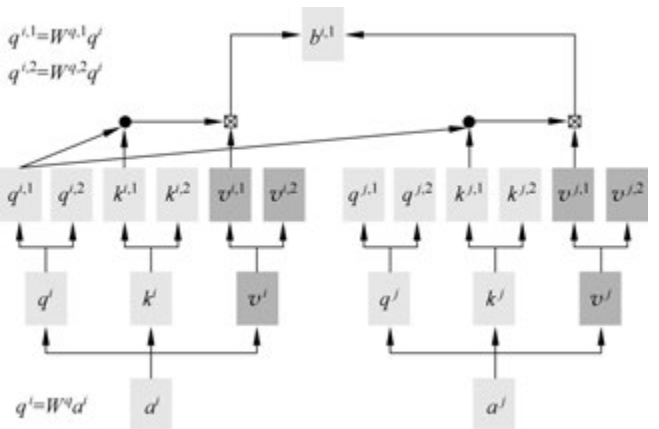


图 3-21　多头注意力机制计算原理

经多头自注意力处理后的特征，再经过 LN 和 MLP 操作，即可得到最终的编码器特征。具体 MLP 的相关操作详见 3.5 节。因 Transformer 解码器的结构同编码器结构类似，本节不再过多赘述，具体内容详见 ViT 论文。

3.5　多层感知机

多层感知机(Multi-Layer Perceptron，MLP)是一种在深度学习中非常常见的前馈神经网络，具有较强的表达能力和泛化能力，可以处理非线性问题和高维数据，被广泛应用于计算机

视觉、自然语言处理和推荐系统等领域。

3.5.1　多层感知机的原理

感知机(Perceptron Learning Algorithm,PLA)由美国科学家 Frank Rosenblatt 于 1957 年提出,是机器学习领域的一个经典算法,也是 MLP 的基础。PLA 的基本原理是通过计算输入特征的线性组合,然后使用激活函数对结果进行二分类。它的输入是一组实数特征向量 $X=(x_1,x_2,\cdots,x_n)$,每个特征都有相应的权重 $W=(w_1,w_2,\cdots,w_n)$以及偏移量 b,而输出 y 是一个二值化结果,表示输入样本属于两个类别中的某一类。感知机的结构如图 3-22 所示。感知机的目的是找到一个能够将不同类别的样本正确分类的超平面,如图 3-23 中橙色的线,从而实现对未知样本的预测。

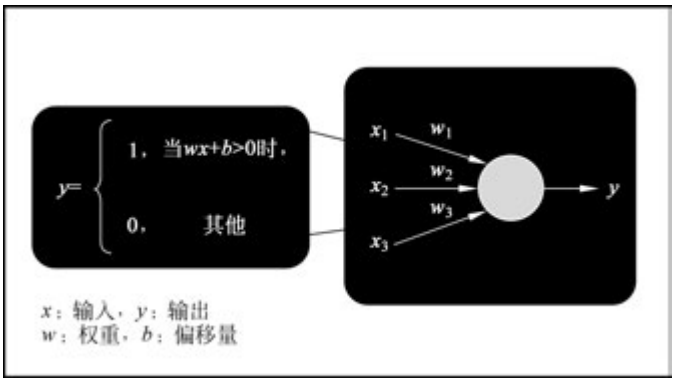


图 3-22　感知机的结构

感知机是一个无法处理异或问题的线性模型,图 3-24 展示了二维空间中的一个例子。而 MLP 克服了线性模型的限制,它首先学习图 3-24 中虚线的分类,点 1 和 3 为一类记为“+”,点 2 和 4 为一类记为“−”;其次再学习实线的分类,点 1 和 2 为一类记为“+”,点 3 和 4 为一类记为“−”;最后判断每个点两次分类时的符号是否一致,并以此分为两类,如图 3-25 所示,点 1 和 4 为一类,点 2 和 3 为另一类,符合图中按颜色的分类。上述学习两色点分类的过程在神经网络中被称为隐藏层。通过加入一个或多个隐藏层,网络就能够克服线性模型的限制,从而获得处理更普遍的函数关系类型的能力。为了实现这一目标,各层之间通常以全连接的方式堆叠在一起,其中的结果层层传递,直到生成最后的输出,而这种结构就被称为多层感知机,如图 3-26 所示。

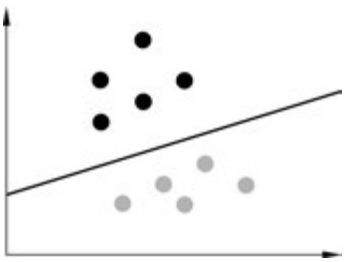


图 3-23　感知机在二维空间中的分类

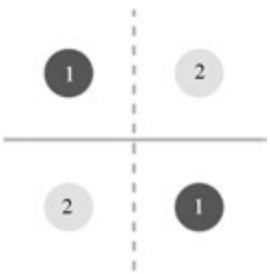


图 3-24　感知机无法找到一条直线分类两种颜色的点

扫码看彩图

	1	2	3	4
按垂直线分类	+	−	+	−
按水平线分类	+	+	−	−
结果	+	−	−	+

图 3-25　MLP 的学习过程

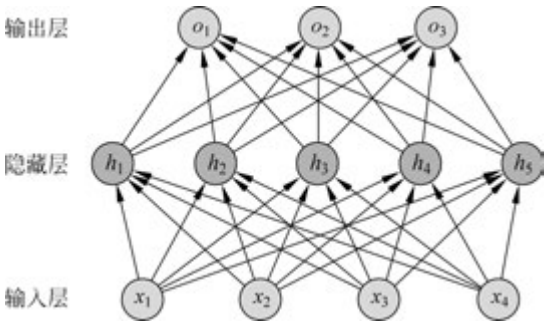


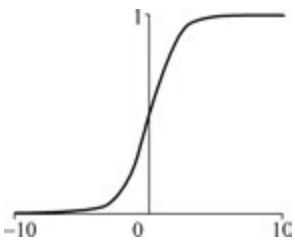
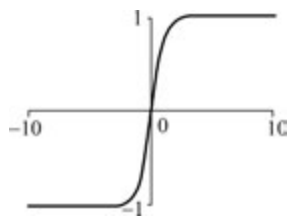
图 3-26　单隐藏层的 MLP

MLP 输入是一组实数特征向量 $X=(x_1,x_2,\cdots,x_n)$，隐藏层每个特征相应权重为 $W_1=(w_1,w_2,\cdots,w_n)$，偏移量为 b_1，输出层每个特征相应权重为 $W_2=(w_1,w_2,\cdots,w_n)$，偏移量为 b_2，那么单隐藏层的 MLP 可以由如下公式表示。

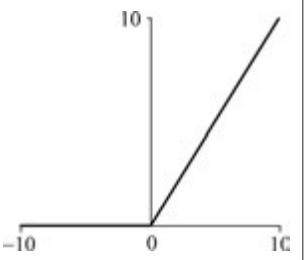
$$\text{hidden}=\sigma(W_1X+b_1),\quad \text{Out}=W_2^{\mathrm{T}}h+b_2 \tag{3-10}$$

其中，hidden 表示隐藏层的计算结果，Out 表示输出层的计算结果，σ 表示激活函数。激活函数的作用是给神经元引入非线性因素，使得神经网络可以任意逼近任何非线性函数，从而可以引入更多的非线性模型中。若不使用激活函数或使用线性的激活函数，无论神经网络有多少层，输出都是输入的线性组合，用一个单层感知机就可以表示了。表 3-10 列举了三个 MLP 常用的激活函数。

表 3-10　MLP 常用的激活函数

函数	公　式	图　像	特　点	缺　点
Sigmoid	$\text{sigmoid}(x)=\dfrac{1}{1+\exp(-x)}$		将输入值映射到范围为[0,1]的区间，具有平滑的S形曲线，非常适合概率预测	当输入过大或过小时，会出现梯度消失。 输出不以0为中心，可能导致额外偏置，影响权重更新效率。 执行指数运算，运算速度慢
双曲正切函数 Tanh	$\tanh(x)=\dfrac{1-\exp(-2x)}{1+\exp(-2x)}$		输入值映射到范围为[−1,1]的区间。以0为中心，具有平滑的S形曲线，通常用于隐藏层中	当输入过大或过小时，会出现梯度消失。 执行指数运算，运算速度慢

续表

函数	公 式	图 像	特 点	缺 点
整流线性单位函数 ReLU	$\text{relu}(x)=\max(x,0)$		不涉及复杂运算，不存在梯度消失的问题	当输入为负时，ReLU 完全失效，在反向传播中可能导致神经元在任何数据上都无法被激活。输出不以 0 为中心，可能导致额外偏置，影响权重更新效率

如下代码展示了一个 MLP 的简单实现。

```
pip install d2l == 0.17                              # 安装 d2l 库
import torch
from torch import nn
from d2l import torch as d2l

# MLP 模型,输入大小为 784,输出大小为 10,隐藏层大小为 256
net = nn.Sequential(nn.Flatten(),nn.Linear(784,256),nn.ReLU(),nn.Linear(256,10))
def init_weight(m):
  if type(m) == nn.Linear:
    nn.init.normal_(m.weight,std = 0.01)             # 初始化权重,没有 bias,默认为 0
net.apply(init_weight)                               # 应用初始化权重函数

#训练代码
batch_size,lr,num_epochs = 256,0.1,10                # 训练参数
loss = nn.CrossEntropyLoss()
trainer = torch.optim.SGD(net.parameters(),lr = lr)
train_iter,test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
d2l.predict_ch3(net,test_iter)
```

3.5.2 纯 MLP 神经网络

Tolstikhin 等提出了 MLP-Mixer[20]，这是一种完全基于多层感知器（MLP）的架构。MLP-Mixer 包含两种类型的层：一种是独立应用于图像补丁的 MLP（即“混合”每个位置的特征），另一种是跨补丁应用的 MLP（即“混合”空间信息）。当在大型数据集上训练或使用现代正则化方案时，MLP-Mixer 在图像分类基准上获得了具有竞争力的分数，其预训练和推理成本与最先进的模型相当，如图 3-27 所示。

MLP-Mixer 包括三部分：Per-patch Fully-connected、Mixer Layer、分类器。

1. Per-patch Fully-connected

MLP-Mixer 通过 Per-patch Fully-connected 将输入图像转换为 2D 的 Table，方便在后面进行局部区域间的信息融合。具体来说，MLP-Mixer 将输入图像相邻无重叠地划分为 S 个 Patch，每个 Patch 通过 MLP 映射为一维特征向量，其中向量长度为 C，最后将每个 Patch 得到的特征向量组合得到大小为 $S\times C$ 的 2D Table。需要注意的是，每个 Patch 使用的映射矩阵相同，即使用的 MLP 参数相同。

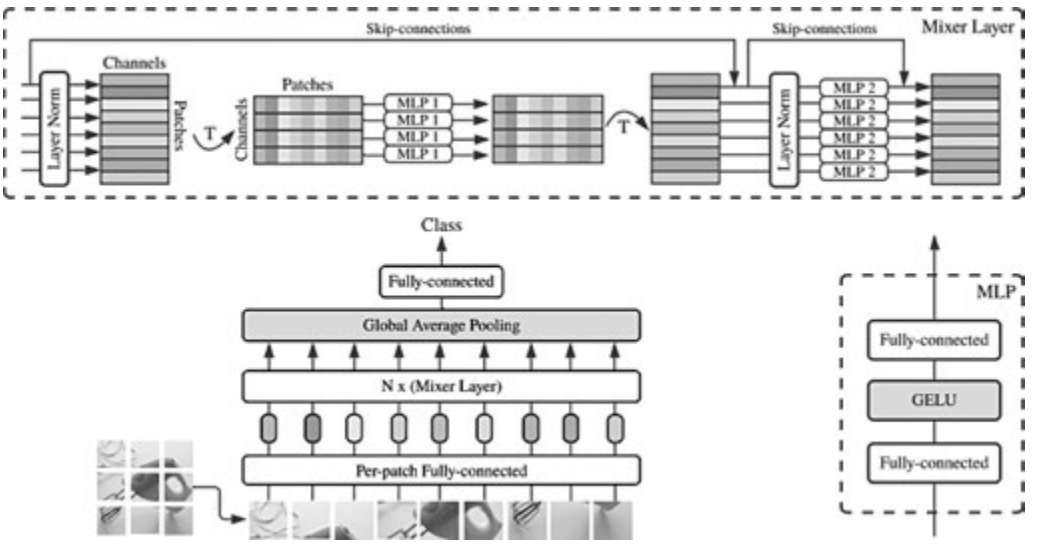


图 3-27　MLP-Mixer 网络结构图

实际上，Per-patch Fully-connected 实现了三维图像(W，H，C)的向量空间到二维(S，C)的向量空间的映射。例如，假设输入图像大小为 240×240×3，模型选取的 Patch 为 16×16，那么一个图片可以划分为(240×240)/(16×16)= 225 个 Patch。结合图片的通道数，每个 Patch 包含 16×16×3 = 768 个值，把这 768 个值做 Flatten(拉平)作为 MLP 的输入，其中，MLP 的输出层神经元个数为 128。这样，每个 Patch 就可以得到长度的 128 的特征向量，组合得到 225×128 的 Table。MLP-Mixer 中 Patch 大小和 MLP 输出单元个数为超参数。

2. Mixer Layer

观察 Per-patch Fully-connected 得到的 Table 会发现(如上组合得到 225×128 的 Table)，Table 的行代表了同一空间位置在不同通道上的信息，列代表了不同空间位置在同一通道上的信息。换句话说，对 Table 的每一行进行操作可以实现通道域的信息融合，对 Table 每一列进行操作实现空间域的信息融合。

从 LN 出来的为 Patches×Channel(即为 Table)，每个 Patch 即上述对同一位置的所有通道(如 16×16×3)进行展开，通过 T 转为 Channel×Patches，token-mixing MLPs(MLP1)对 Table 的列进行映射，所以对 Channel×Patches 的行进行映射，对不同空间位置在同一通道上的信息进行操作，实现空间域的信息融合。channel-mixing MLPs(MLP2)对 Table 的行进行映射，对同一空间位置在不同通道上的信息进行映射，实现通道域的信息融合。

具体代码实现如下。

```
class MlpBlock(nn.Module):
  mlp_dim: int
  def __call__(self, x):
    y = nn.Dense(self.mlp_dim)(x)
    y = nn.gelu(y)
    return nn.Dense(x.shape[-1])(y)
class MixerBlock(nn.Module):
  """Mixer block layer."""
  tokens_mlp_dim: int
  channels_mlp_dim: int
  def __call__(self, x):
```

```
        y = nn.LayerNorm()(x)
        y = jnp.swapaxes(y, 1, 2)
        y = MlpBlock(self.tokens_mlp_dim, name = 'token_mixing')(y)
        y = jnp.swapaxes(y, 1, 2)
        x = x + y
        y = nn.LayerNorm()(x)
        return x + MlpBlock(self.channels_mlp_dim, name = 'channel_mixing')(y)
```

3.6 扩散模型

扩散模型(Diffusion Models,DMs)作为一类概率生成模型,主要由前向扩散、反向扩散以及采样三个阶段组成。如图 3-28 所示,在正向扩散阶段中通过逐渐添加高斯噪声的方式来对原始数据进行破坏。在反向扩散阶段,扩散模型的任务是通过学习逆转扩散过程,进而从噪声数据中恢复原始输入数据,以优化去噪网络。在采样阶段中,经过训练的去噪网络对噪声数据进行处理,以生成新的样本。

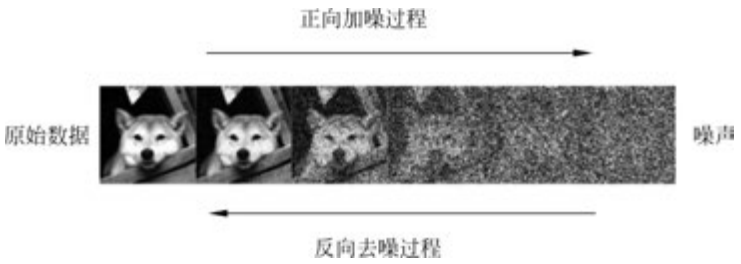


图 3-28 扩散模型处理流程

目前主流的生成模型有生成式对抗网络(Generative Adversarial Networks,GAN)、变分自编码器(Variational Autoencoder,VAE)与基于能量的模型(Energy Based Model,EBM)。与生成式对抗网络相比,扩散模型的训练过程使用了一种截然不同的策略,该策略包括用高斯噪声污染训练数据,然后学习从噪声数据中恢复原始数据,该训练方式更稳定,生成的样本质量更高,且此种训练方式能够解决生成对抗网络的局限性问题,如模式崩溃、对抗性学习的开销大和收敛失败等。变分自编码器将数据映射为潜在空间中的特征,但是降低了数据维度,反观扩散模型在正向扩散的过程中数据维度没有改变。基于能量的模型侧重于提供密度函数非归一化形式的估计值,也就导致较难进行训练。而扩散模型基于高斯模型进行训练,结果相对更加稳定。

目前对扩散模型的研究主要基于三种主要公式:去噪扩散概率模型(Denoising Diffusion Probabilistic Models,DDPMs)、基于评分的生成模型(Score-Based Generative Models,SGMs)与基于随机微分方程的扩散模型(Stochastic Differential Equations,SDEs)。接下来将逐一介绍上述三种扩散模型。

3.6.1 去噪概率扩散模型

去噪概率扩散模型[23]是目前扩散模型中使用最为广泛的一个分支,去噪概率扩散模型可分为以下三个过程:前向扩散过程、反向扩散过程、优化与采样过程。

1. 前向扩散过程

原始数据 x_0 采样于真实数据分布,前向扩散过程 $q(x_t \mid x_{t-1})$ 描述了向原始数据 x_0 添加高斯噪声 $\epsilon_t \sim \mathcal{N}(0, I)$ 的过程,前向扩散过程在两个时间步长 t 之间添加了少量的噪声。通

过逐渐添加越来越多的噪声，噪声样本 x_t 变得越来越嘈杂，最终通过多个时间步长 t，原始数据 x_0 完全被加噪为可处理的噪声数据 x_t。前向扩散过程中某一时刻 t 的样本数据只和上一时刻 $t-1$ 的有关，该过程可以视为马尔可夫过程。由于逐步添加的噪声一般服从高斯分布，所以前向扩散过程不具有任何可训练的参数。前向过程可表示为以下的马尔可夫过程。

$$q(x_t \mid x_{t-1}) := \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I), \forall t \in \{1,2,\cdots,T\} \tag{3-11}$$

其中，t 是时间步长；T 是时间步长的总数；$\{\beta_t \in (0,1)\}_{t=1}^{T}$ 作为噪声调度的一个超参数，用于控制每个时间步长中要添加的噪声量；I 为与输入的原始数据维度相同的单位矩阵。由此前向过程添加的噪声数据是服从高斯分布的，可以使用 VAE 中的重参数技巧[21]，对封闭的一段任意时间步长 t 的 x_t 进行采样。公式如下。

$$q(x_t \mid x_0) := \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I) \tag{3-12}$$

其中，$\bar{\alpha}_t = \prod_{s=1}^{t}\alpha_s$ 并且 $\alpha_t = 1-\beta_t$。α_t 与 β_t 作为噪声调度的超参数，可以互相表示，所以 α_t 与 β_t 在功能上是等效的。通过数学递推，可得到 x_t 的简化公式为

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\epsilon} \tag{3-13}$$

具体代码实现如下。

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
from dataloader import load_transformed_dataset, show_tensor_image
def linear_beta_schedule(timesteps, start = 0.0001, end = 0.02):
    return torch.linspace(start, end, timesteps)
def get_index_from_list(vals, time_step, x_shape):          #每个批次对应的时间索引 t
    batch_size = time_step.shape[0]
    out = vals.gather(-1, time_step.cpu())
    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(time_step.device)
#将图像和时间步长作为输入,然后返回噪声图片
def forward_diffusion_sample(x_0, time_step, device = "cpu"):
    noise = torch.randn_like(x_0)
    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, time_step, x_0.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(sqrt_one_minus_alphas_cumprod,
time_step,x_0.shape)
  #定义均值与方差
Return sqrt_alphas_cumprod_t.to(device * x_0.to(device) + sqrt_one_minus_alphas_cumprod_t.to
(device) * noise.to(device), noise.to(device)
T = 300
betas = linear_beta_schedule(timesteps = T)                 #定义 β 时间表
alphas = 1.0 - betas                                        #预先计算超参数(α 与 β)
alphas_cumprod = torch.cumprod(alphas, axis = 0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value = 1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1.0 - alphas_cumprod)
posterior_variance = betas * (1.0 - alphas_cumprod_prev) / (1.0 - alphas_cumprod)
if __name__ == "__main__":
    dataloader = load_transformed_dataset()                 #模拟前向扩散
    image = next(iter(dataloader))[0]
    plt.figure(figsize = (15, 15))
    plt.axis("off")
```

```
    num_images = 10
    stepsize = int(T / num_images)
    for idx in range(0, T, stepsize):
        time_step = torch.Tensor([idx]).type(torch.int64)
        plt.subplot(1, num_images + 1, int(idx / stepsize) + 1)
        img, noise = forward_diffusion_sample(image, time_step)
        show_tensor_image(img)
    plt.show()
```

2. 反向扩散过程

前向扩散过程是将原始数据加噪为噪声数据，反向过程就是一个去除噪声的过程。在此过程中，通过训练去噪网络递归地根据时间步长 t 逐步去噪。与 GAN 的区别在于，扩散模型是通过去噪网络来迭代去除两个连续时间步长之间的噪声，而不是在单个时间步长中去除所有噪声。反向扩散过程中，以高斯噪声 $x_T \sim N(0, I)$ 作为输入，从 $p(x_{t-1} | x_t)$ 中采样，推断并重构出真实样本。如果 β_t 足够小，$p(x_{t-1} | x_t)$ 的采样结果就仍为高斯分布，该值难以通过公式求解的方式来推断出原始的真实分布。解决方法便是将前向传播每步生成的真实噪声记录下来作为标签，并训练出一个神经网络 p_θ 来对这些噪声的条件概率进行预测。在模型做逆向扩散时，即可对前向扩散中所产生的高斯噪声进行预测，并一步一步推断，还原出最初始的样本数据。

反向过程与正向过程具有相同的迭代过程。在 DDPM 中，反向过程中的公式为

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}\left(x_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}}\epsilon_\theta(x_t, t)\right), \beta_t I\right) \tag{3-14}$$

其中，θ 是去噪网络的可训练参数，$p_\theta(x_{t-1} | x_t)$ 是反向扩散过程。具体代码实现如下。

```
import torch
from forward_noising import (
    get_index_from_list,
    sqrt_one_minus_alphas_cumprod,
    betas,
    posterior_variance,
    sqrt_recip_alphas)
import matplotlib.pyplot as plt
from dataloader import show_tensor_image
from unet import SimpleUnet
# 如果噪声并未全部去除，调用模型以预测图像中的噪声并返回去噪图像
def sample_timestep(model, x, t):
      betas_t = get_index_from_list(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(sqrt_one_minus_alphas_cumprod, t,
x.shape)
    sqrt_recip_alphas_t = get_index_from_list(sqrt_recip_alphas, t, x.shape)
    # 调用模型(预测当前图像的噪声)
    model_mean = sqrt_recip_alphas_t * (x - betas_t * model(x, t) / sqrt_one_minus_alphas_
cumprod_t)
    posterior_variance_t = get_index_from_list(posterior_variance, t, x.shape)
    if t == 0:
        return model_mean
    else:
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise
def sample_plot_image(model, device, img_size, T):
```

```
        img = torch.randn((1, 3, img_size, img_size), device = device)      #采样噪声
        plt.figure(figsize = (15, 15))
        plt.axis("off")
        num_images = 10
        stepsize = int(T / num_images)
        for i in reversed(range(0, T)):                                     #反向去噪迭代过程
            t = torch.tensor([i], device = device, dtype = torch.long)
            img = sample_timestep(model, img, t)
            img = torch.clamp(img, -1.0, 1.0)
            if i % stepsize == 0:
                plt.subplot(1, num_images, int(i / stepsize) + 1)
                show_tensor_image(img.detach().cpu())
        plt.savefig("sample.png")
    if __name__ == "__main__":
        img_size = 64
        T = 300
        model = SimpleUnet()
        device = "cuda" if torch.cuda.is_available() else "cpu"
        print(f"Using device: {device}")
    model.load_state_dict(torch.load("trained_models/ddpm_mse_epochs_500.pth"))
        model.to(device)
        sample_plot_image(model = model, device = device, img_size = img_size, T = T)
```

3. 优化与采样过程

在介绍完扩散模型的前向扩散过程与反向扩散过程后，接下来要考虑的便是如何一步步优化（通过模型学习）该问题。在此反向扩散过程中，中间产生的变量可以看成隐变量，则可将扩散模型看作包含若干隐变量的隐变量模型，该特征的映射过程与 VAE 相似，可以应用变分下限来优化负对数似然性。具体公式如下。

$$\begin{aligned} L = \mathbb{E}[& D_{\mathrm{KL}}(p(x_T \mid x_0) \parallel p(x_T)) + \\ & \sum_{t \geqslant 1} D_{\mathrm{KL}}(p(x_{t-1} \mid x_t, x_0) \parallel p_\theta(x_{t-1} \mid x_t)) - \\ & \log p_\theta(x_0 \mid x_1)] \end{aligned} \tag{3-15}$$

其中，$D_{\mathrm{KL}}(\cdot \parallel \cdot)$是 Kullback-Leibler（KL）散度，用于计算两个分布之间的差距。具体来说，目标 L 的最小化是为了减少 $p_\theta(x_0)$和 $q(x_0)$之间的差异。通过重参数化，可以在训练过程中忽略加权项简化扩散模型，以达到更好的效果。具体公式如下。

$$L = \mathbb{E}_{x_t, t}[\parallel \boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(x_t, t) \parallel_2^2] \tag{3-16}$$

采样过程利用优化的去噪网络来生成新的数据 x_0。具体地说，它首先从噪声数据 $p(x_T)$中获得样本 x_T，然后使用经过训练的网络通过 $p_{\theta^*}(x_{T-1} \mid x_T)$迭代的方式去除噪声。通过一系列的迭代过程，它最终生成了新的数据 $x_0^* \sim p_{\theta^*}(x_0) \approx p(x_0)$。具体公式如下。

$$\begin{aligned} p_{\theta^*}(x_0) &:= p(x_T) p_{\theta^*}(x_{T-1} \mid x_T) \cdots p_{\theta^*}(x_0 \mid x_1) \\ &= p(x_T) \prod_{t=1}^{T} p_{\theta^*}(x_{t-1} \mid x_t) \end{aligned} \tag{3-17}$$

其中，θ^*表示去噪网络的优化参数，$p(x_T)$是输入的噪声数据，$p_{\theta^*}(x_{T-1} \mid x_T)$表示逐步迭代去噪的过程。

采用 U-Net 为主干的整体网络代码如下。

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

```
import math
class Block(nn.Module):
    def __init__(self, in_ch, out_ch, time_emb_dim, up = False):
        super().__init__()
        self.time_mlp = nn.Linear(time_emb_dim, out_ch)
        if up:
            self.conv1 = nn.Conv2d(2 * in_ch, out_ch, 3, padding = 1)
            self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding = 1)
            self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding = 1)
        self.bnorm1 = nn.BatchNorm2d(out_ch)
        self.bnorm2 = nn.BatchNorm2d(out_ch)
        self.relu = nn.ReLU()
    def forward(self,x t):
        h = self.bnorm1(self.relu(self.conv1(x)))
        time_emb = self.relu(self.time_mlp(t))        #时间嵌入
        time_emb = time_emb[(...,) + (None,) * 2]  #扩展最后两个维度
        h = h + time_emb                              #添加时间通道
        h = self.bnorm2(self.relu(self.conv2(h)))
        return self.transform(h)                      #进行上采样或下采样操作
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim
    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device = device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim = -1)
        return embeddings
class SimpleUnet(nn.Module):
    def __init__(self):
        super().__init__()
        image_channels = 3
        down_channels = (64, 128, 256, 512, 1024)
        up_channels = (1024, 512, 256, 128, 64)
        out_dim = 3
        time_emb_dim = 32
      self.time_mlp = nn.Sequential(SinusoidalPositionEmbeddings(time_emb_dim), nn.Linear
(time_emb_dim, time_emb_dim), nn.ReLU())
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding = 1)
        self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i + 1], time_emb_dim)
        for i in range(len(down_channels) - 1)]
            self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i + 1], time_emb_
dim,up = True)
        for i in range(len(up_channels) - 1)])
        self.output = nn.Conv2d(up_channels[-1], out_dim, 1)
    def forward(self, x, timestep):
        t = self.time_mlp(timestep)           #时间嵌入
        x = self.conv0(x)                     #初始卷积
        residual_inputs = [ ]                 #U-Net
        for down in self.downs:
```

```
            x = down(x, t)
            residual_inputs.append(x)
        for up in self.ups:
            residual_x = residual_inputs.pop()
            # Add residual x as additional channels
            x = torch.cat((x, residual_x), dim = 1)
            x = up(x, t)
        return self.output(x)
```

3.6.2 基于分数的生成模型

基于分数的生成模型[24]的主要目标是通过训练一个对抗噪声干扰的深度神经网络，估计每个噪声分布的数据分布相关的梯度（即分数）并使用朗之万动力学（Langevin Dynamics）的方法从估计的数据分布中进行采样来生成新的样本。分数网络 s_θ 是一个参数为 θ 的神经网络，它被训练为通过最小化以下目标来近似 $p(x)$ 的所得分数 $s_\theta(x)\approx\nabla_x \log\ p(x)$：

$$\mathbb{E}_{x\sim p(x)} \parallel s_\theta(x)-\nabla_x \log p(x) \parallel_2^2 \tag{3-18}$$

为了避免分布塌陷到低维流形和低密度区域中不准确的分数估计，将退火的朗之万动力学运用于基于分数的生成模型，其中引入单调递减的预定义噪声 $\sigma_{i=1}^{L}$ 来扰动数据。朗之万动力学的原始采样过程可以表示为

$$\tilde{x}_t=\tilde{x}_{t-1}+\frac{\epsilon}{2}\nabla_{\tilde{x}}\log p(\tilde{x}_{t-1})+\sqrt{\epsilon}z_t \tag{3-19}$$

其中，z_t 是时间步长 t 处的随机正态高斯噪声，ϵ 代表固定步长。当时间步长 $T\to\infty$ 和 $\epsilon\to 0$，分布 $p(\tilde{x}_T)$ 将等于原始数据分布 $p(x)$。在添加分布均值为 σ 的噪声之后，噪声分布为 $q_\sigma(\tilde{x})\triangleq\int p(x)\mathcal{N}(\tilde{x}\mid x,\sigma^2 I)\mathrm{d}x$。基于分数的扩散模型可以通过 $s_\theta(\tilde{x},\sigma)=-\nabla_{\tilde{x}}\log q_\sigma(\tilde{x})$ 来优化网络，噪声得分匹配目标的过程如下。

$$\mathcal{L}(\theta,\sigma)=\frac{1}{2}E_{p(x)}E_{\tilde{x}\sim\mathcal{N}(x,\sigma^2 I)}\left[\left\|s_\theta(\tilde{x},\sigma)+\frac{\tilde{x}-x}{\sigma^2}\right\|_2^2\right] \tag{3-20}$$

3.6.3 基于随机微分方程的扩散模型

随机微分方程是一种用来描述在不确定性和随机力量的作用下，系统随时间如何变化的数学方程。基于随机微分方程的扩散模型[26-28]的前向过程添加了连续时间步长的噪声，具体公式如下。

$$\mathrm{d}x=f(x,t)\mathrm{d}t+g(t)\mathrm{d}w \tag{3-21}$$

其中，$f(x,t)$ 和 $g(t)$ 分别为随机微分方程的漂移系数和扩散系数。具体来说，漂移系数的作用逐渐将原始样本数据进行噪声化处理，而扩散系数主要是衡量所添加噪声量的多少。w 表示标准的布朗运动，$\mathrm{d}w$ 可以视为无穷小的白噪声。基于随机微分方程的扩散模型（SDE）的反向扩散过程具体公式如下。

$$\mathrm{d}x=[f(x,t)-g^2(t)s_\theta(x_t,t)]\mathrm{d}t+g(t)\mathrm{d}w \tag{3-22}$$

其中，$\mathrm{d}t$ 为负无穷小的时间步长，$\mathrm{d}w$ 表示反向扩散过程中的布朗运动，其中 $s_\theta(x_t,t)=\nabla_x\log p(x_t)$，当 $g^2(t)s_\theta(x_t,t)$ 无限趋向于 $f(x,t)$ 时，就可实现逆向求解。为了数值求解反向过程的 SDE，可以训练神经网络通过分数匹配来近似实际分数函数。该分数模型使用以下目标进行训练，具体公式如下。

$$L=\mathbb{E}_t[\lambda(t)\,\mathbb{E}_{x_0}\mathbb{E}_{x_t|x_0}[\,\|s_\theta(x_t,t)-\nabla_x\log p(x_t\mid x_0)\|_2^2]] \tag{3-23}$$

其中，$\lambda(t)$是加权函数。用$\nabla_x\log p(x_t)$代替$\nabla_x\log p(x_t\mid x_0)$可以避免技术上的难题。最后，采样过程获得$x_t$，并应用训练后的网络$\theta^*$生成新的数据。

3.7 图神经网络

随着人工智能的飞速发展，深度学习已经在图像、文本和语音等可在欧氏空间表示的数据中取得了巨大成功，但却一直无法很好地应用于非欧氏空间。图神经网络(Graph Neural Networks，GNN)最早是由 Gori 等[29]于 2005 年提出的，并经过 Scarelli 等[30]的工作得到了进一步的发展和完善。而图神经网络能够很好地学习到非欧氏空间中的特征表示，因此受到了诸多学者的关注。有所区别的是，它处理的对象是图形数据，这使得图神经网络在非欧氏空间中展现出了强大的表示学习能力，并广泛应用于推荐系统、自然语言处理以及机器视觉等众多领域。图神经网络的原理基于消息传递和节点更新的思想，每个节点将周围节点的信息进行聚合和传递，以更新自身的表征向量。具体来说，图神经网络通过定义节点聚合函数和更新函数来控制信息的传递和更新过程，使节点能够从邻居节点中获得信息并更新自身的表征。这个过程在整个图中迭代多次，直到模型达到收敛。通过消息传递和节点更新的操作，图神经网络可以聚合邻居节点特征并使用特征去对图数据进行节点分类、图分类、边预测，还可以顺便得到图的嵌入表示。

接下来将介绍三种经典的基于图数据的神经网络架构：基于谱域的图卷积神经网络(spectral-based Graph Convolutional Networks，spectral-based GCN)、基于空域的图卷积神经网络(spatial-based Graph Convolutional Networks，spatial-based GCN)、GraphSAGE(Graph Sample and Aggregate)与图注意力网络(Graph Attention Networks，GAT)。

3.7.1 基于谱域的图卷积神经网络

基于谱域的图卷积神经网络是图卷积神经网络的一个重要分支，它从图信号处理的角度引入滤波器来定义图卷积。具体而言，它首先把图上的信号变换到谱域(频率域)，然后在谱域上定义卷积操作，最后再把卷积后的结果变换回空域。基于谱域的图卷积神经网络的核心在于对节点特征进行傅里叶变换，生成以拉普拉斯矩阵的特征向量为基的谱图，然后依据卷积定理求得卷积核对特征进行卷积后提取的特征。拉普拉斯矩阵反映了图的结构信息，而特征向量则代表了图中各个节点的特性。通过这种方式，基于谱域的图卷积神经网络能够捕获到图的全局结构信息，从而得到更准确的节点表示。2017 年，Thomas N. Kipf 等[30]提出图神经网络领域的“开山之作”GCN，它首次将图像处理中的卷积操作简单地用到图结构数据处理中来。接下来将详细介绍 GCN 的图卷积过程：给定一批图数据，其中有 N 个具有不同特征的节点(node)，这些节点的特征组成一个 $N\times D$ 维的矩阵 $\boldsymbol{X}$，各个节点之间的关系也会形成一个 $N\times N$ 维的邻接矩阵 $\boldsymbol{A}$，$\boldsymbol{X}$ 和 $\boldsymbol{A}$ 是模型的输入。GCN 也是一个神经网络，层与层之间的传播方式如下所示。

$$H^{(l+1)}=\sigma(\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{3-24}$$

其中，$\widetilde{\boldsymbol{A}}=\boldsymbol{A}+\boldsymbol{I}$，$\boldsymbol{I}$ 是单位矩阵，$\widetilde{\boldsymbol{D}}$ 是$\widetilde{\boldsymbol{A}}$ 的度矩阵，公式为 $\widetilde{D}_{ii}=\sum j\cdot\widetilde{\widetilde{A}}_{ij}$，$H$ 是每一层的特征，l 表示特征层数；$l=1$ 时，$\boldsymbol{H}=\boldsymbol{X}$，$\sigma$ 是非线性激活函数。

如图 3-29 所示，把一个图作为输入，通过若干层 GCN 每个节点的特征从 $\boldsymbol{X}$ 变成了 $\boldsymbol{Z}$。假设构造一个两层的 GCN，激活函数分别采用 ReLU 和 Softmax，则整体的正向传播的公式为

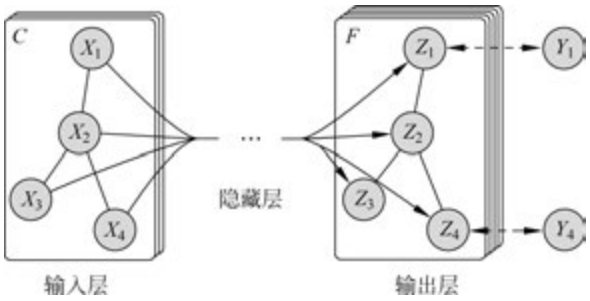


图 3-29　Thomas N. Kipf 等提出的 GCN

$$\boldsymbol{Z}=f(\boldsymbol{X},\boldsymbol{A})=\text{softmax}(\hat{\boldsymbol{A}}\text{ReLU}(\hat{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}^{(0)})\boldsymbol{W}^{(1)}) \tag{3-25}$$

针对所有带标签的节点计算 cross entropy 损失函数，具体公式如下。

$$\mathcal{L}=-\sum_{l\in y_L}\sum_{f=1}^{F}Y_{lf}\ln Z_{lf} \tag{3-26}$$

通过损失函数的变换，GCN 可以执行其他不同的任务，如图分类、边预测等。此外，GCN 在少量带标签的节点下也能够训练，具有半监督分类的特点。图 3-30 展示了 GCN 在俱乐部关系网络数据集上初始化节点的可视化结果，由图 3-30(b)可见，该方法在不经过训练下提取到的初始特征已经具备良好的聚类结果，而这种特性非常利于神经网络的训练，证明了其在深度学习中的潜力。

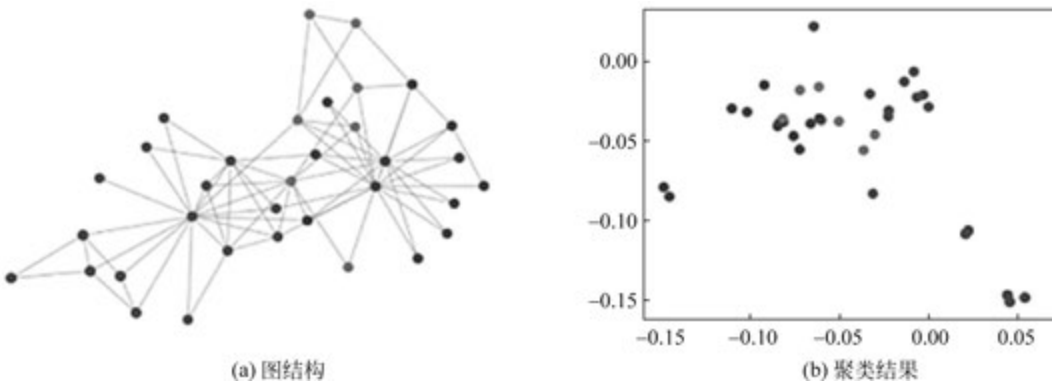


图 3-30　俱乐部关系聚类结果

具体代码实现如下。

```
class GraphConvolution(nn.Module):                    #定义图卷积
    def __init__(self, input_dim, output_dim, use_bias = True):
        super(GraphConvolution, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.use_bias = use_bias
        self.weight = nn.Parameter(torch.Tensor(input_dim, output_dim))
        if self.use_bias:
            self.bias = nn.Parameter(torch.Tensor(output_dim))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()
    def reset_parameters(self):
```

```
            init.kaiming_uniform_(self.weight)
            if self.use_bias:
                init.zeros_(self.bias)
    # 邻接矩阵是稀疏矩阵,因此在计算时使用稀疏矩阵乘法
        def forward(self, adjacency, input_feature):
            support = torch.mm(input_feature, self.weight)
            output = torch.sparse.mm(adjacency, support)
            if self.use_bias:
                output += self.bias
            return output
        def __repr__(self):
            return self.__class__.__name__ + '('+ str(self.input_dim) + ' -> ' + str(self.
    output_dim) + ')'
    class GcnNet(nn.Module):                    # 定义一个包含两层 GraphConvolution 的模型
        def __init__(self, input_dim = 1433):
            super(GcnNet, self).__init__()
            self.gcn1 = GraphConvolution(input_dim, 16)
            self.gcn2 = GraphConvolution(16, 7)
         def forward(self, adjacency, feature):
            h = F.relu(self.gcn1(adjacency, feature))
            logits = self.gcn2(adjacency, h)
            return logits
```

3.7.2　基于空域的图卷积神经网络

基于空域的图卷积神经网络[32]是一种处理图数据的深度学习模型。与卷积神经网络处理图像数据类似，图卷积神经网络旨在从图中提取有用的特征表示。但是，由于图数据的非欧几里得性质，即节点之间的连接不规则，传统的卷积操作无法直接应用于图数据。因此，基于空域的图卷积神经网络设计了一种特殊的卷积操作来适应图结构。基于空域的图卷积神经网络主要关注于直接在图的节点及其邻居上进行卷积操作。其核心思想是通过聚合邻居节点的信息来更新当前节点的表示。这通常涉及以下几个步骤。

（1）初始化节点表示：每个节点被赋予一个初始的特征向量，这个向量可以包含节点的属性信息。

（2）聚合邻居信息：对于每个节点，它从其邻居节点收集信息。这通常是通过将邻居节点的特征向量与某种形式的权重或系数相结合来完成的，这些权重或系数可能基于节点间的连接强度、节点的度或其他属性。

（3）更新节点表示：基于聚合的邻居信息，当前节点的表示被更新。这通常涉及将聚合的信息与当前节点的原始表示进行某种形式的组合，如加权求和或非线性变换。

（4）堆叠多层：为了捕获图的复杂模式，可以堆叠多个图卷积层，每一层都进一步聚合和转换节点的表示。

然而，基于空域的图卷积神经网络也面临一些挑战，例如，如何有效地处理大规模图、如何设计有效的聚合函数，以及如何处理图的动态变化等。目前，基于空域的图卷积神经网络已经在许多领域取得了成功的应用，包括社交网络分析、推荐系统、生物信息学、自然语言处理等。随着深度学习技术的不断发展，基于空域的图卷积神经网络将继续发挥作用，为处理和分析复杂图数据提供强大的工具。

3.7.3 GraphSAGE

图数据和其他类型数据的不同，图数据中的每一个节点可以通过边的关系利用其他节点的信息。GCN 在训练节点收集邻居节点信息的时候，用到了测试和验证集的样本，导致无法快速得到新节点的嵌入，这种学习方式称为直推式学习（Transductive Learning）。然而，我们所处理的大多数机器学习问题都是归纳式学习（Inductive Learning），可以将样本集分为训练/验证/测试，并且训练时只用训练样本。为了解决上述问题，斯坦福大学于 2017 年提出一种基于图的归纳学习方法——GraphSAGE（Graph Sample and Aggregate）[34]。

具体实现中，GraphSAGE 的训练仅保留训练样本之间的边，然后包含采样（Sample）和聚合（Aggregate）两大步骤。采样是指对邻居特征信息进行采样，聚合是指拿到邻居节点的特征嵌入之后，汇聚这些嵌入信息以更新自己的嵌入信息。GraphSAGE 基于 GCN 改进而来，GCN 的迭代公式如下。

$$H^{(l+1)}=\sigma(\boxed{\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-\frac{1}{2}}}\,H^{(l)}W^{(l)}) \tag{3-27}$$

式（3-27）中黑框位置所做的操作可以简单地理解为对邻接矩阵 $\boldsymbol{A}$ 的归一化变换，去掉该部分会发现剩下的结构等同于深度神经网络，加上黑框框选的部分后，通过矩阵乘法实际上所做的就是将节点与节点相邻节点的特征信息进行相加。GraphSAGE 在特征聚合方式上与 GCN 的简单相加不同，GraphSAGE 支持 max-pooling、LSTM、mean 等聚合方式。

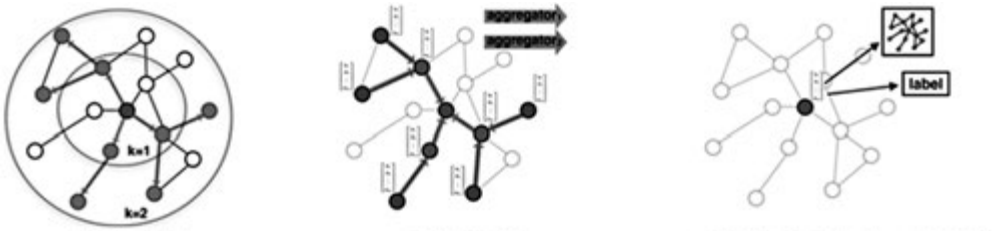
GraphSAGE 传播方法的伪代码如下。

```
算法 1：GraphSAGE 节点嵌入生成算法
输入：图 G(V,E)，输入特征{x_v, ∀ v∈V}，深度 K，V，E 分别代表图的节点与边
权重矩阵 W_k, ∀ k∈{1,2,…,K}，非线性激活函数σ，
可区分的聚合函数 AGGREGATEk, ∀ k∈{1,2,…,K}；邻居函数 N: v→2^V
输出：对每一个节点 v∈V，其向量表示为 z_v
1: h_v^0 ←x_v, ∀ v∈V
2: for k = 1...K do
3:  for v∈V do
4:  h_N(v)^k ←AGGREGATE_k({h_u^(k-1), ∀ u∈N(v)});
5:  h_N(v)^k ←σ(W^k · CONCAT(h_v^(k-1), h_N(v)^k))
6:  end
7:  h_v^k ←h_v^k/‖h_v^k‖_2, ∀ v∈V
8: end
9: z_v ←h_v^K, ∀ v∈V
```

对于图 G 中的某个节点 v，首先进行对层数遍历的 for 循环，以聚合 k 层信息，其次进行对邻居节点 v 遍历的 for 循环来获得邻居特征信息，再通过聚合函数如 AGGREGATE、mean、max、LSTM 等来聚合 $k-1$ 层的邻居节点信息，得到聚合后的 k 层邻居节点信息，然后将聚合后的 k 层邻居节点信息与 $k-1$ 层节点 v 的信息进行拼接，然后通过权重参数 W 计算得到 k 层关于节点 v 的信息。

如图 3-31 所示，以为中心红色节点为目标节点，在一次步骤中，对中心红色节点的一阶邻居和二阶段邻居做随机采样。然后通过聚合策略，把节点的特征信息从二阶邻居聚合到目标节点上，然后用更新后的目标节点的表征可以应用到不同需求的任务上。

综上，GraphSAGE 与 GCN 的核心思想以及各自的优劣势如表 3-11 所示。



(a) 采样邻居节点　(b) 聚合邻居节点　(c) 通过聚合信息预测图上下文和标签

图 3-31 图节点的传播方式

表 3-11 GraphSAGE 与 GCN 的对比

模型	基本思想	优势	劣势
GCN	把一个节点在图中的高维邻接信息降维到一个低维的向量表示	可以捕捉到图的全局信息，从而可以更好地表示节点	是 Transductive Learning(直推学习)，训练的时候会用到验证集、测试集的信息，必须要把全部的节点参与训练才能获得节点 embedding，对于新节点也需要重新训练，才可以产生节点 embedding，从而影响了产出的速率
GraphSAGE	利用节点特征信息和结构信息，从顶点的局部邻居采样并聚合邻居节点和顶点的特征，获取到顶点的 Graph Embedding	可以利用已有的节点信息产生新节点的 embedding，同时可以保证生产的高效性可以应用在大规模的图学习上	聚合计算的时候没有考虑到邻居的重要性(GAT 模型进行了相关的完善)，只涉及了无向图

具体代码实现如下。

```
class NeighborAggregator(nn.Module):                          #聚合节点邻居
    def __init__(self, input_dim, output_dim, use_bias=False, aggr_method="mean"):
        super(NeighborAggregator, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.use_bias = use_bias
        self.aggr_method = aggr_method
        self.weight = nn.Parameter(torch.Tensor(input_dim, output_dim))
        if self.use_bias:
            self.bias = nn.Parameter(torch.Tensor(self.output_dim))
        self.reset_parameters()
     def reset_parameters(self):
        init.kaiming_uniform_(self.weight)
        if self.use_bias:
            init.zeros_(self.bias)
    def forward(self, neighbor_feature):                      #邻居特征聚合方法
        if self.aggr_method == "mean":
            aggr_neighbor = neighbor_feature.mean(dim=1)
        elif self.aggr_method == "sum":
            aggr_neighbor = neighbor_feature.sum(dim=1)
        elif self.aggr_method == "max":
            aggr_neighbor = neighbor_feature.max(dim=1)
        else:
            raise ValueError(.format(self.aggr_method))  #节点特征的更新方法
```

```
            neighbor_hidden = torch.matmul(aggr_neighbor, self.weight)
        if self.use_bias:
            neighbor_hidden += self.bias
        return neighbor_hidden
    def extra_repr(self):
        return 'in_features = {}, out_features = {}, aggr_method = {}'.format(self.input_dim,
self.output_dim, self.aggr_method)
class SageGCN(nn.Module):                    #定义 GCNSage 层
    def __init__(self, input_dim, hidden_dim,activation = F.relu, aggr_neighbor_method =
"mean",
                 aggr_hidden_method = "sum"):
        super(SageGCN, self).__init__()
        assert aggr_neighbor_method in ["mean", "sum", "max"]
        assert aggr_hidden_method in ["sum", "concat"]
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.aggr_neighbor_method = aggr_neighbor_method
        self.aggr_hidden_method = aggr_hidden_method
        self.activation = activation
```

3.7.4 图注意力网络

图注意力网络[35]是一种将注意力机制引入基于空间域的图神经网络中的模型。它通过计算节点之间的相似度，为每个节点分配一个权重，从而实现对邻居节点的加权聚合。这种机制有助于捕捉图中节点之间的关系，提高节点分类和图分类的性能。在图注意力网络中，节点的特征表示和普通的图神经网络中的节点特征表示类似，都是采用节点嵌入的方式对节点的特征表示进行向量化。网络的输入即为这些节点的特征组合。单头图注意力机制结构如图 3-32 所示，给定两个节点 i、j 的特征作为输出，计算两节点之间的注意力权重。

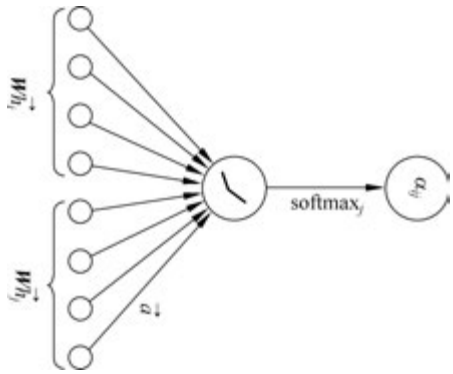


图 3-32　单头图注意力机制结构

节点 i、j 的注意力系数（Attention Coefficients）计算方式为

$$e_{ij} = a(\vec{Wh}_i, \vec{Wh}_j) \tag{3-28}$$

其中，$\boldsymbol{W}$ 是对于各节点进行特征增维的矩阵，h 就是节点的特征，$a(\boldsymbol{W},\boldsymbol{W})$可以表示两个向量的内积，即向量相似度。经过 Softmax 得到注意力权重：

$$\alpha_{ij} = \mathrm{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \tag{3-29}$$

α_{ij} 的注意力权重计算公式如下。

$$\alpha_{ij} = \frac{\exp(\mathrm{LeakyReLU}(\vec{a}^{\mathrm{T}}[\vec{Wh}_i \,\|\, \vec{Wh}_j]))}{\sum_{k \in \mathcal{N}_i} \exp(\mathrm{LeakyReLU}(\vec{a}^{\mathrm{T}}[\vec{Wh}_i \,\|\, \vec{Wh}_k]))} \tag{3-30}$$

其中，$\mathcal{N}_i$ 表示节点 i 的邻居节点，$\|$ 表示特征拼接。基于注意力权重聚合邻居信息后的节点特征的公式如下。

$$\vec{h}'_i = \Big\Vert \sigma\Big(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \boldsymbol{W}^k \vec{h}_j\Big) \tag{3-31}$$

多头注意力本质是引入并行的几个独立的注意力机制，多头注意力机制可以提取信息中的多重含义，防止过拟合。结构如图 3-33 所示，公式如下。

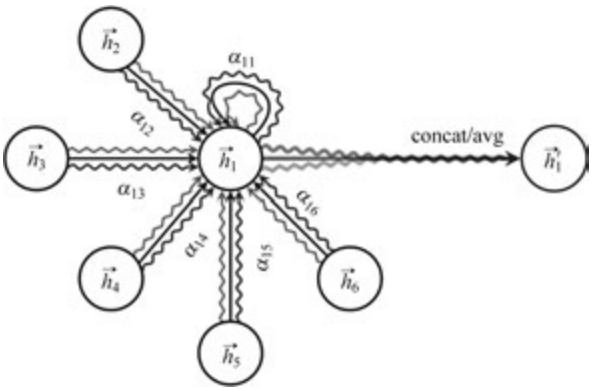


图 3-33　多头图注意力机制结构

$$\vec{h}'_i = \sigma\Big(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \boldsymbol{W}^k \vec{h}_j\Big) \tag{3-32}$$

与传统的基于谱域的图卷积神经网络相比，图注意力网络不需要使用拉普拉斯等矩阵进行复杂的计算，仅是通过一阶邻居节点的表征来更新节点特征，因此算法原理从理解上较为简单。具体代码实现如下。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class GraphAttentionLayer(nn.Module):
#定义图卷积层
    def __init__(self, in_features, out_features, dropout, alpha, concat = True):
        super(GraphAttentionLayer, self).__init__()
        self.dropout = dropout
        self.in_features = in_features
        self.out_features = out_features
        self.alpha = alpha
        self.concat = concat
        self.W = nn.Parameter(torch.zeros(size = (in_features, out_features)))
        nn.init.xavier_uniform_(self.W.data, gain = 1.414)
        self.Q = nn.Parameter(torch.zeros(size = (in_features, out_features)))
        nn.init.xavier_uniform_(self.Q.data, gain = 1.414)
        self.V = nn.Parameter(torch.zeros(size = (in_features, out_features)))
        nn.init.xavier_uniform_(self.V.data, gain = 1.414)
        self.a = nn.Parameter(torch.zeros(size = (2 * out_features, 1)))
        nn.init.xavier_uniform_(self.a.data, gain = 1.414)
        self.leakyrelu = nn.LeakyReLU(self.alpha)
    def forward(self, input, adj):
#图注意力机制
        h = torch.mm(input, self.W)
        q = torch.mm(input, self.Q)
        v = torch.mm(input, self.V)
        N = h.size()[0]
        a_input = torch.cat([h.repeat(1, N).view(N * N, -1), q.repeat(N, 1)], dim = 1).view
(N, -1, 2 * self.out_features)
```

```
        e = self.leakyrelu(torch.matmul(a_input, self.a).squeeze(2))
        zero_vec = -9e15 * torch.ones_like(e)
        attention = torch.where(adj > 0, e, zero_vec)
        attention = F.softmax(attention, dim=1)
        attention = F.dropout(attention, self.dropout, training=self.training)
        h_prime = torch.matmul(attention, v)
        if self.concat:
            return F.elu(h_prime)
        else:
            return h_prime
    def __repr__(self):
        return self.__class__.__name__ + '(' + str(self.in_features) + ' -> ' + str(self.out_
features) + ')'
class GAT(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout, alpha, nheads):
        super(GAT, self).__init__()
#稠密图注意力机制
        self.dropout = dropout
        self.attentions = [GraphAttentionLayer(nfeat, nhid, dropout=dropout, alpha=alpha,
concat=True) for _ in range(nheads)]
        for i, attention in enumerate(self.attentions):
            self.add_module('attention_{}'.format(i), attention)
        self.out_att = GraphAttentionLayer(nhid * nheads, nclass, dropout=dropout, alpha=
alpha,
                           concat=False)
    def forward(self, x, adj):
        x = F.dropout(x, self.dropout, training=self.training)
        x = torch.cat([att(x, adj) for att in self.attentions], dim=1)
        x = F.dropout(x, self.dropout, training=self.training)
        x = F.elu(self.out_att(x, adj))
        return F.log_softmax(x, dim=1)
```

3.8 神经架构搜索

近年来，深度神经网络应用到图像识别、语音识别、目标检测、机器翻译等领域，加速了网络的性能演进与灵活性提升。但这些网络通常结构复杂，需要拥有大量专业知识的人员消耗大量时间调整参数以匹配具体环境。这样通过人工来调整参数的常规方法效率较低且错误频出。为了解决这一问题，Enzo 等[36]在 ICLR 2017 提出神经架构搜索（Neural Architecture Search，NAS）这一概念。神经架构搜索是一种自动设计神经网络的技术，可以通过算法根据样本集自动设计出高性能的网络结构，在某些任务上甚至可以媲美人类专家的水准，甚至发现某些人类之前未曾提出的网络结构，神经架构搜索可以有效地降低神经网络的构建成本。从结构上可以将 NAS 划分为三个组成部分：搜索空间（Search Space）、搜索策略（Search Strategy）和性能评估策略（Performance Estimation Strategy）。

如图 3-34 所示，NAS 的具体流程包括以下步骤。

（1）定义一个搜索空间：搜索空间是 NAS 算法可以搜索的神经网络结构的集合，它定义了 NAS 可以探索的网络架构的范围和类型。

（2）选择搜索策略：使用特定的搜索策略在搜索空间中寻找性能最优的网络架构。

（3）定义性能评估标准：为了衡量不同网络架构的性能，需要定义一个或多个评估标准。

（4）评估候选架构：对于搜索策略生成的每个候选网络架构，使用性能评估标准进行评估。

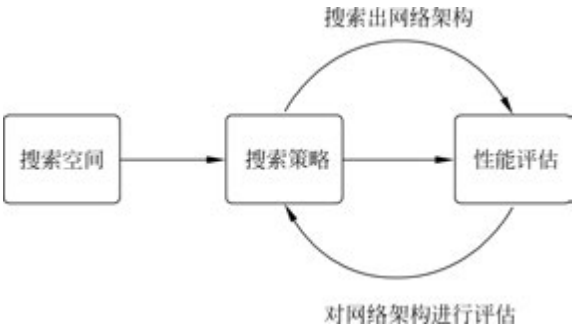


图 3-34　NAS 的整体流程

(5) 迭代优化：根据评估结果，选择性能最好的候选架构作为当前最优解，并继续搜索以寻找更好的架构。

(6) 导出最优架构：在搜索过程结束后，导出性能最优的神经网络架构。这个架构可以作为新任务的起始点，也可以作为其他网络设计的参考。

3.8.1　搜索空间

搜索空间定义了所有可以表达出的架构种类，可以通过结合具体任务的具体空间特征的方法缩小搜索空间的大小、减少 NAS 的时间开销，但往往会限制搜索出的架构的多样性。目前使用的搜索空间主要包括三种：链式搜索空间、基于 Cell/Block 类单元的搜索空间和分层搜索空间。

1. 链式搜索空间

链式搜索空间是一种相对简单的搜索空间。一般地，一个链式搜索空间可以用一个 n 层的序列来表示，其中第 i 层的输出会成为第 $i+1$ 层的输入，如图 3-35 所示。该类型的搜索空间存在三种参数：最大层数 n；每层可执行的操作类型，如池化、卷积等；操作所对应的超参数，如卷积核的尺寸、步长、通道数等。在链式搜索空间的基础上也可以仿照残差网络(ResNet)[37]和稠密连接网络(DenseNet)[38]加入一些复杂的要素，如多分支结构和跳跃连接。总之，链式搜索空间相对简单，搜索出的架构呈线性结构，搜索范围大但搜索效率较低，这种搜索空间往往会采用加入一些其他网络的复杂元素的方法来提高搜索效率。

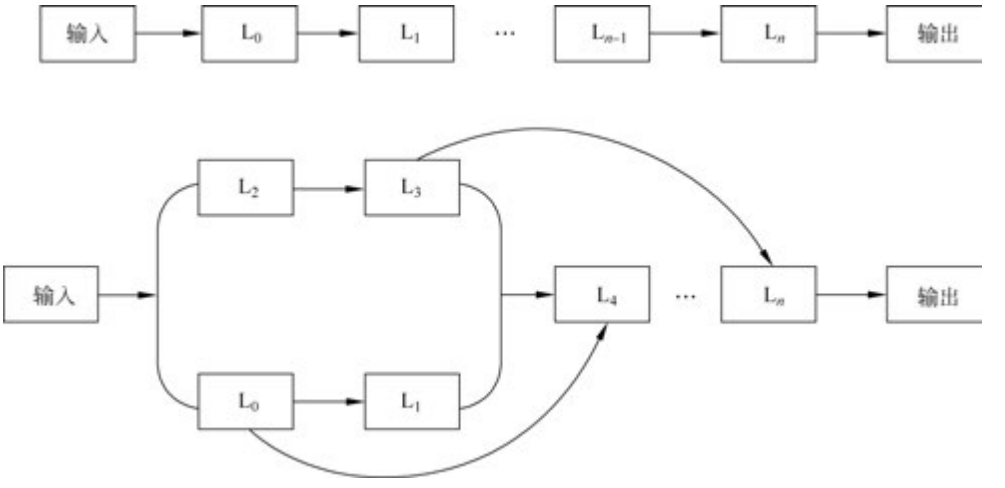


图 3-35　链式搜索空间结构

2. 基于 Cell/Block 类单元的搜索空间

考虑到研究学者设计的神经网络模型中常有重复功能块存在。Zoph[39]等于 2018 年提出基于 Cell 类单元的搜索空间。具体来说，他们提出了两种单元：普通单元，用来保留输入维度；约简单元，用来缩小空间维度。首先，通过对这些单元进行搜索来替代搜索整个网络结构以提升搜索效率；其次，通过提前定义好的方式对不同的优化好的单元进行拼接，得到目标神经网络架构，如图 3-36 所示。与之前的方法相比，使用基于 Cell 类单元的搜索空间存在很多优点：调整不同单元的数量，使单元可以适应不同数据集的特点，便于数据的迁移；NAS 的目标不再是整个网络，而是单元，大幅降低了搜索空间的大小，提升了效率。但同时也存在两个问题：无法保证训练过的单元的优异性能在经过堆叠后依然优秀，针对该问题，文献[40]给出了解决方案；搜索空间的狭小会导致搜索到的结构被限制，可能会降低搜索到真正新颖结构的可能性，即可能会减少搜索结构的多样性。这种方法也成为此后搜索空间的常见主流方法。

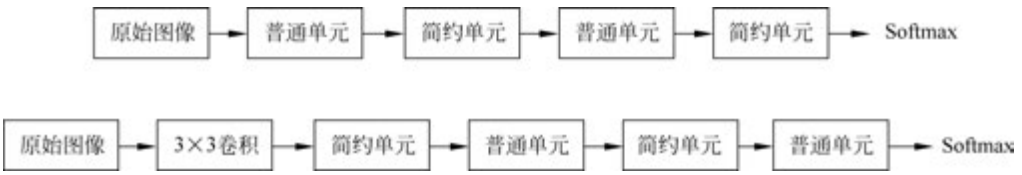


图 3-36　基于 Cell/Block 类单元的搜索空间结构

3. 分层搜索空间

分层搜索空间又叫层次化搜索空间，是由 LIU[40]等于 2018 年提出的方法，它的提出主要是出于缩小搜索空间，提高搜索效率的目的。分层搜索空间所用到的是一种叫作分层表示（Hierarchical Representation）的方法，这种方法的核心就是将搜索空间分为 N 层，每一层都包括多个基元（Motifs）。具体来说，分层搜索空间将搜索空间分为多层，每层由一个或多个计算图组成，计算图的结构类似于扁平架构，扁平架构有单一的源节点和终点节点，其中的每一个节点都是特征图，有向边则表示一些基础操作，如池化、卷积操作等，计算图接收一个特征图作为输入，对其进行一系列的基础操作后进行输出。本文定义基础操作集合 $O=\{o_1,o_2,\cdots,o_n\}$，邻接矩阵元素 $G_{ij}=k$ 表示一条有向边由 i 指向 j，其对应的操作为 o_{ij}，基于此搜索网络架构的实质是选择基础操作集合 O 中的操作并组合到 $\boldsymbol{G}$ 上。

3.8.2　搜索策略

搜索策略的目标便是使用效果最好的搜索算法，对网络结构进行搜索，高效、准确地找到网络结构的最优组合，并对涉及的超参数进行优化，以便搜索最佳的网络结构。目前研究的搜索策略主要是基于强化学习的策略、基于进化算法的策略以及基于梯度的策略。

1. 基于强化学习的搜索策略

最早被用于 NAS 的搜索方法是强化学习（Reinforcement Learning）。基于强化学习的搜索策略[41]主要采取“尝试”的学习机制进行学习，通过于环境交流得到反馈后指导下一步学习行为。强化学习主体由智能体和环境两部分组成，如图 3-37 所示，智能体同时充当着学习者和决策者的角色，通过和环境交流实现目标。在某一时刻，智能体基于当前状态发出动作、环境做出反应，生成新的状态和新的奖励值，通过循环执行，获取最大的累计奖励值。

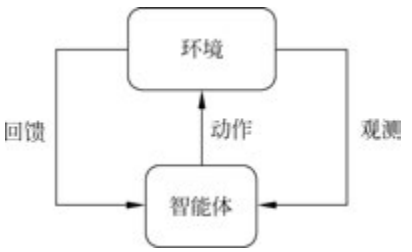


图 3-37　强化学习算法

2. 基于进化算法的搜索策略

进化算法是一种非常古老的方法，它受生物种群整体进化的启发，通过选择、基因重组和变异这三种基本操作来解决优化问题，如图 3-38 所示。Google 公司在 2017 年的论文 *Large-Scale Evolution of Image Classifiers*[42] 中首次将进化算法应用于神经网络自动任务，并在图像分类任务中取得了显著的成果。该方法首先将网络的结构进行编码，并维护一个结构的集合(种群)。通过选择种群中的最佳结构进行训练和评估，只保留性能高的网络，淘汰性能低的网络。通过预先定义的变异操作产生新的候选方案，在训练和评估后将其加入种群中，并在迭代过程中进行优化，直到达到终止条件(例如，达到最大迭代次数或变异后网络性能不再提高)。随后，针对该方法的改进论文 *Regulated Evolution for Image Classifier Architecture Search*[43] 进行了研究，将年龄(aging)的概念引入队列中进行进化，即将整个种群放在一个队列中，新加入一个元素时就会将队首的元素移除，从而使进化更加年轻化，并取得网络性能上的突破。

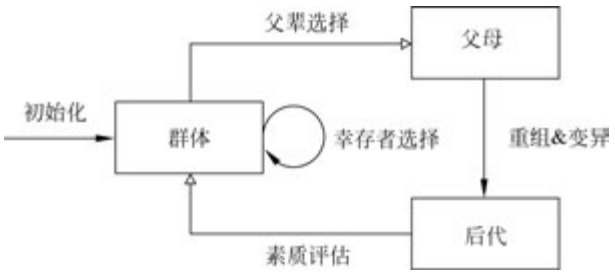


图 3-38 进化算法

3. 基于梯度的搜索策略

前面提到的基于强化学习和进化算法的方法本质上都还是在离散空间中搜索，它们将目标函数看作黑盒。如果搜索空间连续，目标函数可微，那么基于梯度信息可以更有效地搜索。CMU 和 Google 公司的学者在 *DARTS：Differentiable Architecture Search*[44] 一文中提出了 DARTS 方法，将一个要搜索最优结构的 Cell 看作包含 N 个有序节点的有向无环图，节点代表隐式表征(如特征图)，连接节点的有向边代表算子操作。DARTS 方法中最为关键的技巧是将候选操作使用 Softmax 函数进行混合。这样就将搜索空间变成了连续空间，目标函数成为可微函数。这样就可以用基于梯度的优化方法找寻最优结构了。搜索结束后，这些混合的操作会被权重最大的操作替代，形成最终的结果网络。另外，中国科学技术大学和微软发表的论文 *Neural Architecture Optimization*[45] 中提出另一种基于梯度的方法。它的做法是先将网络结构嵌入一个连续的空间，这个空间中的每一个点对应一个网络结构。在这个空间上可以定义准确率的预测函数。以它为目标函数进行基于梯度的优化，找到更优网络结构的嵌入表征。优化完成后，再将这个嵌入表征映射回网络结构。

3.8.3 性能评估策略

神经架构搜索的目的是找出优秀高效的架构，而性能评估策略的作用是对搜索出来的未知架构进行评价，较简单的方法是对这些未知架构进行训练与验证，但这么做需要消耗大量的时间与计算资源，目前工作主要集中在如何降低性能评估策略开销的同时，尽可能精确地评估未知架构的性能。为了减少性能评估所带来的计算量，基于各种模型的搜索方案被提出，通过搜索加速提升网络模型性能。下面将对常见的评估策略进行说明。

1. 基于低保真的性能评估策略

基于低保真的性能评估策略[46]的核心思想是在不牺牲太多精度的前提下，通过降低评估过程中的计算复杂度和资源需求，快速筛选出表现良好的神经网络结构。这种方法的关键在于找到一种能够平衡评估准确性和计算效率的近似方式。具体来说，基于低保真的性能评估策略可能包括以下几方面。

（1）简化模型结构：使用较小的网络模型或降低模型的复杂度，以减少计算量和内存需求。这种简化可以是在网络层数、神经元数量或连接方式等方面进行。

（2）快速训练策略：采用缩短训练周期、减少训练批次大小或使用更快的优化算法等方法，加速模型的训练过程。这有助于在短时间内获得模型的初步性能评估结果。

（3）近似性能度量：使用替代性的性能度量指标，这些指标可能是基于模型的部分输出、特定数据集的子集或简化任务的结果。这些近似度量可以快速给出模型性能的初步估计，而无须进行完整的训练和验证过程。

基于低保真的性能评估策略的优势在于其速度和灵活性。它可以在有限的计算资源下，快速筛选出潜在的优秀结构，并为进一步的精细评估提供候选。然而需要注意的是，低保真评估可能会引入一定的误差，因此其结果需要谨慎对待，并结合其他评估方法进行验证和校准。在实际应用中，基于低保真的性能评估策略可以根据具体需求和场景进行定制和优化。例如，可以根据任务的复杂性、数据集的规模以及计算资源的限制来选择合适的简化程度、训练策略和近似度量方法。

2. 基于权值共享的性能评估策略

权值共享法是减少参数个数的方法。2018 年，Hieu Pham[47]等提出的 ENAS 是用于自动化网络模型设计的方法，该方法快速有效且资源耗费低。具体来说，基于权值共享的性能评估策略包括以下关键步骤。

（1）定义超网络：首先，构建一个包含多个子网络或路径的超网络。这些子网络或路径代表不同的候选结构，而超网络则提供了一个统一的框架来共享权重。

（2）共享权重：在超网络中，不同的子网络或路径共享相同或部分的权重。这意味着在训练或评估过程中，这些权重只需要被更新或计算一次，然后可以被用于多个子网络或路径的性能评估。

（3）性能评估：通过单次或少数几次前向传播，评估超网络中多个子网络或路径的性能。由于权重是共享的，因此可以显著减少计算量，同时获得多个结构的性能信息。

这种基于权值共享的性能评估策略的优势在于其高效性和灵活性。它允许在有限的计算资源下，探索更多的神经网络结构，并快速识别出性能优异的结构。此外，由于权重是共享的，这种策略还可以促进不同结构之间的知识迁移和泛化能力的提升。然而，需要注意的是，基于权值共享的性能评估策略也可能存在一些挑战和限制。例如，如何有效地设计和构建超网络以确保权重的适当共享，以及如何准确地评估不同结构的性能而不受共享权重的影响等仍然是一些问题。

3. 基于早停的性能评估策略

基于早停的性能评估策略[48]是一种在神经架构搜索中广泛使用的技术，用于在训练过程中提前终止表现不佳的模型，以节省计算资源并加速搜索过程。该策略的核心思想是在验证集上监控模型的性能，一旦发现性能开始下降或不再显著提升，就立即停止训练，以避免过拟合和不必要的计算开销。具体来说，基于早停的性能评估策略通常包括以下几个步骤。

（1）数据划分：首先，将数据集划分为训练集、验证集和测试集。训练集用于训练模型，

验证集用于监控模型性能以决定是否提前停止训练，而测试集则用于评估最终模型的性能。

(2) 模型训练与验证：在训练过程中，模型在训练集上进行迭代训练，并在每个 Epoch 或每隔一定数量的 Epoch 后，在验证集上进行性能评估。常用的性能评估指标包括准确率、损失函数值等。

(3) 性能监控与判断：通过监控验证集上的性能指标，判断模型是否出现过拟合或性能下降的趋势。这可以通过观察指标的变化趋势、比较连续多个 Epoch 的性能表现等方式来实现。如果验证集性能连续多个 Epoch 没有提高或开始下降，则可以认为模型已经达到了其最佳性能，或开始出现过拟合。

(4) 提前停止训练：一旦决定提前停止训练，就立即终止当前的训练过程，并保存当前模型的状态。这样可以避免进一步的计算开销，并保留一个性能相对较好的模型。

(5) 最终评估：使用测试集对提前停止训练得到的模型进行最终的性能评估。这一步是为了验证基于早停策略选择的模型在实际应用中的表现。

基于早停的性能评估策略的优势在于它能够有效地避免过拟合，减少不必要的计算开销，并加速神经架构搜索过程。通过提前终止性能不佳的模型训练，可以更加高效地探索潜在的优秀网络结构。然而需要注意的是，早停策略也可能导致错过一些在后期阶段才展现出优秀性能的模型。

4. 基于代理任务的性能评估策略

基于代理任务的性能评估策略[49]的核心思想是利用一个或多个相对简单、快速的代理任务来近似评估目标任务的性能，从而快速筛选出潜在优秀的网络结构，并在后续阶段进行更精细的评估。具体来说，基于代理任务的性能评估策略包括以下几个关键步骤。

(1) 选择代理任务：代理任务的选择是关键，它应该能够反映目标任务的某些关键特性，同时保持相对简单和快速。例如，可以使用较小的数据集、降低输入数据的维度、简化网络结构或使用更少的训练轮次作为代理任务。

(2) 训练与评估代理任务：在选定的代理任务上，对候选网络结构进行训练和评估。由于代理任务相对简单，这个过程通常比直接在目标任务上进行训练和评估要快得多。

(3) 性能排序与筛选：基于代理任务上的性能评估结果，对候选网络结构进行排序和筛选。选择表现优秀的网络结构作为后续评估或搜索的候选对象。

(4) 精细化评估：对于在代理任务上表现优秀的网络结构，可以在目标任务上进行更精细的评估。这通常包括使用完整的数据集、更长的训练周期和更严格的评估指标来确保性能的准确性。

然而需要注意的是，基于代理任务的性能评估策略可能存在一定的局限性。由于代理任务与目标任务之间的差异，代理任务上的性能评估结果可能无法完全反映目标任务上的真实性能。因此，在使用该策略时，需要谨慎选择代理任务，并结合其他评估方法进行综合判断。

小结

深度学习作为机器学习领域中的一个重要分支，其核心在于构建和训练深度神经网络模型。深度学习学习了样本数据的内在规律和表示层次，这些学习过程中获得的信息对诸如文字、图像和声音等数据的解释有很大的帮助。本章首先介绍了深度学习相关概念以及基础知识，然后详细介绍了 7 种主流的深度学习模型，有卷积神经网络、生成式对抗网络、视觉注意力网络、多层感知机、扩散模型、图神经网络与神经架构搜索。总之，现有的深度学习方法在特征提取、模型灵活性等方面具有显著优势，但也面临着可解释性、过拟合等挑战。随着技术的不

断发展和优化，深度学习将在更多领域发挥重要作用，并有望解决当前存在的问题。同时，也需要加强深度学习的理论研究和实践应用，推动其向更加成熟和可靠的方向发展。

习题

1. 解释什么是深度学习，它与传统的机器学习有什么区别。

2. 描述神经网络的基本组成部分，并解释每个部分的作用。

3. 解释什么是前向传播和反向传播，它们在神经网络训练中的作用是什么。

4. 列举几种常见的激活函数，并描述它们的特点和适用场景。

5. 为什么激活函数在神经网络中很重要？

6. 对于一个给定的激活函数，如 ReLU，如何计算其导数？

7. 描述均方误差（MSE）和交叉熵损失（Cross-Entropy Loss）分别适用于哪些类型的任务。

8. 解释梯度下降算法的基本思想，并比较随机梯度下降（SGD）和批量梯度下降（BGD）的优缺点。

9. 列举几种常见的优化器，如 Adam、RMSprop 和 SGD，并简要说明它们的工作原理的差异。

10. 描述卷积神经网络（CNN）的基本结构，并解释卷积层、池化层和全连接层的作用。

11. 解释卷积操作在图像处理中的应用和意义。

12. 对于一个给定的图像分类任务，设计一个基本的 CNN 模型，并解释其设计思路。

参考文献

[1] Fukushima K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position[J]. Biological Cybernetics, 1980, 36(4): 193-202.

[2] 陈琨，王安志. 卷积神经网络的正则化方法综述[J]. 计算机应用研究，2024，41(4)：961-969.

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

[4] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[J]. Computer Science, 2014.

[5] 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.

[6] Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets[J]. Neural Information Processing Systems, 2014.

[7] Mirza M, Osindero S. Conditional generative adversarial Nets[J]. Computer Science, 2014: 2672-2680.

[8] Ghosh A, Kulharia V, Namboodiri V, et al. Multi-agent diverse generative adversarial networks[C]. IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2017.

[9] Mordido, Gonçalo, Yang H, Meinel C. Dropout-GAN: Learning from a dynamic ensemble of discriminators[J]. Computer Science, 2018.

[10] Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks[J]. Computer ence, 2015.

[11] Chen X, Duan Y, Houthooft R, et al. InfoGAN: Interpretable representation learning by information maximizing generative adversarial nets[J]. Neural Information Processing Systems, 2016.

[12] Zhu J Y, Park T, Isola P, et al. Unpaired image-to-image translation using cycle-consistent adversarial networks[C]//IEEE International Conference on Computer Vision (ICCV), 2017: 2242-2251.

[13] Hoang Q, Nguyen T D, Le T, et al. MGAN: Training generative adversarial nets with multiple

generators[C]//International Conference on Learning Representations,2018.
[14] Ghosh A,Kulharia V,Namboodiri V. Message passing multi-agent GANs[J]. Computer Science,2016.
[15] Zhao J,Mathieu M,Lecun Y. Energy-based generative adversarial network[J]. Computer Science,2016.
[16] Durugkar I,Gemp I,Mahadevan S. Generative multi-adversarial networks[J]. Computer Science,2016.
[17] Zhang H,Sindagi V,Patel V M. Image de-raining using a conditional generative adversarial network[J]. IEEE Transactions on Circuits and Systems for Video Technology,2017.
[18] Vaswani A,Shazeer N,Parmar N,et al. Attention is all you need[C]//Proceedings of Advances in Neural Information Processing Systems,Long Beach,Dec,2017. MIT Press,2017: 5998-6008.
[19] Dosovitskiy A,Beyer L,Kolesnikov A,et al. An image is worth 16×16 words: Transformers for image recognition at scale[C]//Proceedings of the International Conference on Learning Representations, Vienna,Austria,2021.
[20] Tolstikhin I O,Houlsby N,Kolesnikov A,et al. MLP-mixer: An all-MLP architecture for vision[J]. Advances in Neural Information Processing Systems,2021,34: 24261-24272.
[21] Kingma D P,Welling M. Auto-encoding variational Bayes[J]. arXiv. org,2014. DOI: 10. 48550/arXiv. 1312. 6114.
[22] Lecun Y,Chopra S,Hadsell R,et al. A tutorial on energy-based learning. In (Bak ± r et al. 2007), 192241[J]. 2007.
[23] Sohl-Dickstein J, Weiss E A, Maheswaranathan N, et al. Deep unsupervised learning using nonequilibrium thermodynamics[J]. JMLR. org,2015.
[24] Song Y,Ermon S. Generative modeling by estimating gradients of the data distribution. 2019[2024-04-16].
[25] Song Y,Ermon S. Improved techniques for training score-based generative models[J]. 2020.
[26] Kawar B,Elad M,Ermon S,et al. Denoising diffusion restoration models[J]. 2022. DOI: 10. 48550/arXiv. 2201. 11793.
[27] Song Y,Durkan C,Murray,I,et al. Maximum likelihood training of score-based diffusion models[J]. Advances in neural information processing systems. 2021,34: 1415-1428.
[28] Song Y,Sohl-Dickstein J,Kingma D P,et al. Score-based generative modeling through stochastic differential equations[C]//International Conference on Learning Representations. 2021.
[29] Gori M,Monfardini G,Scarselli F. A new model for learning in graph domains[C]//IEEE International Joint Conference on Neural Networks. IEEE,2005.
[30] Scarselli F,Gori M,Tsoi A C,et al. The graph neural network model[J]. IEEE Transactions on Neural Networks,2009,20(1): 61.
[31] Point O,Related S O R. Spectral networks and deep locally connected networks on graphs[J].
[32] Micheli A. Neural network for graphs: A contextual constructive approach[J]. IEEE Transactions on Neural Networks,2009,20(3): 498-511.
[33] Niepert M,Ahmed M,Kutzkov K. Learning convolutional neural networks for graphs[J]. JMLR. org, 2016. DOI: 10. 48550/arXiv. 1605. 05273.
[34] Hamilton W L,Ying R,Leskovec J. Inductive representation learning on large graphs[J]. 2017. DOI: 10. 48550/arXiv. 1706. 02216.
[35] Velickovic P,Cucurull G,Casanova A,et al. Graph attention networks[J]. stat,2017,1050: 20.
[36] Enzo,Leiva-Aravena,Eduardo,et al. Neural architecture search with reinforcement learning[J]. Science of the Total Environment,2019.
[37] He K,Zhang X,Ren S,et al. Deep residual learning for image recognition[J]. IEEE,2016. DOI: 10. 1109/CVPR. 2016. 90.
[38] Huang G,Liu Z,Laurens V D M,et al. Densely connected convolutional networks[J]. IEEE Computer Society,2016. DOI: 10. 1109/CVPR. 2017. 243.
[39] Zoph B,Vasudevan V,Shlens J,et al. Learning transferable architectures for scalable image recognition[J]. 2017. DOI: 10. 48550/arXiv. 1707. 07012.

[40] Liu H, Simonyan K, Vinyals O, et al. Hierarchical representations for efficient architecture search[J]. 2017. DOI: 10.48550/arXiv.1711.00436.

[41] HU Y, WANG X, GU Q. PWSNAS: Powering weight sharing NAS with general search space shrinking framework[J]. IEEE Transactions on Neural Networks and Learning Systems, 2022, 22: 1-14.

[42] Real E, Moore S, Selle A, et al. Large-scale evolution of image classifiers[J]. 2017. DOI: 10.48550/arXiv.1703.01041.

[43] Real E, Aggarwal A, Huang Y, et al. Regularized evolution for image classifier architecture search[C]. Proceedings of the AAAI Conference on Artificial Intelligence, 2018, 33. DOI: 10.1609/aaai.v33i01.33014780.

[44] Liu H, Simonyan K, Yang Y. Darts: Differentiable architecture search[J]. arXiv preprint arXiv: 1806.09055, 2018.

[45] Luo R, Tian F, Qin T, et al. Neural architecture optimization[J]. 2018. DOI: 10.48550/arXiv.1808.07233.

[46] Klein A, Falkner S, Bartels S, et al. Fast Bayesian optimization of machine learning hyperparameters on large datasets[C]//2016.

[47] Pham H, Guan M Y, Zoph B, et al. Efficient neural architecture search via parameter sharing[J]. 2018. DOI: 10.48550/arXiv.1802.03268.

[48] Domhan T, Springenberg J T, Hutter F. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves[C]//International Conference on Artificial Intelligence. AAAI Press, 2015.

[49] Mitchell T M. Machine learning[M]. New York: McGraw-Hill, 2003.