基于 LeNet-5 卷积神经网络的手写数字识别

尹恒

智能 2019-01 班 2019114895

Abstract 摘要

本课程设计基于 Pytorch 机器学习框架，构建 LeNet-5 卷积神经网络，对 MNIST 手写数字数据集进行识别、分类，并采用精确率 Precission、召回率 Recall、F1-Score 对模型的泛化性能进行评估。在本课程设计报告中，介绍了卷积神经网络的结构、网络层以及损失函数、激活函数等概念，同时对本课程设计的实验设计部分作了详细阐述，并给出最后的实验结果及分析总结。

通过对卷积神经网络的参数学习率、优化器进行超参数调整，观察到随着学习率的变化，训练时损失函数的变化情况，得出学习率不宜过大，会导致无法收敛的问题；也不宜过小，可能导致收敛速度过慢的问题。同时还对比了 Adam 算法与 SGD 随机梯度下降算法在本课程设计中的表现，这两者的训练速度相仿，但是训练的效果 SGD 略逊于 Adam 算法的结论。通过调整超参数得到了使得模型性能表现最好的一组参数：Adam 优化器，学习率为 0.001。此时精确率、召回率、F1-Score 分别为 0.991291、0.991331、0.991293。与其他的机器学习算法模型的对比结果如下表所示。

机器学习模型	运行时间	精确率 Precision	召回率 Recall	F1-Score
高斯朴素贝叶斯	00:03	0.692811	0.557209	0.521213
KNN	01:09	0.986905	0.986528	0.986681
逻辑回归	00:09	0.928134	0.927880	0.927941
SVM（线性核）	02:07	0.888582	0.884402	0.884494
SVM（高斯核）	08:33	0.989923	0.989885	0.989901
决策树（max=15）	00:11	0.984501	0.984272	0.984369
随机森林（n=15）	00:05	0.999735	0.999735	0.999735
LeNet-5（最优情况）	03:16	0.991291	0.991331	0.991293

1 方法介绍

1.1 卷积神经网络

1.1.1 简介

卷积神经网络（Convolutional Neural Network，CNN 或 ConvNet）是一种具有局部连接、权重共享等特性的深层前馈神经网络[1]。其模仿生物的视知觉机制，卷积神经网络是目前深度学习技术领域极具代表性的神经网络，基于卷积神经网络的各种模型，在图像特征分类、场景识别等问题上的表现十分优秀。

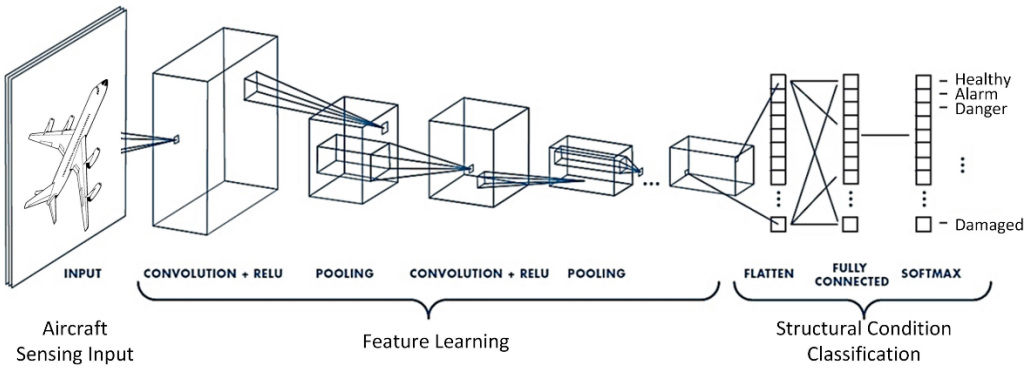


图 1.1 一种用于飞机健康状况检测的卷积神经网络模型结构

卷积神经网络 CNN 主要用来处理图像信息，在卷积神经网络出现之前，传统的前馈式全连接网络处理图像信息时会出现以下的问题：

a) 参数过多

针对一个输入为 28*28*3 维的图像（图像高度、宽度为 28px，有 3 个色彩通道 RGB），全连接网络的输入层到第一个隐藏层的每个神经元就有 28*28*3 个互相独立的连接，而每个连接都对应着各自的权重参数。随着神经元数量的增多，权重参数的规模急剧增加，导致整个网络训练的效率十分低下，同时可能出现过拟合情况。上述示例仅仅针对的是小像素图像，如今随着硬件性能的提高，图像的大小动辄几千像素，且一般为 RGB 三个色彩通道，因此全连接神经网络模型很难直接将图像作为模型的输入。

b) 不具有局部不变性

图像数据一般具有**局部不变性**特点，局部不变性是指图像目标经过平移/旋转/尺度变化后

仍然能取得相似的检测结果。例如，两朵玫瑰花可能出现在一个图像的两个位置，但其实际上拥有相同的特征，而全连接神经网络很难对这些不变的特征进行识别与处理，只能当作完全不同的事物进行训练与测试。

而卷积神经网络通过局部连接与权重共享很好的解决了上述问题，有效的解决了：使用神经网络处理图像数据的问题。

a) 局部连接

相比于传统的全连接神经网络，卷积神经网络卷积层的每一个神经元只与下一层的某个局部区域的神经元相连接，这就是局部连接，这样能够大大降低网络中需要训练的参数的数量。

b) 权重共享

权重共享是指，对于同一个卷积核，其对于这一层上所有的神经元都是相同的。通俗的解释是，每个卷积核只负责对图像中特定的数据特征进行提取，因此可以通过提高卷积核的数量，从而获得输入数据的更多特征。

卷积神经网络一般由三种类型的层构成，分别为：**卷积层**、**池化层**、**全连接层**，通过将这些层进行组合，就可以构建一个卷积神经网络。

图 1.2 一种卷积神经网络结构

1.1.2 卷积层

卷积层的作用是提取输入数据的一个局部特征，不同的卷积核相当于不同的特征提取器（Fliter）。卷积层每个神经元和原图像只在一个小区域进行全连接，称为局部连接，该连接的空间大小被称之为感受野（Receptive Field）。因为在处理图像这样的高维度输入时，让每个神经元都与前一层中的所有神经元进行全连接是不现实的。在前向传播的时候，让每个滤波器都在输入数据的宽度和高度上做卷积运算，然后计算这个滤波器和输入数据对应每一个区域的内积，最终会生成一个 2 维的特征映射。在每个卷积层上，一般有多个滤波器组成集合（比如 12 个），每个都会生成一个不同的二维特征映射。将这些激活映射在深度方向上层叠起来就生成了这个卷积层的输出 3D 数据。每张特征映射对应的所有神经元参数都相同（因为实际上就是

同一个滤波器在图像上不同位置滑动的结果，每到一个位置就是一个神经元），称为权重共享共享。

卷积层的所有神经元与原始图像卷积后，输出数据体的尺寸由三个超参数控制：深度（depth），步长（stride）和零填充（padding）。

a) 深度

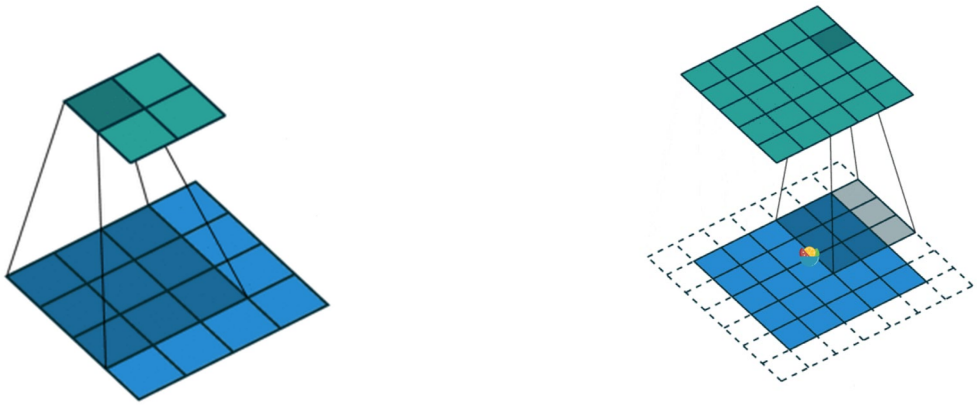
深度即指 Fliter 过滤器的数量，一个卷积层往往有多个过滤器。

b) 步长

步长为过滤器每次移动的像素的数量，即当步长为 1，滤波器每次移动 1 个像素。当步长为 2（实际中很少使用比 2 大的步长），滤波器滑动时每次移动 2 个像素。这个操作会让输出输入数据的维度降低。

c) 零填充

通过前面的分析可知，输入图像与卷积核进行卷积后输入图像的边缘被“修剪”掉了（边缘处只检测了部分像素点，丢失了图片边界处的众多信息）。这是因为边缘上的像素永远不会位于卷积核中心，而卷积核也没法扩展到边缘区域以外。为解决这个问题，可以在进行卷积操作前，对原图像进行边界填充（Padding），也就是在图像的边界上填充一些值，以防止进行卷积操作的时的信息丢失，通常都用“0”来进行填充的。



a) 未进行零填充的卷积操作　　b) 进行零填充后的卷积操作

图 1.3 卷积操作示意图

1.1.3 池化层

池化层（Pooling）也被称之为汇聚层或子采样层，顾名思义，其在神经网络中起到的作用即为降采样，降低特征数量，从而减少参数数量。数据经过卷积层的处理之后，实际维度并不会下降太多，甚至可能会增加，通过连续的卷积操作之后，参数的数量会增大，可能出现参数爆炸的情况。因此需要在连续的卷积层之间会周期性地插入一个池化层，能够逐渐降低数据体的空间（宽、高）尺寸，这样的话就能减少网络中参数的数量，使得计算资源耗费变少，也能有效控制过拟合。

池化的具体操作是将一个像素点与它周围的数据点进行聚合统计，缩减特征图的尺寸，然后对其相邻的区域取平均值或最大值。在卷积神经网络通常使用的池化层有最大池化（Max Pooling）和平均池化（Mean-Pooling）。

a) 最大池化（Max Pooling）

顾名思义，最大池化即为在池化过程中取数据切片中的最大值。在池化操作中最常用的是2*2 的过滤器（Fliter），以步长为 2 来对切片进行降采样，如下图所示，一个 1*4*4 的图像数据通过 2*2 的过滤器最大池化后，得到了一个 1*2*2 的降采样数据。

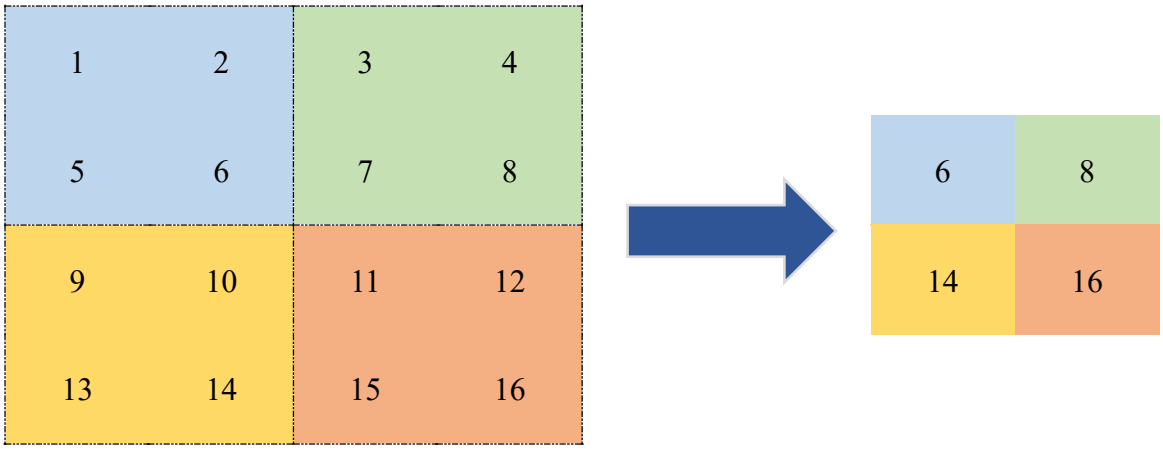


图 1.4 最大池化操作

b) 平均池化（Mean-Pooling）

平均池化与最大池化类似，只是取值为切片中数据的均值，其余操作均一致，如下图所示，一个 1*4*4 的图像数据通过 2*2 的过滤器平均池化后，得到了一个 1*2*2 的降采样数据。

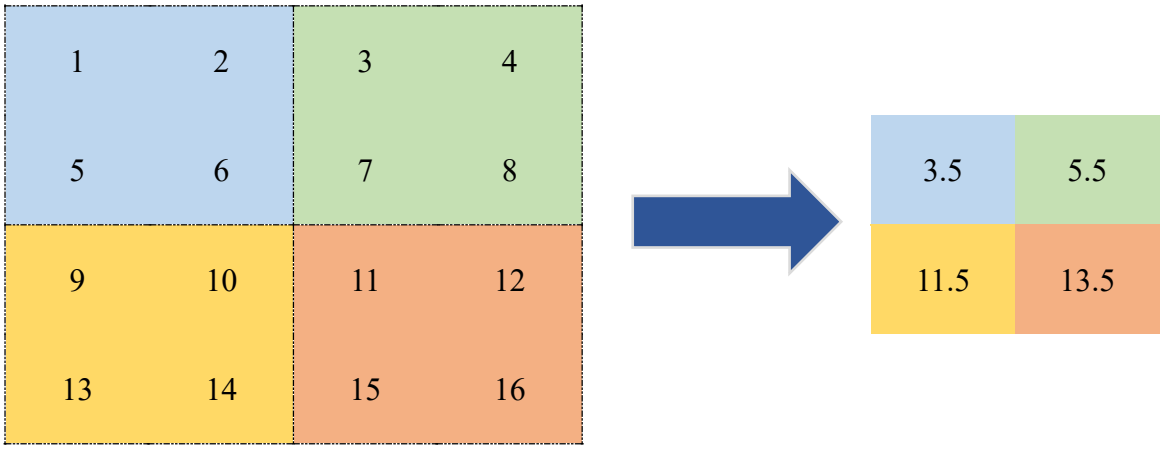


图 1.5 平均池化操作

1.1.4 全连接层

全连接层，顾名思义，神经元对于前一层中的所有激活数据是全连接的，在卷积神经网络中一般作为末尾的输出层。通过与前一个网络层的神经元全部连接，整合前层网络提取的特征，并把这些特征映射到样本标记空间。通过对前一个网络层的输出进行加权求和$y_{w,b}(x) = f(\sum_{i=1}^{n} w_i x_i + b_i)$，其中$w_i$是全连接层中的权重系数，$x_i$是上一层第 i 个神经元的值，$b_i$是全连接层的偏置量。将计算得到的值经过激活函数得到输出值，这也是卷积神经网络的最终输出。

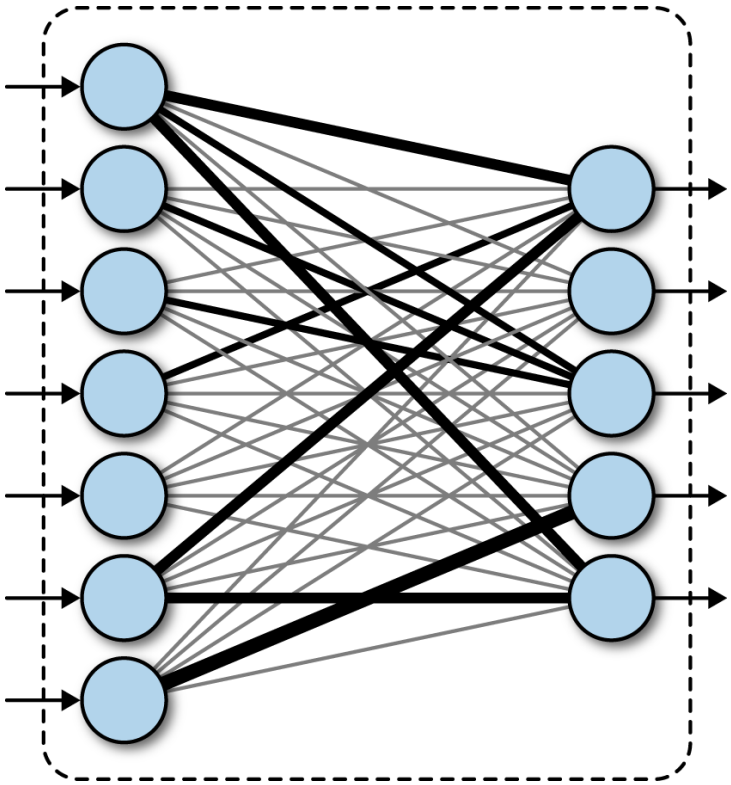


图 1.6 全连接层示意图

1.1.5 激活函数

激活函数在神经网络模型中有着十分重要的地位。为了增强网络的表示能力和学习能力，激活函数一般具有以下的性质。

a) 连续并可导的非线性函数。可导的激活函数可以直接利用数值优化的方法来学习网络参数（例如梯度下降算法）。

b) 激活函数及其导函数要尽可能简单，方便计算，有利于提高网络的前向计算效率和学习效率。

c) 激活函数的导函数值域需要在一个合适的区间内，不能太大亦不能太小，否则辉影响训练的效率和稳定性。

如果不使用激活函数，那么每一层节点的输入都是上一层输出的线性组合，无论神经网络增加到多少层，神经网络的输出都是输入数据的线性组合，那么隐藏层的增加对神经网络新能的提高就失去了意义。对于复杂的非线性问题而言，拟合的效果较差，同时也增加了不必要的训练性能开销。因此需要引入非线性的激活函数，给隐藏层的输出增加非线性的元素，这样能够显著的提高神经网络模型的拟合能力。加入激活函数之后，神经网络理论上能够拟合任意复杂的函数。

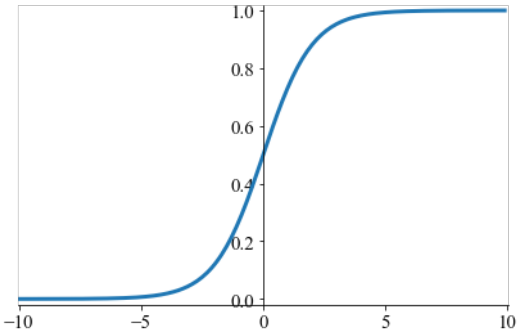
以下将对常用的激活函数进行介绍。

a) Sigmoid 函数

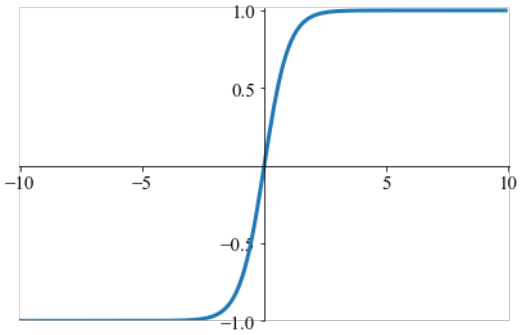
Sigmoid 函数是一个单调、有界、可导的函数，其表达式为：

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.1}$$

根据表达式可绘制出 Sigmoid 函数的图像，如下所示，可以看出 Sigmoid 函数的值域为（0，1）的开区间，它能够把输入的数转化为（0，1）之间的某个数，因此，其输出常被作为概率。在历史上，Sigmoid 函数被广泛使用过，但近年来，因为它的值域仅在（0，1）之间，因此在进行反向传播计算时，存在着梯度消失等问题，使得训练速度变慢甚至难以收敛的问题。同时，由于其存在着幂运算，使得计算机求解的耗时较长，对于规模较大的神经网络，收敛速度会大大增加。



a) Sigmoid 函数图像



b) Tanh 函数图像

图 1.7 Sigmoid 和 Tanh 函数图像

b) Tanh 函数

Tanh 激活函数是 Sigmoid 函数的变种，Tanh 函数的表达式为：

$$\mathrm{Tan\,h}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{1.2}$$

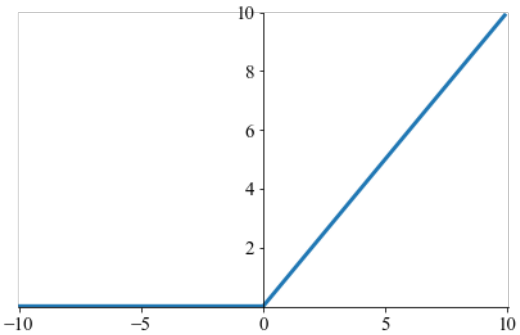
根据表达式可绘制出 Tanh 函数的图像，如上图右所示，可以看出 Tanh 和 Sigmoid 函数的不同点是 Tanh 函数的输出值落在[-1,1]之间，输出结果的平均值为 0，因此 Tanh 输出可以进行标准化。但梯度消失的问题以及幂运算的问题依然存在。

c) ReLU 函数

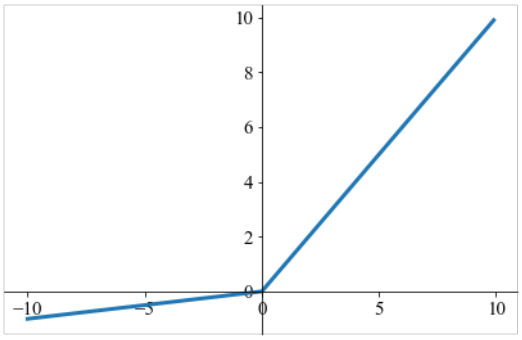
ReLU（Rectified Linear Unit，线性整流）函数是目前神经网络中被广泛运用的激活函数，其表达式为：

$$ReLU = max(0, x) \tag{1.3}$$

从表达式即可看出，对于小于 0 的部分，ReLU 函数的取值都是 0，对于大于 0 的情况，其函数值为本身，因此可以画出这个分段函数的图像，如下所示。



a) ReLU 函数图像



b) Leaky-ReLU 函数图像

图 1.8 ReLU 和 Leaky ReLU 函数图像

ReLU 函数的出现，相较于之前的 Sigmoid 函数、Tanh 函数等有着以下的优势：由于小于 0 的部分，激活后为 0，可以减少训练中的计算量，同时由于其没有幂计算的步骤，计算效率也有所提升。同时，它还能减轻梯度消失和梯度爆炸的问题。因此近年来 ReLU 激活函数常常被人们所使用。然而，由于其只是简单地将小于 0 的输入的激活值置为 0，这样处理会导致神经网络的计算过程丢失部分特征，因此出现了带泄露的 ReLU 函数（Leaky ReLU）对小于 0 的部分进行修正（小于 0 时为泄露较小的线性函数）。

1.1.6 损失函数

损失函数（Loss Function）是用来量化模型预测结果与真实值之间差异程度的函数，也被称为误差函数（Error Function）。这个差异越小，说明预测结果越准确，而算法模型的训练目标就是使得这个差异减小。优化算法会通过优化损失函数来不断调整模型权重，使其最好地拟合样本数据。无论是传统的机器学习，还是深度学习、神经网络都需要用到损失函数，从而实现对误差的衡量。常用的损失函数包括平方损失函数和交叉熵损失函数，下面将对其进行介绍。

a) 均方差损失 MSE

均方误差（Mean Square Error,MSE）是回归损失函数中最常用的误差，它是预测值$\hat{y}$与目标值y之间差值平方和的均值，其函数表达式为：

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y_i})^2 \tag{1.4}$$

b) 交叉熵损失 Cross-Entropy Loss

交叉熵损失函数运用了极大似然估计的思想，常用于分类问题，特别是运用在神经网络中。对于二分类问题，交叉熵损失函数的形式为：

$$L\big(Y, f(X)\big) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot ln(p_i) + (1 - y_i) \cdot ln(1 - p_i)] \tag{1.5}$$

其中，y_i表示样本 i 的真实标签，正类为 1，负类为 0；p_i表示样本 i 预测为正的概率。

对于多分类问题，交叉熵损失函数可以拓展为：

$$L\big(Y, f(X)\big) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{c} y_{i_c} \cdot ln\big(p_{i_c}\big) \tag{1.6}$$

其中，C 表示类别的数量；y_{i_c}是指示变量（1/0），如果样本 i 的类型为 c 则对应值为 1，否则为 0；p_{i_c}表示对于样本 i 属于类别 c 的概率。

1.1.7 优化方法

对于一个权重参数未经训练的神经网络而言，优化方法存在的意义就是：通过调整神经网络的权重参数，使得损失函数的值减小，从而使得神经网络的拟合效果更好。常用的优化算法有：批梯度下降算法（Batch Gradient Descent ,BGD）、随机梯度下降算法（Stochastic Gradient Descent ,BGD）、自适应矩估计算法（Adaptive moment estimation ,Adam），下面将一一介绍。

a) 批梯度下降算法 BGD

批梯度下降法又称全梯度下降法，对于梯度下降而言，为了使得损失最小，先假定一个初始的权重值，用梯度下降法做迭代逼近。通过为每个权重计算一个梯度，从而更新权值，使目标函数尽可能最小化。批梯度下降法，每迭代一步，都要用到训练集所有的数据，进行权值的更新，更新公式如下所示：

$$W_{k+1} = W_k - \alpha \frac{1}{n} \sum_{i=1}^{n} g_k^i \tag{1.7}$$

批处理梯度下降涉及到在每一步整个训练集上的计算，因此在非常大的训练数据上它是非常缓慢的。相反的，批量梯度下降对于凸或相对光滑的误差流形是很好的。在这种情况下，可直接向最佳解决方案移动。

b) 随机梯度下降算法 SGD

随机梯度下降算法是机器学习中常用的优化算法，与 BGD 相比，随机梯度下降算法使用单个随机样本的梯度作为方向，步长是一个可随时调整的参数，其更新公式如下所示：

$$W_{k+1} = W_k - a g_k^i \tag{1.8}$$

由更新公示可以看出，SGD 每次迭代只使用一个样本计算梯度，训练速度相对 BGD 更快。但如果输入数据的的分布不均衡，包含较多噪声，则会使得损失函数在收敛过程中产生震荡。因此引入了 batch 的概念，即每次使用一个 batch 的数据来计算梯度，然后将梯度平均之后再使用 SGD 更新参数，这样能够减少训练过程中的震荡，引入 batch 后的更新公式如下，其中 m

为一个批量包含样本的个数，需要根据实际情况决定。

$$W_{k+1} = W_k - \alpha \frac{1}{m} \sum_{j=1}^{m} g_k^{i_j} \tag{1.9}$$

c) 自适应矩估计算法 Adam

Adam 算法和传统的随机梯度下降不同。随机梯度下降保持单一的学习率（即 alpha）更新所有的权重，学习率在训练过程中并不会改变。而 Adam 通过计算梯度的一阶矩估计和二阶矩估计而为不同的参数设计独立的自适应性学习率。

1.2 LeNet-5 网络

LeNet-5 是一个经典的卷积神经网络模型。它于 1998 年在 LeCun 发表的《xxxx》论文中首次提出。它是 CNN 卷积神经网络的开山之作，其成功的应用于邮政编码的识别上。其网络结果如下图所示。

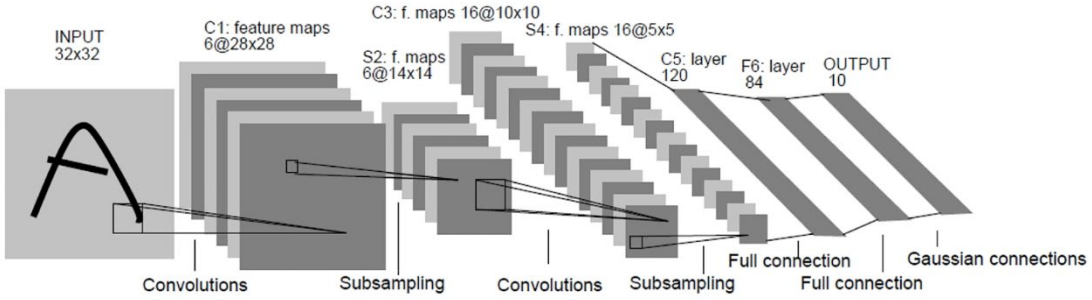


图 1.9 LeNet-5 网络结构图

如上图所示，LeNet-5 一共包含 7 层（不含输入层 INPUT），分别为卷积层 C1、池化层 S2、卷积层 C3、池化层 S4、卷积层 C5、全连接层 F6、输出层 OUTPUT。

a) 输入层 INPUT

LeNet-5 接受的输入大小为 1*32*32（一个色彩通道，即灰度图像）。

b) 卷积层 C1

输入为 1*32*32 的图像数据，使用 6 个 5*5 大小的卷积核，步长为 1，卷积后得到的结果是 6 个 28*28 大小的特征图。

c) 池化层 S2

池化层的核大小为 2*2，在 LeCun 的论文中，采用的池化方式为 4 个输入相加后乘一个可训练的参数，再加上一个可训练的偏置值，结果通过 Sigmoid 激活函数激活。但由于 Sigmoid 作为激活函数容易出现梯度消失、梯度爆炸等问题，导致出现收敛速度过慢的问题。因此在此改为最大池化+ReLU 函数激活的方式。

d) 卷积层 C3

使用 16 个 1*5*5 大小的卷积核，padding=0，stride=1 进行卷积，得到 16 个 10×10 大小的特征映射。C3 中可训练的参数并没有直接连接到 S2 中所有的特征映射，而是采用稀疏连接的方式，如下图所示。之所以采用这种方式，是因为论文发表时（1998 年）计算机的计算能力较弱，不支持直接连接的方式。而由于计算机性能的飞速发展，现在已经可以采用直接连接的方式。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

图 1.10 池化层 S2 与卷积层 C3 的连接方式

e) 池化层 S4

与 S2 类似，不再赘述。

f) 卷积层 C5

使用 120 个 1*5*5 大小的卷积核，与卷积层 C3 不同，C5 的输入是池化层 S4 的全部特征映射，由于 S4 得到的 16 个特征映射的大小为 5*5，与 C5 卷积核大小一致，因此经过 C5 得到 1*1 大小的特征映射 120 个。。

g) 全连接层 F6

F6 是全连接层，共有 84 个神经元，与 C5 层进行全连接，即每个神经元都与 C5 层的 120 个特征图相连。计算输入向量和权重向量之间的点积，再加上一个偏置，结果通过 Sigmoid 函数输出，在本课程设计中采用 ReLU 函数作为激活函数。

h) 输出层 OUTPUT

最后的 Output 层也是全连接层，在论文中是 Gaussian Connections，采用了 RBF 函数（即径向欧式距离函数），计算输入向量和参数向量之间的欧式距离。在本实验中采用 Softmax 取代，输出最后的分类结果。

综上所示，LeNet-5 的网络参数配置如下表所示。

表 1.1 LeNet-5 网络参数配置表

网络层	核大小	核数量	步长	输入大小	输出大小
卷积层 C1	5*5	6	1	32*32*1	28*28*6
池化层 S2	2*2	\	2	28*28*6	14*14*6
卷积层 C3	5*5	16	1	14*14*6	10*10*16
池化层 S4	2*2	\	2	10*10*16	5*5*16
卷积层 C5	5*5	120	1	5*5*16	1*1*120
全连接层 F6	\	\	\	1*1*120	1*1*84
输出层 OUTPUT	\	\	\	1*1*84	1*1*10

1.3 Pytorch 框架

PyTorch 是一个基于 Torch 的 Python 开源机器学习库，可用于图像处理、自然语言处理等算法模型的构建。在没有机器学习框架以前，进行深度学习相关技术的研究，需要不断的创建各种繁琐的环境代码。此时就涌现了很多优秀的深度学习框架，如 TensorFlow、PyTorch 等，这些框架的出现使得研究者可以更关注算法本身，而不需要完成很多重复的代码工作。Pytorch 提供了神经网络模块以及数据集加载处理模块。

在本课程设计中，拟采用 Pytorch 框架搭建 LeNet-5 网络结构模型，并通过其中的 DataLoader 对数据进行加载，torchvision 对原始数据集进行增强和标准化，利用 Pytorch 内置的优化器进行参数训练，最后进行前向计算，实现测试集上的预测。

2 数据集介绍

本次实验基于 Yann LeCun 等提供的 MNIST 手写数字数据集。MNIST 数据集是一个十分经典的数据集，常用于计算机视觉（Computer Vision）和深度学习，被誉为计算机视觉领域的“Hello World”。修改于美国国家标准与技术研究所（National Institute of Standards and Technology，NIST）的两个数据库。数据集中数据由 250 个不同人手写的数字构成，其中 50%是高中学生, 50%来自人口普查局 （The Census Bureau）的工作人员。测试集（Test Set）也是同样比例的手写数字数据。

MNIST 数据库包含 60，000 张训练图像和 10，000 张测试图像。每一条数据的维度都是 1*28*28=784 维。训练集的一半和测试集的一半取自 NIST 的训练数据集，而另一半训练集和另一半测试集取自 NIST 的测试数据集。原始数据集为 ubyte 字节文件，而非常见的 png、jpg 等图像格式，以下是利用 python 读取该数据集，对数据集中部分数据的可视化。

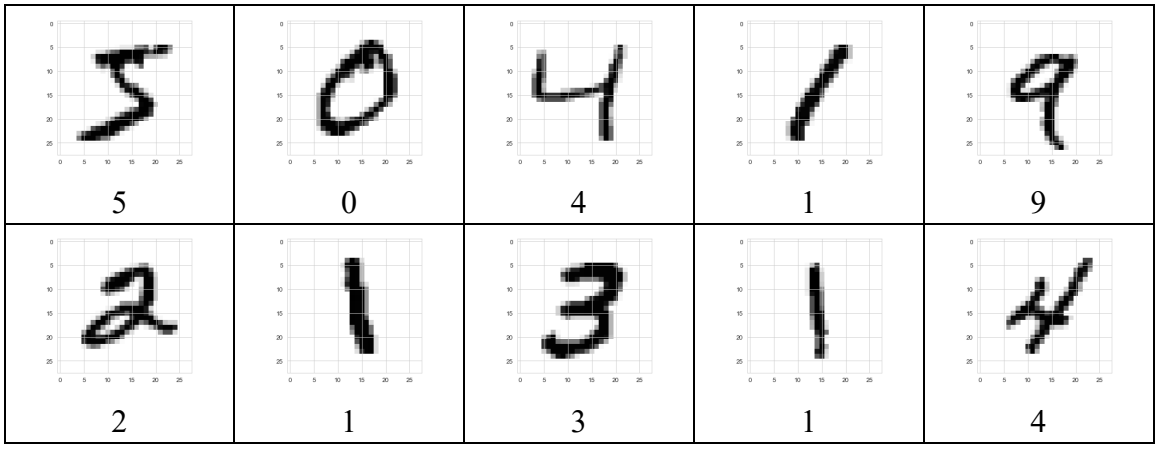


图 2.1 MNIST 数据集中的部分数据可视化

通过肉眼，能够清晰地辨认出数字背后的标签。但不同图片的书写风格存在着较大的差异。

该数据集可通过其官方网址获取，或通过 Pytorch 内置的数据加载工具下载。

3 实验设计

本次课程设计实验的步骤大致分为：数据增加与预处理、卷积神经网络搭建、神经网络参数训练、模型性能测试、性能对比五个部分，下面将分别对其进行介绍。

3.1 数据增强与标准化

3.1.1 数据增强

数据增强是指让有限的数据通过各种手段，产生更多的数据，增加训练样本的数量以及多样性，能够有效的减弱过拟合情况，使得训练得到的模型拥有更强的泛化性能，但这样也可能带来一些噪声数据，同时可能会使得模型在训练集上的表现稍弱于未进行增强的数据。

针对图像处理，数据增强的方式有翻转、裁剪、旋转、位置变换、平移等，针对 MNIST 数据集，制作者在制作时将数字放在了图像中间的 20*20 像素的空间中（全图为 28*28），因此可以对原始训练集进行平移和旋转操作。但在操作的时候需注意，平移和旋转的范围需要控制，以防数据丢失其原有特征，使得标签不对应，甚至引入错误数据（例如旋转角度过大时，数字 6、9 对应的图像会出现标签丢失的问题）。

3.1.2 数据标准化

MNIST 数据集提供的原始数据是一个个 1*28*28 大小的图像数据，原始图像的每一个像素，都是由 0-255 之间的灰度值构成。如果直接将其输入 LeNet-5 卷积神经网络中，会因为输入性质的不稳定，减慢神经网络的学习训练速度，使得收敛速度变慢。

因此需要在输入神经网络之前，需要对数据（包含训练集与测试集）进行标准化处理。即通过均值与标准差对每张图像数据进行处理。

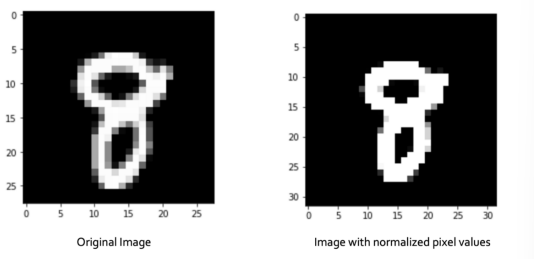


图 3.1 原始图像与标准化后的图像对比

3.1.3 实施方法

在使用 Pytorch 进行实验时，可以使用 Pytorch 的处理图像视频的 torchvision 工具集直接下载 MNIST 的训练和测试图片，torchvision 包含了一些常用的数据集、模型和转换函数等等，比如图片分类、语义切分、目标识别、实例分割、关键点检测、视频分类等工具。

因此，数据增强和预处理可采用 torchvision 中提供的方法进行。数据增强仅针对训练集，而不针对测试集，而数据标准化针对训练集和测试集。首先利用 torchvision 进行随机平移（5

个像素以内）、随机旋转（10 度以内），实现数据增强，再利用样本均值、标准差对数据进行标准化。

3.2 卷积神经网络搭建

本课程设计使用的卷积神经网络为经典的 LeNet-5 网络。具体的网络结构参见“1.2 LeNet-5 网络”节，此处不再赘述。利用 Pytorch 能够快速搭建该网络，代码如下所示。

```
# LeNet-5 卷积神经网络模型
class LeNet5(nn.Module):
  def __init__(self):
    super(LeNet5, self).__init__()
    # 卷积层
    self.conv1 = nn.Conv2d(1, 6, 5)
    # 池化层
    self.pool1 = nn.MaxPool2d(2, 2)
    # 卷积层
    self.conv2 = nn.Conv2d(6, 16, 5)
    # 池化层
    self.pool2 = nn.MaxPool2d(2, 2)
    # 全连接层
    self.fc1 = nn.Linear(16 * 4 * 4, 120)
    # 全连接层
    self.fc2 = nn.Linear(120, 84)
    # 全连接层(输出)
    self.fc3 = nn.Linear(84, 10)

  # 前向传播进行预测
  def forward(self, x):
    x = self.pool1(functional.relu(self.conv1(x)))
    x = self.pool2(functional.relu(self.conv2(x)))
    x = x.view(-1, 16 * 4 * 4)
    x = functional.relu(self.fc1(x))
    x = functional.relu(self.fc2(x))
    x = self.fc3(x)
      return x
```

3.3 神经网络参数训练

本课程设计采用交叉熵损失函数来衡量预测值与真实值之间的差异。分别通过随机梯度下

降 SGD 算法和 Adam 优化算法对参数进行训练。

3.4 超参数调整与模型性能评价

3.4.1 超参数调整

本课程设计中涉及到的超参数主要为学习率、参数优化器的选择等，因此需要对这两个参数进行调整，以对比模型效果，epoch 固定为 10，batch_size 固定为 32。

学习率的调整范围为：0.1、0.01、0.001、0.0001（4 种）；

参数优化器的选择为：SGD、Adam（2 种）。

3.4.2 性能评价指标

本次实验涉及到多分类问题的评价，与二分类问题类似，都可用混淆矩阵、$Precision$精确度、$Recall$召回率、$F1-score$进行评价。

但其定义与二分类问题中有所不同，涉及到$macro-P/R/F1$，$micro-P/R/F1$。对于多分类问题的混淆矩阵，其维度为$n*n$（n 为分类数目）。$Macro-averaging$是先在每个二分类上分别计算各类的指标，然后取平均值。$Micro-averaging$则先计算总TP值，其次算总FP值，然后按指标公式计算。这两个指标有如下的差异：

① $Macro-averaging$ 是在分别计算了每一类的指标后，求其算术平均值，对每一类都是等同而视的。可以说，在类别样本数目分布不均衡时，$Macro$会给予样本数目较少的类别与样本数据较大的类别同等的重视程度。

② $Micro-averaging$ 是详细统计了多分类中，每一个样本的预测结果，然后再计算相应指标的，重视的是每一个样本的结果。在$Micro-averaging$中，其实已经不存在多分类的区别了，所有的类都成了一个类。重点关注每一个样本的结果，而弱化了类别的区别，对于整体数据集的结果来说，其实是更趋近于客观的结果。

由于本次实验中 MNIST 数据集数据分布较为平衡，因此使用$\underline{Macro-average}$指标。

a) 精准率（Precision，P）

即预测正例中预测正确的占比，也被称为查准率，可以体现算法的准确度，计算公式如下。

$$marcoP = \frac{1}{n}\sum_{1}^{n} P_i \tag{3.1}$$

其中：

$$P_i = \frac{TP_i}{TP_i + FP_i} \tag{3.2}$$

b) 召回率（Recall，R）

即真实正例中预测正确的占比，也被称为查全率，可以体现算法对结果的查全能力，计算公式如下。

$$marcoR = \frac{1}{n}\sum_{1}^{n} R_i \tag{3.3}$$

其中：

$$R_i = \frac{TP_i}{TP_i + FN_i} \tag{3.4}$$

c) F1-Score

F1-Score 是分类问题的一个衡量指标，它同时兼顾了分类模型的精确率和召回率。F1 分数可以看作是模型精确率和召回率的一种加权平均，它的最大值是 1，最小值是 0。

$$marcoF_1 = \frac{2 \times macroP \times macroR}{macroP + macroR} \tag{3.5}$$

3.5 性能对比

在得到卷积神经网络对 MNIST 测试集的预测结果后，将使用之前在实验中用过的机器学习分类算法进行性能对比。分别采用朴素贝叶斯、KNN、逻辑回归、SVM（线性核/高斯核）、决策树、随机森林等算法对 MNIST 数据集进行训练与测试，利用精准率（Precision，P）和召回率（Recall，R）以及 F1-Score 对模型泛化性能进行评价，并与 LeNet-5 卷积神经网络模型进行对比，得出相应结论。

4 实验环境

4.1 实验平台介绍

本次实验基于联想 R9000P 笔记本平台完成，具体的软硬件配置信息如下表所示。

表 4.1 实验软硬件环境配置

配置项	配置内容
系统	Windows 11 Professional 21H2 22000.708
CPU	AMD Ryzen 7 5800H 3.20 GHz
GPU	Nvidia RTX 3060 Laptop 6GiB
RAM	16GiB
NVIDIA Driver	512.59
Cuda	11.6
Cudnn	8.2
Python	3.6.13
Conda	4.10.3
IDE	PyCharm 2022.1

4.2 Python 依赖环境

本课程设计中 LeNet5 卷积神经网络的搭建、训练与测试基于 Pytorch 实现，同时为了与 SVM、决策树、逻辑回归等机器学习分类算法进行对比，用到了 sklearn 库。因此本课程设计主要依赖于下列 Python 软件包（篇幅限制，此处仅列出部分，完整环境依赖见附件）。

表 4.2 Python 依赖环境表

软件包名	版本
scipy	1.4.1
torch	1.10.2+cu113
torchvision	0.11.3
jupyter	1.0.0

5 实验结果

按照上述实验设计进行实验，得到的结果如下。

5.1 不同超参数下表现

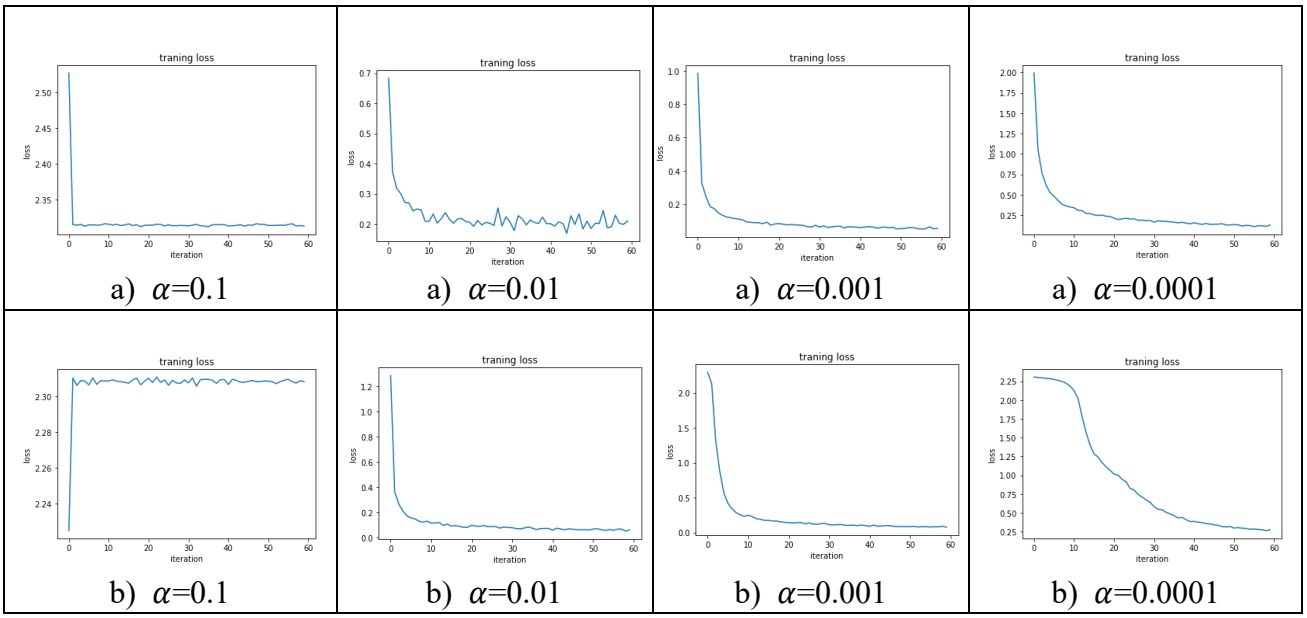
按照上述实验设计，在本课程设计中对学习率、优化器的选择这两个个超参数进行了调参工作。在 epoch 为 10，batch_size 为 32 的情况下，得到在不同参数组合下，训练出来的模型的性能表现以及模型的训练时长，如下表所示：

表 5.1 不同超参数组合下 CNN 的性能表现

优化器	学习率	耗时	精确率	召回率	F1-Score
Adam	0.1	03:24	0.111350	0.100102	0.020590
	0.01	03:19	0.979034	0.975318	0.975574
	0.001	03:17	0.991291	0.991331	0.991293
	0.0001	03:12	0.982057	0.982309	0.982141
SGD	0.1	03:15	0.111350	0.100102	0.020590
	0.01	03:16	0.989225	0.989405	0.989290
	0.001	03:13	0.985928	0.985272	0.985488
	0.0001	03:14	0.960183	0.959393	0.959675

从上表种可以看出，Adam 优化器的训练时间与 SGD 相差不大，理论上讲，Adam 优化器的运行效率要高于 SGD，但可能是由于 epoch 的轮数不够多，不能体现出 Adam 的优势，SGD 最大的缺点是下降速度慢，而且可能会在沟壑的两边持续震荡，停留在一个局部最优点。当学习率为 0.1 时，可以发现用两个优化器训练出来的模型都没有收敛，精确率仅为 0.11，F1-Score 甚至仅为 0.02，完全不可用。使用 Adam 优化器时，学习率为 0.001 的模型表现最佳；使用 SGD 优化器时，学习率为 0.01 的模型表现最佳。整体来看，使用 Adam 优化器时，学习率为 0.001 的模型表现最佳（蓝色部分），精确率达到 0.991291，召回率达到 0.991331，F1-Score 同样达到 0.991293。模型表现十分不错，各项指标已经突破了 99%。

同时，在训练的过程中还记录下了不同超参数组合下的损失函数值变化曲线，如下图所示：



a) α=0.1　a) α=0.01　a) α=0.001　a) α=0.0001

b) α=0.1　b) α=0.01　b) α=0.001　b) α=0.0001

图 5.1 不同超参数组合下损失函数变化曲线（a 为 Adam，b 为 SGD）

从以上损失函数值变化曲线可以看出，当学习率设置得过小时，SGD 的收敛速度较慢，而 Adam 勉强能够收敛，分析原因，可能是 Adam 算法能够自适应调整学习率。而设置得过大（α = 0.1），算法有可能不完全收敛，损失函数的值一直不变化，如 a(1)、b(1)，可能陷入了局部最优点，甚至损失还向增大的方向移动。因此学习率无论过高还是过低都可能带来问题，设置一个合适的学习率，对于算法的稳定性以及效率都十分重要。

5.2 对比其他机器学习算法

在实验中，分别采用高斯朴素贝叶斯、KNN（K 近邻算法）、逻辑回归算法、SVM（线性核/高斯核）、决策树、随机森林与 CNN 进行泛化性能对比。统一使用 MNIST 数据集划分的训练集与测试集，同时与卷积神经网络模型采用相同的数据预处理方法。除 CNN 外，其他机器学习算法均基于 sklearn 库实现，算法的执行结果如下表所示。

表 5.2 算法泛化性能对比

机器学习模型	运行时间	精确率 Precision	召回率 Recall	F1-Score
高斯朴素贝叶斯	00:03	0.692811	0.557209	0.521213
KNN	01:09	0.986905	0.986528	0.986681
逻辑回归	00:09	0.928134	0.927880	0.927941
SVM（线性核）	02:07	0.888582	0.884402	0.884494
SVM（高斯核）	08:33	0.989923	0.989885	0.989901
决策树（max=15）	00:11	0.984501	0.984272	0.984369
随机森林（n=15）	00:05	0.999735	0.999735	0.999735
LeNet-5（最优情况）	03:16	0.991291	0.991331	0.991293

根据上表，通过可视化手段，可用 Echarts 绘制出如下折线图，方便观察算法模型的效果。

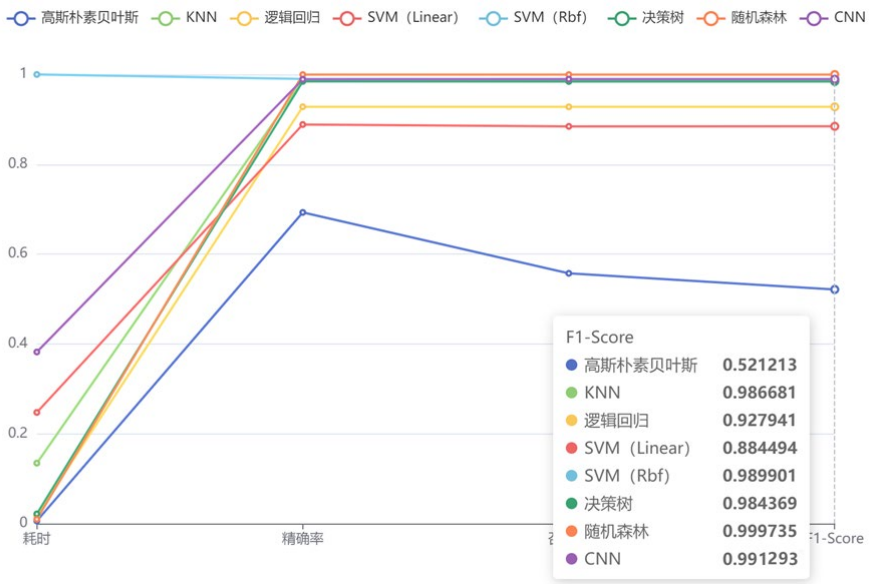


图 5.2 模型效果对比折线图

在这些模型当中，集成学习算法---随机森林的效果最佳，不仅训练的时间极短，而且各项指标都领先于其他算法，尤其在其基学习器---决策树已经达到 98%以上的精确率后，通过随机森林，还能够使得指标再次提升接近一个百分点，体现了**集成学习方法**的引入给指标提升带来的贡献。同时，在平时实验中效果较好的高斯核 SVM 算法，在本次课程设计中的表现仍然十分亮眼，各项指标均接近 99%，但 SVM 算法的训练时间较多，效率较低，这与 SVM 的原理息息相关，在大量数据下计算缓慢。KNN 算法与 SVM 算法的效果接近，同时效率较高。CNN 的模型效果与随机森林、高斯核 SVM 算法十分接近，时间开销明显低于 SVM 算法，但高于随机森林算法，究其原因，神经网络中需要训练的参数太多，且需要计算梯度等值，涉及到求导。其中表现最差的是高斯朴素贝叶斯算法，其 F1-Score 仅有 0.52，识别精确率也仅有 0.69，很难投入实际应用中，究其原因，可能是贝叶斯算法假设属性之间相互独立，然而这个假设在实际应用中往往是不成立的，在属性个数比较多或者属性之间相关性较大时，分类效果不好，这里的图像数据即是属于属性多且属性之间关联性较大的情形，因此朴素贝叶斯算法在此处的效果较差。

6 总结分析

通过完成本次《机器学习实验》课程设计，在课堂学习了神经网络的基础之上，利用 Pytorch 实现了一个基于 LeNet-5 卷积神经网络的手写数字分类器。在完成课程设计中，领略到 CNN 卷积神经网络在图像处理方面相较于全连接网络的优势，卷积神经网络的两大特性---局部连接、权重共享为图像处理提供了新的思路，解决了传统全连接神经网络难以处理图像数据的问题。通过复现 CNN 的开山之作 LeNet-5，通过这个经典的 CNN 网络模型的对 MNIST 手写数字数据集进行模型训练和识别，达到了 99%以上的正确率。同时，在训练神经网络、调整超参数的过程中，更加意识到一个合适的学习率对于算法模型的训练能够起到至关重要的作用。最后在各种模型的对比中，将实验课大部分用到的模型都用来与 CNN 对比，发现即使是传统的机器学习算法，性能表现、时间开销甚至优于卷积神经网络。在深度学习概念非常火热的今天，得到这样的实验结果，仍然让我们感受到了传统机器学习算法的魅力，说明它们还有很大的用武之地，而不应该被人们认为是落后于时代的东西。

7 参考文献

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[2] Tabian, I., Fu, H., & Khodaei, Z. S. (2019). A Convolutional Neural Network for Impact Detection and Characterization of Complex Composite Structures. Sensors, 19(22), 4933. MDPI AG. Retrieved from http://dx.doi.org/10.3390/s19224933.

[3] Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2017). Building machines that learn and think like people. Behavioral and brain sciences, 40.

[4] 张园,王书旺 & 马永兵.(2021).机器和深度学习算法在手写数字识别中的应用比较. 信息化研究(05),60-64.

[5] 汪志成,何坚强,翁嘉鑫 & 苗荣.(2022).基于改进 LeNet-5 的压印字符识别. 计算机仿真(02),441-446.

[6] 谢沛松,胡黄水 & 张金栋.(2021).改进 LeNet-5 网络模型图像分类. 长春工业大学学报(05),455-461. doi:10.15923/j.cnki.cn22-1382/t.2021.5.12.

[7] 肖驰.(2020).四种机器学习算法在 MNIST 数据集上的对比研究. 智能计算机与应用(12),185-188.

8 附录

8.1 Utils 工具代码

```
# 封装了一些工具函数
import os
from sklearn import metrics
import numpy as np
import gzip

# 用于模型泛化性能的评价
# 利用 sklearn 包实现
# 接受的参数为 y_pred（预测标签） y_true（真实值标签）
def model_test(y_pred, y_true):
  accuracy_score = metrics.accuracy_score(y_pred=y_pred, y_true=y_true)
  precision_score = metrics.precision_score(y_pred=y_pred, y_true=y_true,
average="macro")
  recall_score = metrics.recall_score(y_pred=y_pred, y_true=y_true,
average="macro")
  f1_score = metrics.f1_score(y_pred=y_pred, y_true=y_true, average="macro")
  print("Precission:\t"+str(precision_score)+"\nRecall_score:\t"+str(recall_s
core)+"\nF1-score:\t"+str(f1_score))

# 打印分类报告
def model_report(y_pred, y_true):
  print(metrics.classification_report(y_pred, y_true))

# 数据集加载函数，用于加载 MNIST 数据集
def load_data(data_folder):
  files = [
      'train-labels-idx1-ubyte.gz', 'train-images-idx3-ubyte.gz',
      't10k-labels-idx1-ubyte.gz', 't10k-images-idx3-ubyte.gz'
  ]# 原始文件列表
  paths = []
  for fname in files: # 将文件读入内存
    paths.append(os.path.join(data_folder,fname))
  with gzip.open(paths[0], 'rb') as lbpath: # 读入训练集 label
    y_train = np.frombuffer(lbpath.read(), np.uint8, offset=8)
  with gzip.open(paths[1], 'rb') as imgpath: # 读入训练集 image
    x_train = np.frombuffer(
        imgpath.read(), np.uint8, offset=16).reshape(len(y_train), 28, 28)
  with gzip.open(paths[2], 'rb') as lbpath: # 读入测试集 label
```

```
    y_test = np.frombuffer(lbpath.read(), np.uint8, offset=8)
  with gzip.open(paths[3], 'rb') as imgpath: #读入测试机 image
    x_test = np.frombuffer(
        imgpath.read(), np.uint8, offset=16).reshape(len(y_test), 28, 28)
      return x_train, y_train, x_test, y_test
```

8.2 训练代码

```
import torch
import torchvision
import matplotlib.pyplot as plt
from torch import optim

import lenet5
import utils

# 调用 GPU 模块
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# 返回指定 batch_size 的训练数据
def get_train_loader(batch_size=32):
  train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('Data/', train=True, download=True,
                               transform=torchvision.transforms.Compose([
                                 # 数据增强
                                 torchvision.transforms.RandomAffine(degrees=0,
translate=(0.1, 0.1)),
                                 torchvision.transforms.RandomRotation((-10,
10)),  # 将图片随机旋转（-10,10）度
                                 torchvision.transforms.ToTensor(),
                                 # 标准化
                                 torchvision.transforms.Normalize(
                                   (0.1307,), (0.3081,))
                               ])),
    batch_size=batch_size, shuffle=True)
  return train_loader


def print_loss_curve(loss_list):
  plt.plot(loss_list)
  plt.title('traning loss')
  plt.xlabel('iteration')
```

```
  plt.ylabel('loss')
  plt.show()
  pass


# 保存模型
def save_model(path, model):
  torch.save(model.state_dict(), path)


def train(model, train_loader, epoch=5, lr=0.001):
  optimizer = optim.SGD(
    model.parameters(),
    lr=lr,
    momentum=0.9
  )
  # 交叉熵损失函数
  loss_function = torch.nn.CrossEntropyLoss()
  loss_list = []
  for epoch in range(epoch):
    running_loss = 0.0
    for batch_idx, (images, labels) in enumerate(train_loader, start=0):
      # 使用GPU
      images, labels = images.to(DEVICE), labels.to(DEVICE)
      # 读取一个batch的数据
      optimizer.zero_grad()  # 梯度清零，初始化
      outputs = model(images)  # 前向传播
      loss = loss_function(outputs, labels)  # 计算误差
      loss.backward()  # 反向传播
      optimizer.step()  # 权重更新
      running_loss += loss.item()  # 误差累计
      # 每300个batch 打印一次损失值
      if batch_idx % 300 == 299:
        print('epoch:{} batch_idx:{} loss:{}'
              .format(epoch + 1, batch_idx + 1, running_loss / 300))
        loss_list.append(running_loss / 300)
        running_loss = 0.0  # 误差清零
  print('Finished Training.')
  print_loss_curve(loss_list)


if __name__ == '__main__':
```

```
  # 实例化网络
  model = lenet5.LeNet5().to(DEVICE)
  # 获得训练数据
  train_loader = get_train_loader()
  # 进行训练
  train(model, train_loader, 5, 0.1)
  # 保存参数
      save_model("Model/model.pth",model)
```

8.3 测试代码

```
# 加载指定位置的模型
import numpy as np
import torch
import torchvision

import lenet5
import utils

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# 返回指定batch_size 的测试数据
def get_train_loader(batch_size):
  test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('data/', train=False, download=True,
                               transform=torchvision.transforms.Compose([
                                 torchvision.transforms.ToTensor(),
                                 # 标准化
                                 torchvision.transforms.Normalize(
                                   (0.1307,), (0.3081,))
                               ])),
    batch_size=batch_size, shuffle=True)
  return test_loader


# 从文件路径读取模型,并返回
def load_model(path):
  cnn = lenet5.LeNet5().to(DEVICE)
  cnn.load_state_dict(torch.load(path))
  return cnn
```

```
def predict(test_loader, model):
  y_pred = np.array(0)
  y_label = np.array(0)
  for data in test_loader:
    images, labels = data
    images, labels = images.to(DEVICE), labels.to(DEVICE)
    outputs = model(images)
    _, predict = torch.max(outputs.data, 1)
    #
    y_pred = np.append(y_pred, (predict.cpu().detach().numpy()))
    y_label = np.append(y_label, (labels.cpu().detach().numpy()))
  # 返回预测结果
  return y_pred, y_label


if __name__ == '__main__':
  # 加载模型
  cnn = load_model("Model/sgd-0.1.pth")
  # 获取测试集
  test_loader = get_train_loader(32)
  # 进行预测
  y_pred, y_label = predict(test_loader, cnn)
  # 模型评价
  utils.model_test(y_pred, y_label)
  utils.model_report(y_pred, y_label)
```