基于支持向量机和神经网络的手写数字识别

摘要

本文利用 scikit-learn 库中的 digits 数据集，探究了手写数字识别问题。通过将手写数字识别问题视为一个十分类问题（分别是 0 到 9 十个类别），构建了三种不同的分类器：支持向量机（SVM）、全连接神经网络（DNN）和卷积神经网络（CNN）。

在支持向量机分类器的实现中，对数据特征进行了标准化处理，并使用了 20%的数据作为测试集。经过训练，SVM 分类器在测试集上达到了 98%的准确率。

对于全连接神经网络分类器，首先将数据标签转换为独热编码。随后，设计了一个包含 64 个输入神经元、两个隐藏层（分别有 50 和 25 个神经元）以及 10 个输出神经元的 DNN。该网络在测试集上实现了 97%的准确率。

卷积神经网络分类器的实现中，将数据标签还原为 8x8 的矩阵格式。构建了一个包含 8x8 输入层、两个卷积层、两个池化层、512 个神经元的全连接层以及 10 个神经元的输出层的 CNN。该网络在测试集上取得了 99%以上的准确率。（仅有两个数据分类错误）

在构建神经网络时，除输出层使用 SoftMax 激活函数外，其他层均使用 ReLU 激活函数。为了减轻过拟合，在非输出层的全连接层后引入了 Dropout 层。此外，对于全连接神经网络，还额外实施了正则系数为 0.001 的正则化策略。

关键词：手写数字识别，支持向量机，神经网络

目录

一、 问题分析

手写数字识别是一个非常经典的机器学习问题。其主要任务是，给定一个有$m \times n$灰度像素的图片判断其上的字为数字几。

由于阿拉伯数字只有 0 到 9 十种可以所以该类识别任务可以归结为将说给图片分类到 0 到 9 十个类别中的哪一类。

而$m \times n$像素的灰度图片一个像素点只有一个数字信息，图中总共有mn个像素点，可以直接将则些像素点对应的数字信息当做特征。

因此，手写数字识别问题就化归为了由mn个特征将数据分为十类的多分类问题。因而可以使用常见的多分类算法解决例如支持向量机、全连接神经网络。对于这种图像识别问题，还可以使用卷积神经网络的方式解决。

二、 算法理论介绍

2.1 支持向量机分类器（SVC）

支持向量机（SVM）是一种监督学习算法，用于分类任务的支持向量机也被称为 SVC。SVM 的目标是找到一个最优的分类决策边界，将不同类别的数据点尽可能最好地分开。在二维空间中，它是一条直线；在三维空间中，它是一个平面。在更高维的空间中，超平面是一个多维的超平面。

SVM 的核心思想就是找到最大间隔的超平面。这个间隔是指超平面到最近的训练数据点（称为支持向量）的距离，而最大化间隔可以尽可能的提高模型的泛化能力。

2.1.1 数学表达

假设我们有一个训练集$\{(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})\}$,其中$x^{(i)}$是输入特征，$y^{(i)}$是对应的标签，取值为 +1 或 -1。

对于线性可分的数据集，超平面可以表示为：$\omega^T x + b = 0$,其中，ω是超平面的法向量，它决定了超平面的方向。b 是位移项，它决定了超平面与原点的距离。

对于每个支持向量$x^{(i)}$，如果它对应正例则应该有$\omega^T x^{(i)} + b \geq 1$，反之负例就有$\omega^T x^{(i)} + b \leq -1$。也就是所有正例都位于超平面的同一侧，所有负例都位于超平面的另一侧。据此还可以将约束条件统一为$y^{(i)}(\omega^T x + b) \geq 1$。

而支持向量机需要找的目标函数为$min_{\omega,b} \frac{1}{2} ||\omega||^2$。

这个优化问题本身较为复杂，但可以先解其拉格朗日对偶问题。因为其拉格朗日对偶问题是一个二次优化问题，可以用二次规划的算法解决，但除了常规的二次规划解法外支持向量机问题可以用专门的算法更快更好地解决，它被称为序列最小优化（Sequential Minimal Optimization, SMO）算法。

前面谈的是用线性超平面分割，然而对大多数问题这是难以实现的。因为大多数问题必须要使用非线性决策边界才行，为此需要引入核函数来解决。

2.1.2 核方法

常见的核函数如下，

线性核（Linear Kernel）：$K(x_i, x_j) = x_i * x_j$它适用于线性可分的数据集。

多项式核（Polynomial Kernel）：$K(x_i, x_j) = (x_i * x_j + 1)^d$通过计算两个向量的多项式函数来映射数据。

高斯径向基函数核（Gaussian Radial Basis Function）：$K(x_i, x_j) = e^{-\gamma||x_i - x_j||^2}$它也被称为高斯核，它通过计算两个向量之间的欧氏距离来映射数据。然而 RBF 核函数的γ是一个超参数，需要手动给出。

但是在本文这并不是重点因此只介绍到这里为止，这是因为本文在实现的时候直接利用了 sk-learn 库的 SVC 类实现，这些数学推导已经在 SVC 类中实现了。

2.2 全连接神经网络（DNN）

全连接神经网络（Fully Connected Neural Network），也称为密集连接神经网络（Dense Neural Network），是一种基本的神经网络结构，其中每个神经元都与前一层的所有神经元相连接。

2.2.1 结构特点

全连接神经网络的主要特点包括：

层与层之间的全连接：每一层的每个神经元都连接到上一层的所有神经元。

参数数量：由于每个连接都有权重，因此全连接网络的参数数量非常多，尤其是在输入层和隐藏层之间。

激活函数：每个神经元通常会使用非线性激活函数，如 Sigmoid、ReLU 或 Tanh，以引入非线性特性。

输出层：输出层的神经元数量取决于任务类型。例如，在多分类问题中，通常有一个输出神经元对应于每个类别。

2.2.2 工作原理

全连接神经网络整体的工作原理如下：

前向传播：输入数据经过权重和偏置的线性变换，然后通过激活函数进行非线性处理。这个过程从输入层开始，逐层传递到输出层。

损失函数：输出层的激活值与真实标签之间的差异通过损失函数来衡量，常见的损失函数有均方误差（Mean Squared Error, MSE）$MSE = \frac{1}{N}\sum_{i=1}^{n}(Y_i - \widehat{Y_l})^2$和交叉熵（Cross-Entropy）$CE = -\sum_i \sum_j Y_{i,j} \ln(\widehat{Y_{l,J}})$。

反向传播：通过计算损失函数关于网络参数的梯度，使用梯度下降法或其他优化算法更新网络的权重和偏置。

权重更新：权重更新的目标是减小损失函数的值，从而使网络输出更接近真实标签。

全连接神经网络的单个神经元的工作原理是，先从接收其连接的上层神经元的输出，将其与对应权值相乘，再加上偏置值。再将输入值传入激活函数得到对应的输出并传递给下一层神经元。

2.2.3 训练过程

全连接神经网络的训练过程通常包括以下几个步骤：

数据预处理：对输入数据进行归一化或标准化处理。

模型构建：定义网络的层数、每层的神经元数量和激活函数。

参数初始化：随机初始化网络的权重和偏置。

前向传播：根据当前权重和偏置，计算网络输出。

计算损失：使用损失函数计算网络输出与真实标签之间的差异。

反向传播：计算损失函数关于网络参数的梯度。

权重更新：根据梯度下降法或其他优化算法更新网络的权重和偏置。

重复：重复步骤 4 到 7，直到满足停止条件。

2.3 卷积神经网络（CNN）

卷积神经网络在大题上与全连接神经网络类似。但是额外多了卷积层和池化层。

2.3.1 层次结构

卷积层（Convolutional Layer）：卷积层通过使用卷积核（或过滤器）在输入数据上滑动，以提取特征。每个卷积核会生成一个特征图（feature map），表示它在输入数据上检测到的特定特征。

激活函数：为了引入非线性因素，通常在卷积层之后会应用激活函数，如 ReLU 等。

池化层（Pooling Layer）：池化层用于降低特征图的维度，同时保留最重要的信息。最常用的池化方法是最大池化（max pooling）。

全连接层（Fully Connected Layer）：在网络的最后几层通常是全连接层，它们将卷积和池化层提取的特征进行整合，并输出最终的分类结果。

2.3.2 工作原理

CNN 的工作原理可以概括为以下几个步骤：

卷积：在输入图像上应用卷积核，通过卷积操作提取特征。

激活：使用激活函数对卷积层的输出进行非线性变换。

池化：降低特征图的维度，同时保留最重要的信息。

全连接：将池化层输出的特征进行整合，并输出最终的分类结果。

优化：通过反向传播和梯度下降等算法优化网络的权重。

2.4 减少过拟合手段

过拟合通常发生在模型过于复杂，以至于学到了训练数据中的噪声和特异性，从而导致模型在未知数据上的泛化能力下降。正则化通过对模型的复杂性施加惩罚来鼓励模型学习更加泛化的模式。

2.4.1 丢弃层（Dropout Layer）

为了减少过拟合现象，可以在神经网络中使用丢弃层，在训练过程中随机丢弃一部分神经元。这里的“丢弃”指的是在一次更新时随机选择一部分的神经元不进行更新，但在最终应用网络模型的时候这些神经元仍然会参与计算。

2.4.2 正则化

正则化与前面提到的丢弃层有明显的不同，因为丢弃层是神经网络专门的方法而正则化却是一种在各种机器学习模型中都可以用来防止过拟合的技术。

L1 正则化（Lasso 正则化）

L1 正则化通过添加模型权重向量的绝对值之和的惩罚项来实现的。数学上，L1 正则化的目标函数可以表示为：

$$J(\theta) = Loss(h_\theta(x), y) + \lambda \sum_{i=1}^{n} |\theta_i|$$

其中，θ是模型参数，λ是正则化系数，n 是参数的数量，Loss 是原本的损失函数。

L1 正则化的特点是可以产生稀疏的权重矩阵，甚至可能会使某些特征的权重会变为 0，从而实现特征选择。

L2 正则化（Ridge 正则化）

L2 正则化通过添加模型权重向量的平方之和的惩罚项来实现。数学上，L2 正则化的目标函数可以表示为：

$$J(\theta) = Loss(h_\theta(x), y) + \frac{\lambda}{2n} \sum_{i=1}^{n} \theta_i^2$$

其中，θ是模型参数，λ是正则化系数，n 是参数的数量，Loss 是原本的损失函数。在这里为λ除以 2n 是为了下一步进行求导的数学推导方便，并不是必须的。

L2 正则化使得权重矩阵中的权重普遍较小，但不会产生稀疏性。

2.5 优化方法

2.5.1 梯度下降 GD

梯度下降（Gradient Descent）是一种用于优化机器学习模型参数的算法，其目标是找到能够最小化损失函数的参数值。它的基本思想是沿着损失函数的负梯度方向更新参数，从而逐渐减少损失函数的值。具体来说，对于参数θ，梯度下降算法更新参数的方式是：

$$\theta_{n+1} = \theta_n - \eta \nabla_\theta J(\theta)$$

其中，$J(\theta)$是损失函数，$\nabla_\theta J(\theta)$表示损失函数关于参数θ的梯度，η是学习率，控制每次迭代中参数更新的步长。

2.5.2 均方根传播 RMSprop

RMSprop（Root Mean Square Propagation）是一种自适应学习率优化算法，它由 Geoffrey Hinton 和 Tijmen Tieleman 在 2012 年提出。相比于其他自适应学习率优化算法，它可以更有效解决地处理稀疏数据时学习率下降过快的问题。相比于随机梯度下降算法 SDG 或其他梯度下降算法，它则可以自适应地改变学习率，往往会取得更好的解。

RMSprop 的更新公式如下：

$$s_{n+1} = \beta s_n + (1-\beta)\left(\nabla_\theta J(\theta)\right)^2$$

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{s+\varepsilon}} \nabla_\theta J(\theta)$$

其中，$J(\theta)$是损失函数，$\nabla_\theta J(\theta)$表示损失函数关于参数θ的梯度，η是初始学习率（注意，不是真实的学习率），衰减系数β一般取 0.9。s是中间参数，ε是为了防止分母为 0 的小数。

2.5.3 适应性矩估计 Adam

Adam（Adaptive Moment Estimation）是一种自适应学习率优化算法，它是动量优化器和 RMSProp 算法的结合。Adam 由 Diederik P. Kingma 和 Jimmy Ba 于 2015 年提出，旨在解决 AdaGrad 算法在处理稀疏数据时学习率过快下降的问题，但却引入了更复杂的参数更新机制。

Adam 的更新公式如下：

$$m_{n+1} = \beta_1 m_n - (1-\beta_1)\nabla_\theta J(\theta)$$

$$s_{n+1} = \beta_2 s_n + (1-\beta_2)\left(\nabla_\theta J(\theta)\right)^2$$

$$\hat{m} = \frac{m}{1-\beta_1^t}$$

$$\hat{s} = \frac{s}{1-\beta_2^t}$$

$$\theta_{n+1} = \theta_n + \eta * \frac{\hat{m}}{\hat{s}+\varepsilon}$$

其中，$J(\theta)$是损失函数，$\nabla_\theta J(\theta)$表示损失函数关于参数θ的梯度，η是初始学习率（注意，不是真实的学习率），衰减系数有两个分别是β_1和β_2。s和 m 都是中间参数，ε是为了防止分母为 0 的小数，t 是从 1 开始计的循环的次数。

三、 数据集介绍

本文使用的是 scikit-learn 内置的 digits 数据集。该数据集是一个手写数字数据集。它可以直接通过 datasets 模块加载。数据集的结构主要包括以下几个部分：

• 以向量形式存储的特征集合

由于该数据集存储的是 8x8 像素的手写图片，所以特征集合的向量是一个 64 维的向量。向量中的数字则是由 0 到 16 的整数，用来表达像素的灰度信息。而数据集中一共有 1797 张图片，因此它一共有 1797 个向量。

例如，以下图片就是其中的一个数据。当然，右侧的图片就是由原来的向量经过处理上色得来的。

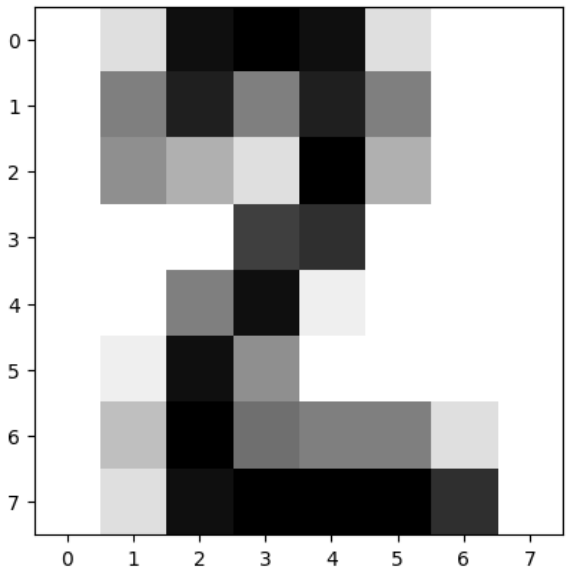

图 1 数据集演示图

• 以数字形式存储的标签集合

它包含了每个样本的标签，即 0 到 9 的数字，并且是直接以数字形式表示。

• DESCR 字符串

这是一个字符串，提供了数据集的描述信息。本文说的数据集信息都可以从 DESCR 字符串中得到。

• 数据集来源

本文所用的 digits 数据集来自于 Modified National Institute of Standards and Technology (NIST)数据库。这个数据库是一个得到广泛使用的手写数字数据库，通常用于机器学习和模式识别的实验。

四、 代码实现

4.1 支持向量机的实现

4.1.1 数据导入和预处理

到了代码实现层，首先要做的就是导入数据。本文所使用的数字手写体数据集恰为 Scikit-learn 库内置的 digits 数据集，因此直接使用 sk-learn 库中的 load_digits()函数即可导入。为了方便起见，还需要将数据的特征和标签进行分离，为此可以设置 load_digits()函数的形参 return_X_y 为 True。

即导入数据的代码仅为

```
#载入数据
from sklearn.datasets import load_digits

x, y = load_digits(return_X_y=True)
```

接着需要划分训练集与测试集。在此可以直接利用 sk-learn 库的 train_test_split()函数实现。

本文约定，训练集占数据集的 80%，测试集占数据集的 20%，因此设定 train_test_split()函数的形参 test_size 为 0.2。另外，为了使得本文的结果便于读者复现，在此额外设置随机种子 random_state 为 202406。

因此，划分训练集与测试集的代码为

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=202406)
#划分训练集与测试集
```

以上的代码在接下来模型中同样使用了，后面便不再赘述。

对于支持向量机分类器来说，为了提高该分类器的稳定性还需要额外对数据集进行标准化操作。标准化后的数据$x^* = \frac{x-E(x)}{\sqrt{D(x)}}$，其中 E(x)表示 x 的平均值，D（x）表示 x 的方差或者说标准差的平方。通过将数据进行标准化可以保证数据的各个维度的均值为 0 并且方差为 1，从而避免分类结果被某些维度中过大的数据主导，进而提高分类的精准性。

Sk-learn 库中提供了专门用于标准化的工具类 StandardScaler，可以借此完成数据的标准化。

```
from sklearn.preprocessing import StandardScaler

#进行标准化操作
scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
#使用训练数据的参数进行标准化
x_test_scaled = scaler.transform(x_test)
```

这里首先实例化了一个 StandardScaler 对象 scaler，然后可以利用 scaler 对象的 fit_transform 方法进行标准化。值得注意的一点是，为了保证训练集和测试集的数据分布一致，测试集的标准化参数需要使用训练集得到。

4.1.2 模型建立与训练

前文已经介绍了支持向量机分类器模型的原理在此便不再赘述。事实上，sk-learn 库以 SVC 对象的形式直接提供了支持向量机分类器。至于核函数的选择，本文根据能简就简的原则先使用了线性核函数，最终发现效果已经非常令人满意便不再使用更为复杂的核函数了。因此使用以下语句便可以构建支持向量机分类器。

```
from sklearn.svm import SVC

#构建和训练模型
classifier = SVC(kernel='linear', random_state=202406)
```

至于模型的训练，直接使用 sk-learn 默认的学习率等参数就已经可以得到令人满意的结果了，因此本文并没有进行额外的设置，就只是把训练集及其标签传入了而已。

```
    classifier.fit(x_train_scaled, y_train)
```

4.2 全连接神经网络的实现

4.2.1 数据预处理

根据全连接神经网络模型的特点，除了常规的导入数据划分训练集和测试集之外，还需要对标签进行独热编码。

所谓独热编码就是将多分类问题需要的分类转化为向量的方法。对于手写数字识别来说，总共有 0 到 9 十种类型，所以可以将分类标签转化为 10 维向量，即用第 i 行为 1 其余各行均为 0 的 10 维列向量表示表示标签 i-1。

举例来说，如果独热向量的第一行是 1，其余行是 0 就表示该图片的标签是 0。类似的，如果独热向量的第一行是 5，其余行是 0 就表示该图片的标签是 4。（注意，这里的行的下标是从 1 开始的）

使用独热编码一是与全连接神经网络的结构相适应，二是他避免了直接使用 1,2,3 这样大小和顺序关系的数字带来的误解。所幸，TensorFlow 提供的 keras API 直接提供了将标签进行独热编码的函数 to_catagorical()。

在使用如下代码导入并划分训练集和数据集后，

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

x, y = load_digits(return_X_y=True)   #载入数据
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=202406)   #划分训练集与测试集
```

直接使用该函数即可对标签进行独热编码。

```
from keras.utils import to_categorical

#对标签进行独热编码
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

4.2.2 网络结构的定义

由于数据集中的图片是 8*8 总共 64 像素的图片，因此网络的输入层需要有 64 个神经元。至于网络的输出层，由于最终是 10 个类别的多分类问题，所以只需要设置 10 个神经元作为输出层，每个神经元输出属于对应类别的概率即可。

至于激活函数的选择，由于输出层输出的是概率信息，所以选择 SoftMax 函数。而输入层和隐藏层的激活函数则可以选择 ReLU 函数，这是因为 ReLU 函数相比于 sigmoid 或 tanh 等函数计算速度快并且不存在梯度消失的问题，是一个极为优质的选择。

关于隐藏层的结构和内部神经元的数量，并不存在确切的理论可以直接得到，本文经过尝试最终决定使用两层隐藏层。第一层隐藏层设置 50 个全连接神经元，第二层 25 个。

然而在实际训练的时候发现模型出现了过拟合现象，为此还需要进行正则化。经过反复测试，可以使用 L1 正则化的方式，并且在各个隐藏层之后加入了 Dropout 层。

综合上述对网络结构的分析，可以用如下代码建立全连接神经网络。

```
from keras import models, layers, regularizers

#定义网络结构
DNN = models.Sequential()
#定义各个全连接层
DNN.add(layers.Dense(50, activation='relu', input_shape=(64,),
kernel_regularizer=regularizers.l1(0.001)))
DNN.add(layers.Dropout(0.001))
DNN.add(layers.Dense(25, activation='relu',
kernel_regularizer=regularizers.l1(0.001)))
DNN.add(layers.Dropout(0.001))
#输出层选择 softmax 函数为激活函数
DNN.add(layers.Dense(10, activation='softmax'))
```

在这里，DNN 是表示该全连接神经网络的对象，通过它的 add()方法就可以为其增加各层。

layers.Dense()表示添加的是全连接层，其第一个参数的该层神经元的个数。activation 参数可以用于设置激活函数。input_shape 参数则表示网络输出层的数量，只需要第一个隐藏层填写即可。kernel_regularizer 参数则是设置正则项的参数，regularizers.l1()表示进行 L1 正则化，0.001 就是正则化系数。

类似的，Layers.Dropout()表示添加的是 Dropout 层，0.001 是 Dropout 系数，表示在训练时需要让 0.1%的神经元“失去作用”。

前面代码的展示或许还不能直观地表达网络的结构，可以利用 keras 提供的 plot_model 绘制网络结构图，具体代码如下：

```
from tensorflow.keras.utils import plot_model
import matplotlib.pyplot as plt

#绘制网络结构示意图
plot_model(DNN, to_file='DNN.png', show_shapes=True,
show_layer_names=False,rankdir='TB')
plt.figure(figsize=(10, 10))
img=plt.imread('DNN.png')
plt.imshow(img)
plt.axis('off')
plt.show()
```

便可以得到这样的网络结构示意图

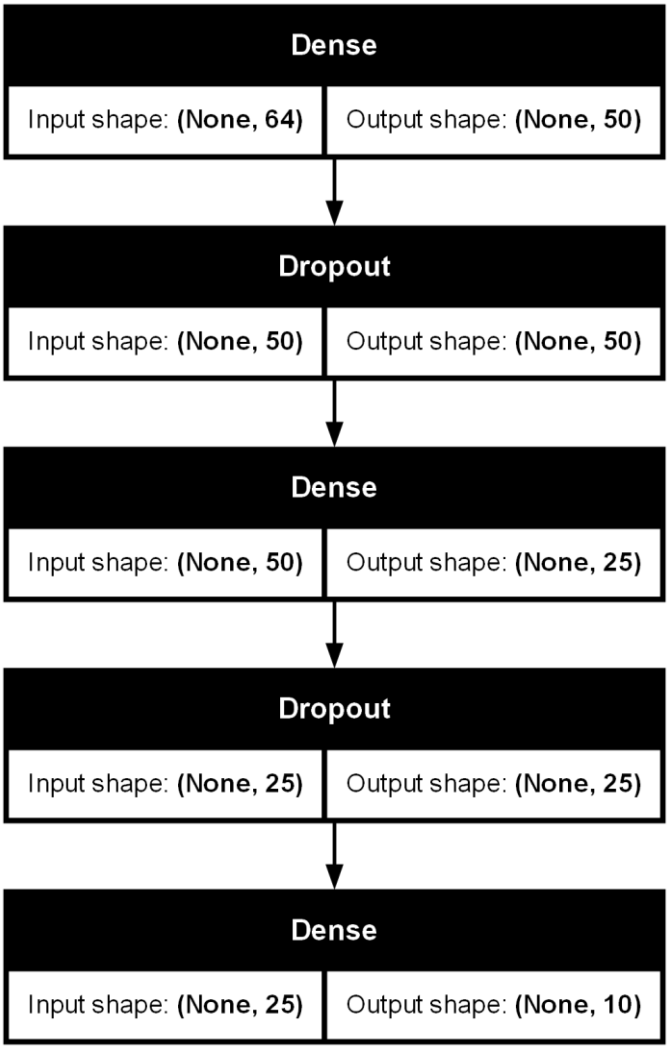

图 2 DNN 网络结构示意图

4. 2. 3 网络训练

到了网络训练阶段，需要给出学习率等参数。具体代码如下：

```
from keras.optimizers import RMSprop

#设置训练参数并进行训练
#设置学习率为 0.001
DNN.compile(optimizer=RMSprop(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])
#进行 20 次训练，每批 64 个数据
DNN.fit(x_train, y_train, epochs=20, batch_size=64)
```

在这里，对于全连接神经网络 DNN.compile()的 优化器 optimizer 参数需要设置成前文提到的 RMSprop。而 learning_rate 就是设置学习率的地方，本文设置为 0.001。loss 是设置损失函数的，对于多分类问题需要设置为交叉熵损失函数 categorical crossentropy，而 metrics 是设置模型评估所使用的，在此由于本文后续不使用 keras 库进行模型评估所以就只在这里设置了准确率 accuracy。至此，网络的全部参数就已设置完毕，接下来是设置训练参数的。

在 DNN.fit()先是传入了训练集的特征和标签，然后设置要进行 20 次训练 epochs=20，每批的大小为 64，batch_size=64。

4.3 卷积神经网络的实现

4.3.1 网络结构的定义

卷积神经网络的数据预处理和全连接神经网络基本一致，在此直接给出代码。

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

x, y = load_digits(return_X_y=True)   #载入数据
x = x.reshape(-1, 8, 8, 1)
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=202406)   #划分训练集与测试集

from keras.utils import to_categorical

#对标签进行独热编码
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

可以看到，这与全连接神经网络的区别就在于第五行多了一句

x = x.reshape(-1,8,8,1)

这是为了将原来是向量的数据特征先转变为矩阵便于后续进行卷积和池化操作。也就是说，在卷积层和池化层数据特征是矩阵，而在后续的全连接层则应当是向量才行，所以在最后一个池化层和第一个全连接层直接有一个扁平化层，可以通过CNN.add(layers.Flatten())加入。

第一个卷积层经过测试，可以设置为有 32 个 3x3 的卷积核，步长为 1，激活函数仍然使用 ReLU 函数，卷积 padding 则使用 same。添加的方法在语句中便不难看出。

```
CNN.add(layers.Convolution2D(input_shape=(8, 8, 1), filters=32,
kernel_size=3, strides=1, padding='same',activation='relu'))
```

而第一个池化层则大小为 2，步长也为 2，仍然选用 same padding。

此外还需要第二个卷积层，有 64 个步长为 1 的 3x3 卷积核，第二个池化层，步长和大小以及 padding 不变。

至于全连接层，则直接使用 512 个神经元，ReLU 函数为激活函数，另外设置系数为 0.5 的 Dropout 层。

输出层则还是 10 个神经元，选用 softmax 函数作为激活函数。

由于该卷积神经网络的性能已经较为良好不需要再进行额外的正则化。以下是最终的定义代码

```
from keras import models, layers

#定义网络结构
CNN = models.Sequential()

#定义卷积层
#该有 32 个 3x3 的卷积核，步长为 1
CNN.add(layers.Convolution2D(input_shape=(8, 8, 1), filters=32,
kernel_size=3, strides=1, padding='same',activation='relu'))
#定义池化层
CNN.add(layers.MaxPool2D(pool_size=2, strides=2, padding='same'))
CNN.add(layers.Convolution2D(filters=64, kernel_size=3, strides=1,
padding='same', activation='relu'))
```

```
CNN.add(layers.MaxPool2D(pool_size=2, strides=2, padding='same'))
#扁平化
CNN.add(layers.Flatten())
#全连接层
CNN.add(layers.Dense(units=512, activation='relu'))
CNN.add(layers.Dropout(0.5))
CNN.add(layers.Dense(units=10, activation='softmax'))
```

在这里，可以用与全连接神经网络类似的方法绘制网络结构示意图。

```
from tensorflow.keras.utils import plot_model
import matplotlib.pyplot as plt

#绘制网络结构示意图
plot_model(CNN, to_file='CNN.png', show_shapes=True,
show_layer_names=False,rankdir='TB')
plt.figure(figsize=(10, 10))
img=plt.imread('CNN.png')
plt.imshow(img)
plt.axis('off')
plt.show()
```

得到如下的示意图。

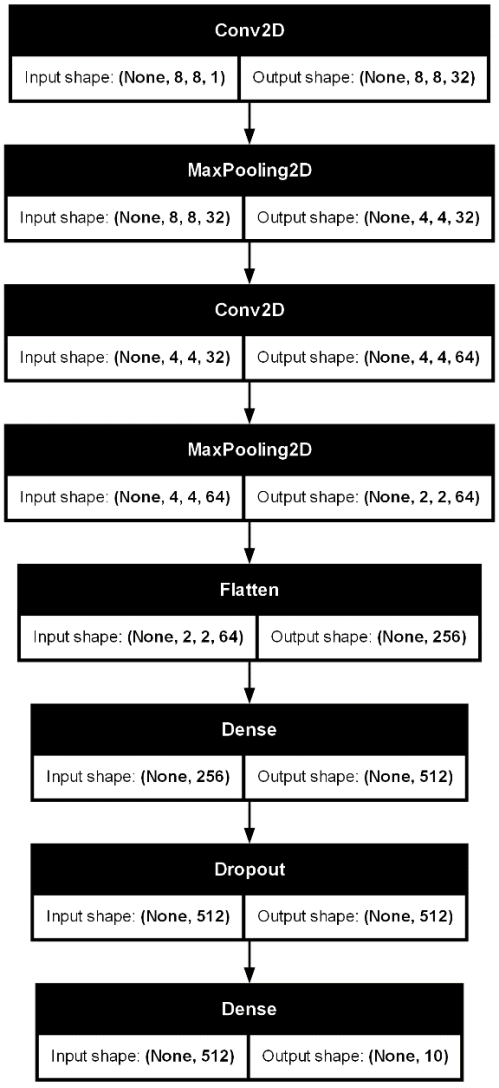

图 3 CNN 网络结构示意图

4.3.2 网络训练

网络训练和全连接神经网络并没有特别大的区别，还是进行 20 次训练，每批 64 个数据。

但是卷积神经网络的优化器需要设置成 Adam，学习率则还是保持 0.001 不变。

因此最终的代码和全连接神经网络并无太大的区别。

```
from keras.optimizers import Adam

#设置训练参数并进行训练
#设置学习率为 0.001
CNN.compile(optimizer=Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])
#进行 20 次训练，每批 64 个数据
CNN.fit(x_train, y_train, epochs=20, batch_size=64)
```

五、 运行结果分析

多分类问题的评估参数有准确率 acc、召回率 recall、F1 值、混淆矩阵等，这些都可以由 sk-learn 的 classification_report 分类报告得出。

准确率 acc 的计算公式为$acc = \frac{\text{准确分类的样本数}}{\text{全部样本数}}$，而$recall = \frac{\text{预测为正样本数}}{\text{全部正样本数}}$，F1 值则是 acc 和 recall 的调和平均$F1 = \frac{2}{\frac{1}{acc}+\frac{1}{recall}}$。

对于支持向量机分类器来说，可以用如下代码得到

```
from sklearn.metrics import classification_report, confusion_matrix

#进行模型评估
y_pred = classifier.predict(x_test_scaled)
acc = classifier.score(x_test_scaled, y_test)
print(f"准确率:{acc * 100:.2f}%")
print("-----分类报告如下-----")
print(classification_report(y_test, y_pred))
print("混淆矩阵如下")
print(confusion_matrix(y_test, y_pred))
```

为了让混淆矩阵更直观还可以用如下代码绘制热力图。

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

# 绘制混淆矩阵的热力图
plt.figure(figsize=(10, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=np.arange(10), yticklabels=np.arange(10))
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

最终可以得到支持向量机的分析数据如左下：

表 1 SVM 分类报告

数字	准确率	召回率	F1 值
0	100%	100%	1
1	94%	100%	0.97
2	98%	100%	0.99
3	97%	97%	97%
4	100%	100%	100%
5	97%	97%	97%
6	100%	97%	99%
7	97%	97%	97%
8	97%	95%	0.96
9	100%	97%	0.98
宏平均	98%	98%	98%
权平均	98%	98%	98%
总准确率		98%	

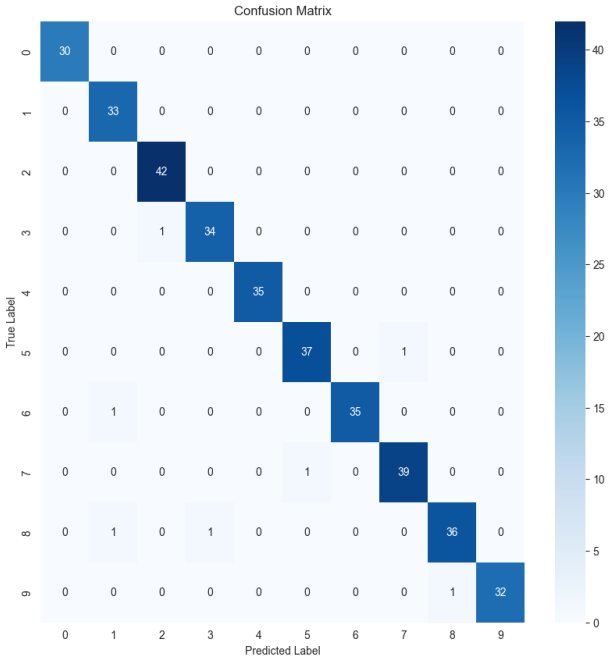

图 4 SVM 分类结果热力图

而由混淆矩阵得到的热力图如右上：

虽然本文在划分训练集与测试集时设置了固定的随机数种子码，但是由于 DNN 和 CNN 中均设置了 Dropout 层，每次都次迭代计算都会有随机的一批神经元没更新，所以两个神经网络的运行结果仍然具有一定的随机性。并且由于训练的时候有一部分神经元没有工作因此即便是训练集运行测试时的准确率也会同训练时不一致。

对于两个神经网络的分析，方法是同支持向量机一致的。

可以用如下代码获得分析报告（对 CNN 网络只需把代码中的模 DNN 换成 CNN）

```
from sklearn.metrics import classification_report, confusion_matrix

#进行模型评估
train_loss, train_acc = DNN.evaluate(x_train, y_train, batch_size=64)
print(f"训练集损失函数{train_loss:.4f},准确率{train_acc * 100:.2f}%")
test_loss, test_acc = DNN.evaluate(x_test, y_test, batch_size=64)
print(f"测试集损失函数：{test_loss:.4f},准确率：{test_acc * 100:.2f}%")
y_pred = DNN.predict(x_test).argmax(axis=-1)
y_true = y_test.argmax(axis=-1)
print("-----分类报告如下-----")
print(classification_report(y_true, y_pred))
print("混淆矩阵如下")
cm=confusion_matrix(y_true, y_pred)
print(cm)
print("-----网络报告如下-----")
print(DNN.summary())
```

据此可以得到全连接神经网络的分析结果为:

表 2 DNN 分类报告

数字	准确率	召回率	F1 值
0	100%	97%	0.98
1	92%	100%	0.96
2	100%	100%	1
3	97%	94%	96%
4	97%	94%	96%
5	100%	97%	99%
6	100%	97%	99%
7	95%	97%	96%
8	97%	95%	0.96
9	94%	100%	0.97
宏平均	97%	97%	97%
权平均	97%	97%	97%
总准确率		97%	

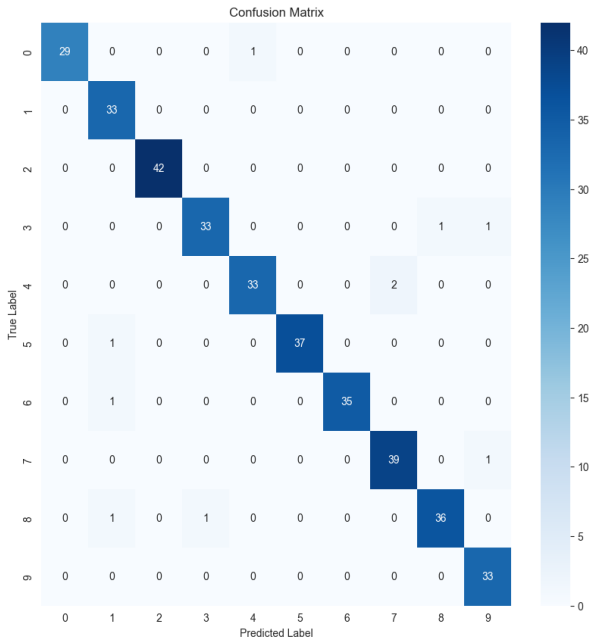

图 5 DNN 分类结果热力图

卷积神经网络为：

表 3 CNN 分类报告

数字	准确率	召回率	F1 值
0	100%	97%	0.98
1	100%	100%	1
2	100%	100%	1
3	100%	100%	1
4	95%	100%	97%
5	100%	97%	99%
6	100%	100%	1
7	100%	100%	1
8	100%	100%	1
9	100%	100%	1
宏平均	99%	99%	99%
权平均	99%	99%	99%
总准确率		99%	

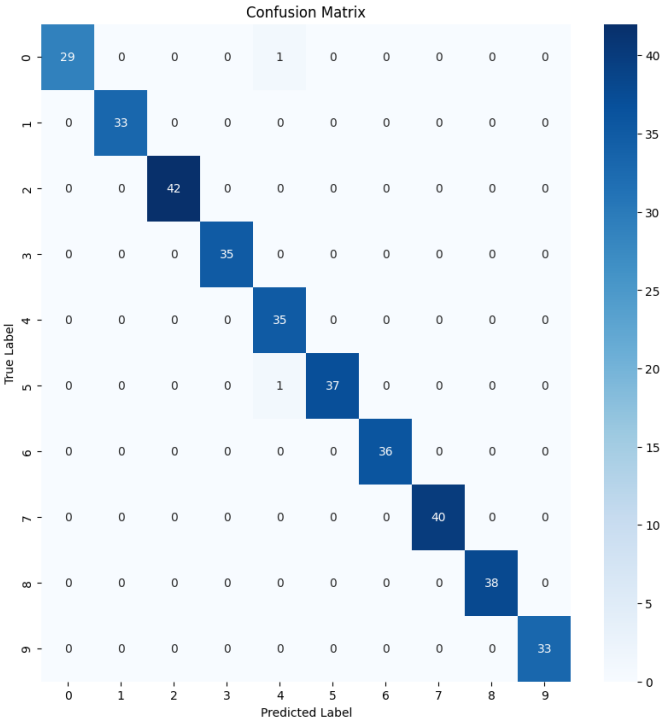

图 6 CNN 分类结果热力图

从这些数据来看，三种分类方法的准确率都在 95%以上。

而全连接神经网络的准确性稍差为 97%，支持向量机为 98%，卷积神经网络则最为优异 99%。事实上，从混淆矩阵热力图中可以看出，卷积神经网络只分类错了 2 个样本数量远远小于其他两种方法。

六、 附录

6.1 附录一：支持向量机分类器实现代码

附录一：支持向量机分类器实现代码

语言：Python
运行环境：
Python 3.12.0
jupyterlab 4.1.6
scikit-learn 1.4.1post1
numpy 1.26.3
matplotlib 3.8.2
seaborn 0.13.2
运行方式：使用 jupyter 分段运行
备注：此处代码由 jupyter 整理而来，并非本文真正使用的代码。真正运行的代码见附件 SVC.ipynb

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.datasets import load_digits
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

# 载入数据
x, y = load_digits(return_X_y=True)
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=202406)  # 划分训练集与测试集

# 进行标准化操作
scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
# 使用训练数据的参数进行标准化
x_test_scaled = scaler.transform(x_test)
print(x_train_scaled.shape, x_test_scaled.shape, y_train.shape,
y_test.shape)

# 构建和训练模型
classifier = SVC(kernel='linear', random_state=202406)
classifier.fit(x_train_scaled, y_train)

# 进行模型评估
y_pred = classifier.predict(x_test_scaled)
acc = classifier.score(x_test_scaled, y_test)
print(f"准确率:{acc * 100:.2f}%")
print("-----分类报告如下-----")
print(classification_report(y_test, y_pred))
print("混淆矩阵如下")
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
# 绘制混淆矩阵的热力图
plt.figure(figsize=(10, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=np.arange(10), yticklabels=np.arange(10))
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# 随机展示一个样例
index = np.random.randint(0, len(x_test))
sample = x_test[index]
print(sample)
sample = sample.reshape(8, 8)
plt.imshow(sample, cmap=plt.cm.gray_r, interpolation='nearest')
plt.title(f"True Label:{y_test[index]},Predicted
Label:{y_pred[index]}")
plt.show()
```

6.2 附录二：全连接神经网络实现代码

附录二：全连接神经网络实现代码
语言：Python 运行环境： Python 3.12.0 jupyterlab 4.1.6 scikit-learn 1.4.1post1 numpy 1.26.3 matplotlib 3.8.2 TensorFlow 2.16.1(CPU 版) Keras 3.3.3 Pydot 2.0.0 Graphviz 0.20.3 h5py 3.11.0 运行方式：使用 jupyter 分段运行 备注：此处代码由 jupyter 整理而来，并非本文真正使用的代码。真正运行的代码见附件 DNN.ipynb

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from keras import models, layers, regularizers
from keras.optimizers import RMSprop
from keras.utils import to_categorical
from sklearn.datasets import load_digits
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import plot_model

x, y = load_digits(return_X_y=True)  # 载入数据
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=202406)  # 划分训练集与测试集
print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
```

```
# 对标签进行独热编码
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# 定义网络结构
DNN = models.Sequential()
# 定义各个全连接层
DNN.add(layers.Dense(50, activation='relu', input_shape=(64,),
kernel_regularizer=regularizers.l1(0.001)))
DNN.add(layers.Dropout(0.001))
DNN.add(layers.Dense(25, activation='relu',
kernel_regularizer=regularizers.l1(0.001)))
DNN.add(layers.Dropout(0.001))
# 输出层选择 softmax 函数为激活函数
DNN.add(layers.Dense(10, activation='softmax'))

# 绘制网络结构示意图
plot_model(DNN, to_file='DNN.png', show_shapes=True,
show_layer_names=False, rankdir='TB')
plt.figure(figsize=(10, 10))
img = plt.imread('DNN.png')
plt.imshow(img)
plt.axis('off')
plt.show()

# 设置训练参数并进行训练
# 设置学习率为 0.001
DNN.compile(optimizer=RMSprop(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])
# 进行 20 次训练，每批 64 个数据
DNN.fit(x_train, y_train, epochs=20, batch_size=64)

# 进行模型评估
train_loss, train_acc = DNN.evaluate(x_train, y_train, batch_size=64)
print(f"训练集损失函数{train_loss:.4f},准确率{train_acc * 100:.2f}%")
test_loss, test_acc = DNN.evaluate(x_test, y_test, batch_size=64)
print(f"测试集损失函数：{test_loss:.4f},准确率：{test_acc * 100:.2f}%")
y_pred = DNN.predict(x_test).argmax(axis=-1)
y_true = y_test.argmax(axis=-1)
print("-----分类报告如下-----")
print(classification_report(y_true, y_pred))
print("混淆矩阵如下")
cm = confusion_matrix(y_true, y_pred)
print(cm)
print("-----网络报告如下-----")
print(DNN.summary())

# 绘制混淆矩阵的热力图
plt.figure(figsize=(10, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=np.arange(10), yticklabels=np.arange(10))
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# 随机展示一个样例
```

```
index = np.random.randint(0, len(x_test))
sample = x_test[index]
sample = sample.reshape(8, 8)
plt.imshow(sample, cmap=plt.cm.gray_r, interpolation='nearest')
plt.title(f"True Label:{y_true[index]},Predicted
Label:{y_pred[index]}")
plt.show()
print(np.argmax(y_pred[index]))

# 测试完毕，保存网络模型
DNN.save('DNN.h5')
```

6.3 附录三：卷积神经网络实现代码

附录三：卷积神经网络实现代码

语言：Python

运行环境：

Python 3.12.0

jupyterlab 4.1.6

scikit-learn 1.4.1post1

numpy 1.26.3

matplotlib 3.8.2

TensorFlow 2.16.1(CPU 版)

Keras 3.3.3

Pydot 2.0.0

Graphviz 0.20.3

h5py 3.11.0

运行方式：使用 jupyter 分段运行

备注：此处代码由 jupyter 整理而来，并非本文真正使用的代码。真正运行的代码见附件 CNN.ipynb

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from keras import models, layers
from keras.optimizers import Adam
from keras.utils import to_categorical
from sklearn.datasets import load_digits
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import plot_model

x, y = load_digits(return_X_y=True)  # 载入数据
x = x.reshape(-1, 8, 8, 1)
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=202406)  # 划分训练集与测试集
print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)

# 对标签进行独热编码
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# 定义网络结构
```

```
CNN = models.Sequential()

# 定义卷积层
# 该有 32 个 3x3 的卷积核，步长为 1
CNN.add(layers.Convolution2D(input_shape=(8, 8, 1), filters=32,
kernel_size=3, strides=1, padding='same',
                             activation='relu'))
# 定义池化层
CNN.add(layers.MaxPool2D(pool_size=2, strides=2, padding='same'))
CNN.add(layers.Convolution2D(filters=64, kernel_size=3, strides=1,
padding='same', activation='relu'))
CNN.add(layers.MaxPool2D(pool_size=2, strides=2, padding='same'))
# 扁平化
CNN.add(layers.Flatten())
# 全连接层
CNN.add(layers.Dense(units=512, activation='relu'))
CNN.add(layers.Dropout(0.5))
CNN.add(layers.Dense(units=10, activation='softmax'))

# 绘制网络结构示意图
plot_model(CNN, to_file='CNN.png', show_shapes=True,
show_layer_names=False, rankdir='TB')
plt.figure(figsize=(10, 10))
img = plt.imread('CNN.png')
plt.imshow(img)
plt.axis('off')
plt.show()

# 设置训练参数并进行训练
# 设置学习率为 0.001
CNN.compile(optimizer=Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])
# 进行 20 次训练，每批 64 个数据
CNN.fit(x_train, y_train, epochs=20, batch_size=64)

# 进行模型评估
train_loss, train_acc = CNN.evaluate(x_train, y_train, batch_size=64)
print(f"训练集损失函数{train_loss:.4f},准确率{train_acc * 100:.2f}%")
test_loss, test_acc = CNN.evaluate(x_test, y_test, batch_size=64)
print(f"测试集损失函数：{test_loss:.4f},准确率：{test_acc * 100:.2f}%")
y_pred = CNN.predict(x_test).argmax(axis=-1)
y_true = y_test.argmax(axis=-1)
print("-----分类报告如下-----")
print(classification_report(y_true, y_pred))
print("混淆矩阵如下")
cm = confusion_matrix(y_true, y_pred)
print(cm)
print("-----网络报告如下-----")
print(CNN.summary())

# 绘制混淆矩阵的热力图
plt.figure(figsize=(10, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=np.arange(10), yticklabels=np.arange(10))
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
# 随机展示一个样例
index = np.random.randint(0, len(x_test))
sample = x_test[index]
sample = sample.reshape(8, 8)
plt.imshow(sample, cmap=plt.cm.gray_r, interpolation='nearest')
plt.title(f"True Label:{y_true[index]},Predicted
Label:{y_pred[index]}")
plt.show()
print(np.argmax(y_pred[index]))

# 测试完毕，保存网络模型
CNN.save('CNN.h5')
```

附件文件列表：

文件名称	文件说明
SVC.ipynb	实际进行运行测试的 jupyter 文件
SVC.py	由 jupyter 文件整理的全部代码（并非本文真实运行的代码）
CNN.ipynb	实际进行运行测试的 jupyter 文件
CNN.py	由 jupyter 文件整理的全部代码（并非本文真实运行的代码）
CNN.h5	保存的全连接神经网络模型
CNN.png	全连接神经网络的结构示意图
DNN.ipynb	实际进行运行测试的 jupyter 文件
DNN.py	由 jupyter 文件整理的全部代码（并非本文真实运行的代码）
DNN.h5	保存的卷积神经网络模型
DNN.png	卷积神经网络的结构示意图