题目

# 摘要

# 目录

[摘要 1](#_Toc168590820)

[目录 2](#_Toc168590821)

[一、 算法理论介绍 1](#_Toc168590822)

[1.1 支持向量机分类器（SVC） 1](#_Toc168590823)

[1.1.1 数学表达 1](#_Toc168590824)

[1.1.2 核方法 1](#_Toc168590825)

[1.2 全连接神经网络（DNN） 1](#_Toc168590826)

[1.2.1 结构特点 2](#_Toc168590827)

[1.2.2 工作原理 2](#_Toc168590828)

[1.2.3 训练过程 2](#_Toc168590829)

[1.3 卷积神经网络（CNN） 2](#_Toc168590830)

[1.3.1 层次结构 2](#_Toc168590831)

[1.3.2 工作原理 3](#_Toc168590832)

[1.4 减少过拟合手段 3](#_Toc168590833)

[1.4.1 丢弃层（Dropout Layer） 3](#_Toc168590834)

[1.4.2 正则化 3](#_Toc168590835)

[1.5 优化方法 4](#_Toc168590836)

[1.5.1 梯度下降GD 4](#_Toc168590837)

[1.5.2 均方根传播RMSprop 4](#_Toc168590838)

[1.5.3 适应性矩估计Adam 4](#_Toc168590839)

[二、 数据集介绍 6](#_Toc168590840)

[2.1 以向量形式存储的特征集合 6](#_Toc168590841)

[2.2 以数字形式存储的标签集合 6](#_Toc168590842)

[2.3 DESCR字符串 6](#_Toc168590843)

[2.4 数据集来源 6](#_Toc168590844)

[三、 运行环境介绍 7](#_Toc168590845)

[3.1 基本信息 7](#_Toc168590846)

[3.2 软件平台 7](#_Toc168590847)

[3.2.1 Python 7](#_Toc168590848)

[3.2.2 Jupyter 7](#_Toc168590849)

[3.2.3 scikit-learn 7](#_Toc168590850)

[3.2.4 Keras 8](#_Toc168590851)

[四、 代码实现 9](#_Toc168590852)

[4.1 支持向量机的实现 9](#_Toc168590853)

[4.1.1 数据导入和预处理 9](#_Toc168590854)

[4.1.2 模型建立与训练 10](#_Toc168590855)

[4.2 全连接神经网络的实现 10](#_Toc168590856)

[4.2.1 数据预处理 10](#_Toc168590857)

[4.2.2 网络结构的定义 10](#_Toc168590858)

[4.2.3 网络训练 12](#_Toc168590859)

[4.3 卷积神经网络的实现 13](#_Toc168590860)

[4.3.1 网络结构的定义 13](#_Toc168590861)

[4.3.2 网络训练 15](#_Toc168590862)

[五、 运行结果分析 16](#_Toc168590863)

[六、 附录 19](#_Toc168590864)

[6.1 附录一：支持向量机分类器实现代码 19](#_Toc168590865)

[6.2 附录二：全连接神经网络实现代码 20](#_Toc168590866)

[6.3 附录三：卷积神经网络实现代码 21](#_Toc168590867)

# 算法理论介绍

## 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。

### 数学表达

假设我们有一个训练集,其中是输入特征，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：,其中，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，如果它对应正例则应该有，反之负例就有。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。

而支持向量机需要找的目标函数为。

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。

### 核方法

常见的核函数如下，

线性核（Linear Kernel）：它适用于线性可分的数据集。

多项式核（Polynomial Kernel）：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）：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。

## 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。

### 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。

### 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（MSE）和交叉熵（Cross-Entropy）。

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。

全连接神经网络的一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。

### 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。

## 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。

### 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。

### 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。

## 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。

### 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。

### 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：

其中，是模型参数，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：

其中，是模型参数，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。

## 优化方法

### 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，梯度下降算法更新参数的方式是：

其中，是损失函数，表示损失函数关于参数的梯度，是学习率，控制每次迭代中参数更新的步长。

### 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：

其中，是损失函数，表示损失函数关于参数的梯度，是初始学习率（注意，不是真实的学习率），衰减系数一般取0.9。是中间参数，是为了防止分母为0的小数。

### 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：

其中，是损失函数，表示损失函数关于参数的梯度，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和。和m都是中间参数，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。

# 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，这张图片是原来的向量经过处理上色得来的。

正方形

描述已自动生成

## 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。

# 运行环境介绍

## 基本信息

操作系统：Windows 11 专业版 23H2

处理器：Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz、

机带RAM：32.0 GB

显卡：NVIDIA GeForce GTX 1050 Ti

## 软件平台

### Python

Python是一种以可读性强和简洁的语法著称的多编程范式高级编程语言。它被广泛运用于科学计算、人工智能、数据分析等多个领域。与此同时，它还用有着非常庞大的标准库和第三方库生态系统。

### Jupyter

Jupyter Notebook是一个开源的Web应用程序，用于创建和共享包含实时代码、可视化和解释性文本的文档。

Jupyter Notebook的主要特点包括：

交互式编程环境：用户可以编写代码并立即执行，查看代码的输出。

内嵌数据可视化：可以轻松地创建和展示图表。

强分享性：以.ipynb扩展名保存的notebook文件可以通过电子邮件、GitHub等多种方式与其他用户共享。

高扩展性：支持超过40种编程语言，包括数据科学常用的Python、R和Julia等。

Jupyter Notebook特别适合本文这种机器学习项目，因为它允许用户在同一个环境中进行数据清洗、统计建模、构建和训练机器学习模型以及可视化结果。

### scikit-learn

scikit-learn（通常被简称为sk-learn），是一个开源的机器学习库，它为Python语言提供了一系列用于数据分析和数据挖掘的简单而有效的工具。sk-learn建立于NumPy、SciPy和matplotlib之上，是Python中机器学习最流行的库之一。

它具有以下的特点：

广泛的算法集：sk-learn提供了广泛的机器学习算法，包括分类、回归、聚类、降维、模型选择和预处理工具。

统一的API接口：sk-learn为所有模型提供了一个统一的API接口，这意味着一旦你熟悉了一种模型，就很容易应用到其他模型上。

广泛的内置数据集：sk-learn提供了一些内置的数据集，如著名的Iris数据集和MNIST数据集。

详细的文档和示例：sk-learn官方提供了非常全面说明文档，提供了大量的示例代码。

兼容性：sk-learn与其他科学计算库（如NumPy和pandas）兼容性良好，可以轻松地集成到机器学习的工作流程中。

### Keras

Keras是一个运行于TensorFlow等架构的高级神经网络Python API，旨在让深度学习模型的原型设计变得尽可能简单。

Keras具有以下主要特点：

用户友好：Keras提供了一种简单、一致的API，使得构建和试验神经网络变得非常直观并且允许用户轻松地自定义新的层、模型或损失函数，这使得研究人员能够根据需要以非常简单的方式扩展Keras的功能。

模块化：Keras模型是通过堆叠不同的层来构建的，每一层都是一个小模块，这样的设计使得模型非常模块化，易于理解和维护。

广泛的预训练模型：Keras提供了大量的预训练模型，如VGG16、InceptionResNetV3等，这些模型可以用于图像识别、文本处理等任务，并且可以很容易地被重新训练或微调以适应新的任务。

与TensorFlow集成：由于Keras现在是TensorFlow的一部分，它能够充分利用TensorFlow的强大功能，包括分布式训练和强大的GPU加速。（但是在本文，为了简洁起见使用了CPU版的TensorFlow因而不能使用GPU加速）

# 代码实现

## 支持向量机的实现

### 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，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。

### 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) |

## 全连接神经网络的实现

### 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，keras库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) |

### 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

图示, 日程表

描述已自动生成

### 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。

## 卷积神经网络的实现

### 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。

图示, 日程表

描述已自动生成

### 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) |

# 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，而，F1值则是acc和recall的调和平均。

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() |

图表, 散点图

描述已自动生成最终可以得到支持向量机的分析数据如左下：

|  |  |  |  |
| --- | --- | --- | --- |
| 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% | | |

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()) |

图表, 散点图

描述已自动生成据此可以得到全连接神经网络的分析结果为:

|  |  |  |  |
| --- | --- | --- | --- |
| 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% | | |

卷积神经网络为：

|  |  |  |  |
| --- | --- | --- | --- |
| 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% | | |

从这些数据来看，三种分类方法的准确率都在图表, 散点图

描述已自动生成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。

# 附录

## 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  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 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") print(confusion\_matrix(y\_test, y\_pred))  *# 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() |

## 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  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 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  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'))  *# 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") print(confusion\_matrix(y\_true, y\_pred))  *# 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])) |

## 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  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 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  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'))  *# 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") print(confusion\_matrix(y\_true, y\_pred))  *# 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])) |