# 高级人工智能大作业

**姓　　名：** 赵路路

**学　　号：** 2023202210120

**小组成员：** 王亚龙、沈静远、甘泽昊、赵家璇

**编写日期：** 2023 年 12 月 26 日

# 目 录

# 1 手写数字识别实验

## 1.1 实验要求

使用以下算法实现手写数字识别任务：

（1）KNN 算法，可以使用 sklearn 库，也可以自己实现。

（2）全连接的神经网络模型，可以使用 sklearn 库。

（3）全连接的神经网络模型，仅可使用 numpy 库，手动实现。

（4）CNN 卷积神经网络模型，可以使用 sklearn 库。

最终使用测试集评估算法的性能，比较各算法的差异，并形成实验报告：

（1）对于 KNN 算法，需设置不同的 K 值，比较结果差异；

（2）对于神经网络模型，需设置不同学习率、隐层节点数，比较结果差异；

（3）对比上述所有模型的性能差异。

## 1.2 数据集介绍

DBRHD （Pen-Based Recognition of Handwritten Digits）是 UCI 机器学习中心提供的手写数字数据集，他们使用一台 WACOM PL-100V 压感平板，连接到基于 Intel 486 的 PC 的串口收集手写样本，最终收集到来源于 44 位不同书写者的手写数字数据，每个数据条项包括 16 维特征及标签。

（1）训练集：来源于 30 位书写者的 7494 张手写数字特征及其对应标签。

（2）测试集：来源于 14 位书写者的 3498 张手写数字特征及其对应标签。

数据分布如图 1 所示：

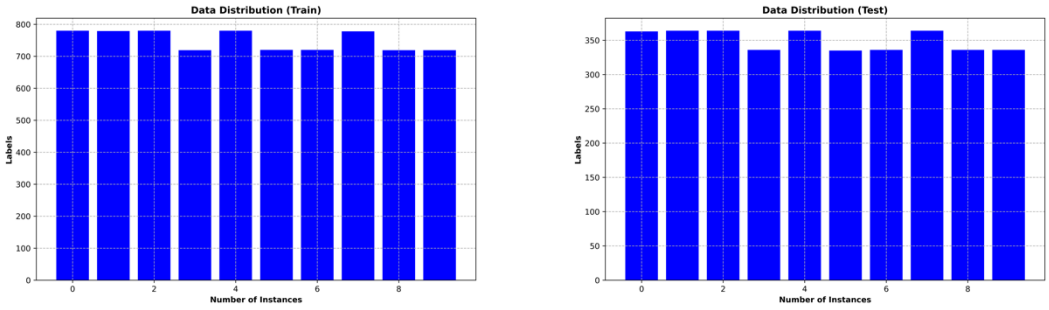


图 1 数据集分布柱状图

## 1.3 算法一：KNN

### 1.3.1 算法介绍

K 最近邻算法（K-Nearest Neighbors，KNN）是一种常用的监督学习算法，用于分类和回归问题。它基于样本之间的距离度量进行预测，即根据最邻近的 K 个训练样本的标签来确定新样本的标签。如图 2 所示，KNN 算法的基本思想是，如果一个样本在特征空间中的 K 个最近邻居中的大多数属于某个类别，则该样本也属于这个类别。

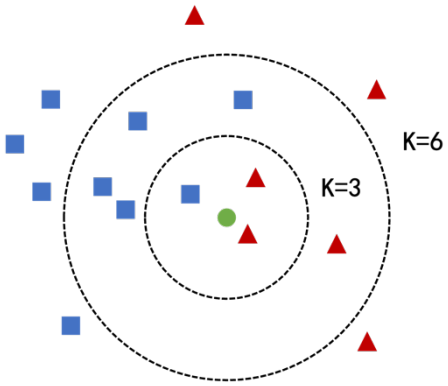


**图 2 KNN 算法示意图**

KNN 算法的主要步骤如下：

（1）准备数据集：收集训练样本数据集，包括输入特征和对应的标签。

（2）选择 K 值：确定 K 的取值，即要考虑的最近邻居的数量。K 值的选择会影响算法的性能，通常需要通过交叉验证等方法进行选择。

（3）计算距离：对于一个新的测试样本，计算它与训练样本集中每个样本之间的距离。常用的距离度量方法包括欧氏距离、曼哈顿距离等。

（4）选择最近邻居：根据距离大小选择与测试样本最近的 K 个训练样本。

（5）进行预测：对于分类问题，根据 K 个最近邻居的标签进行投票，将得票最多的类别作为测试样本的预测类别。对于回归问题，可以取 K 个最近邻居的平均值作为测试样本的预测值。

KNN 算法简单易用、无需训练过程、对异常值不敏感。然而，由于需要计算测试样本与所有训练样本之间的距离，KNN 算法的计算复杂度较高。此外，特征空间的维度较高时，需进行降维或特征选择等处理。

### 1.3.2 代码实现

（1）sklearn 库实现

如图 3 所示，初始化 KNN 分类器 KNeighborsClassifier，输入训练集数据，对测试集中数据进行类别预测，计算测试集上的准确率。

```
1 个用法  zllwhu
@get_time
def train_and_predict():
    for i in k:
        print("当前 k 值: " + str(i))
        knn = KNeighborsClassifier(n_neighbors=i)
        knn.fit(X_train, y_train)
        y_pred = knn.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        result_accuracy.append(accuracy)
        print("当前精度: " + str(accuracy))
```

图 3 KNN 算法 sklearn 实现代码图

设置 K 值为 1～7 重复实验，得到分类准确率结果如图 4 所示。

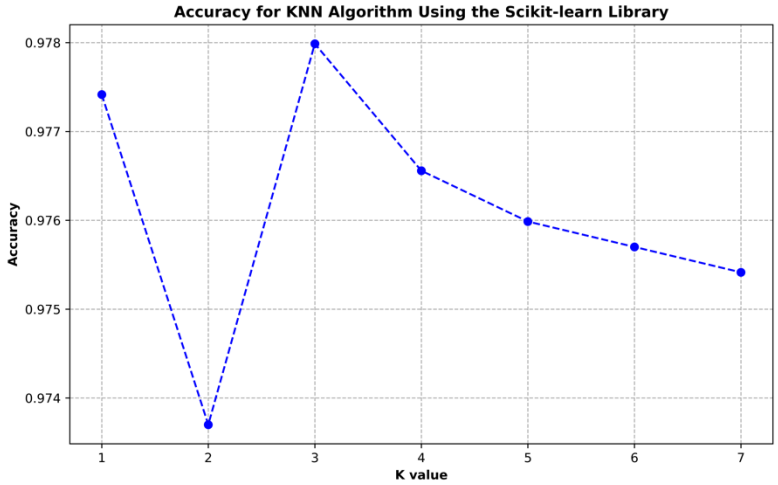


图 4 KNN 算法 sklearn 实现结果图

（2）手动实现

如图 5 所示，构建一个 KNN 分类器，定义预测函数。预测函数需首先计算输入的待预测样本与训练集所有样本之间的距离（定义为欧式距离），然后按照距离从小到大进行排序，选取前 K 个最近的训练样本点，

根据这 K 个点的标签投票表决待预测样本的标签。

```
zllwhu
class KNN:
    zllwhu
    def __init__(self, k, X_train, y_train):
        self.k = k
        self.X_train = X_train
        self.y_train = y_train

    zllwhu
    def predict(self, X_test):
        distances = []
        for i in range(len(self.X_train)):
            distance = euclidean_distance(self.X_train[i], X_test)
            distances.append((distance, self.y_train[i]))
        distances.sort(key=lambda x: x[0])
        k_nearest = distances[:self.k]
        k_nearest_labels = [label for (_, label) in k_nearest]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]
```

图 5 KNN 算法手动实现代码图

```
            return most_common[0][0]
```

设置 K 值为 1～7 重复实验，得到分类准确率结果如图 6 所示。

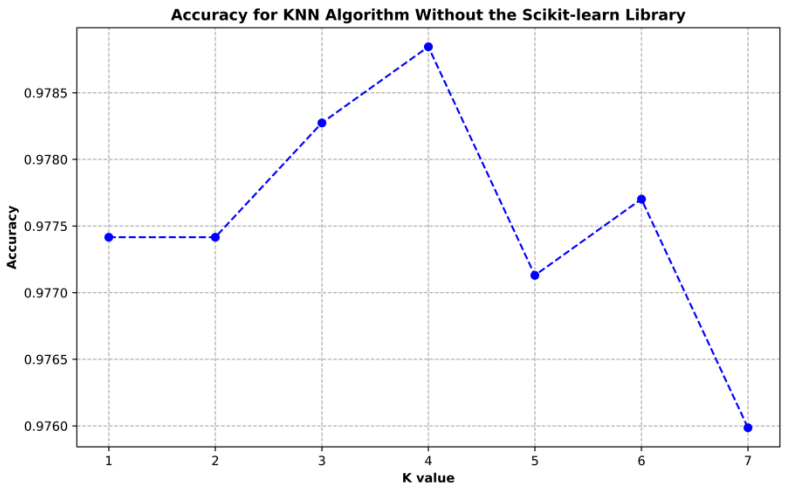


图 6 KNN 算法手动实现结果图

### 1.3.3 实验结果

统计 sklearn 和手动实现的 KNN 算法测试结果，如表 1 和表 2 所示。

**表 1 KNN 算法测试结果表（准确率）**

| 实现方式 | K 值 | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| sklearn | 0.9774 | 0.9737 | 0.9780 | 0.9766 | 0.9760 | 0.9757 | 0.9754 |
| 手动 | 0.9774 | 0.9774 | 0.9783 | 0.9788 | 0.9771 | 0.9777 | 0.9760 |

**表 2 KNN 算法测试结果表（错误数）**

| 实现方式 | K 值 | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| sklearn | 79 | 91 | 76 | 81 | 83 | 85 | 86 |
| 手动 | 79 | 79 | 75 | 74 | 80 | 78 | 83 |

重复实验多次，统计并计算 sklearn 和手动实现的 KNN 算法对测试集样本进行预测的平均耗时，如表 3 所示。

**表 3 KNN 算法测试平均耗时表**

| 实现方式 | 平均耗时/秒 |
|---|---|
| sklearn | 2.8612 |
| 手动 | 1376.2428 |

通过统计并分析实验结果，可以发现 sklearn 实现和手动实现的 KNN 算法，在准确率指标上的表现差距极小，但手动实现的计算耗时远高于 sklearn 实现，原因分析如下：

（1）sklearn 库实现 KNN 算法时会对训练数据随机打乱（shuffle），在前 K 个最近邻样本点投票表决时，出现相同票数会随机选择最终输出标签；在手动实现中，对于相同票数情况，选择在训练集中顺序靠前的标签输出，故二者在准确率指标上略有差异。

（2）在计算性能上，由于 sklearn 库经过高度优化，采用 Cython 等底层实现方法提高运行效率。此外，sklearn 库在计算时采用并行化技术加速运算。

## 1.4 算法二：全连接神经网络

### 1.4.1 算法介绍

全连接神经网络（Fully Connected Neural Network，FCNN），也被称为多层感知机（Multilayer Perceptron，MLP），是一种经典的人工神经网络模型。它由多个神经元层组成，其中每个神经元与前一层的所有神经元连接，并将其输入加权求和后通过激活函数进行非线性转换。

如图 7 所示，为全连接神经网络的一般结构示意。

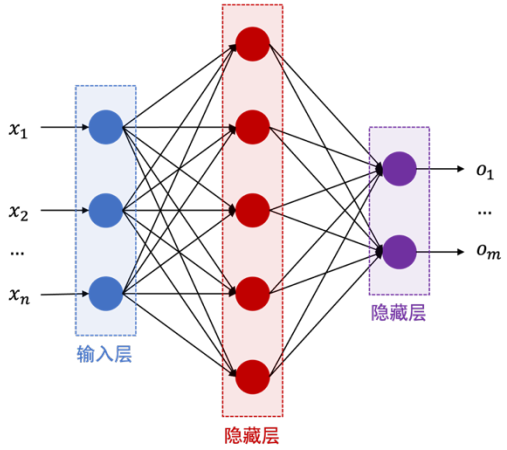


**图 7 全连接神经网络一般结构示意图**

（1）输入层（Input Layer）：接受输入数据的层。每个输入特征都对应一个输入神经元。

（2）隐藏层（Hidden Layers）：中间层，位于输入层和输出层之间。每个隐藏层包含多个神经元，这些神经元与前一层的所有神经元相连接。

（3）输出层（Output Layer）：产生模型的输出结果。输出层的神经元数量取决于任务的要求，例如二分类问题通常使用一个神经元，多分类问题可能使用多个神经元。

全连接神经网络的训练过程主要包括前向传播和反向传播两个阶段：

（1）前向传播（Forward Propagation）：在前向传播中，输入数据通过网络的各个层，每个神经元将其输入加权求和并应用激活函数。这样，网络逐层计算输出，直到达到输出层，产生模型的预测结果。

（2）反向传播（Backpropagation）：在反向传播中，通过比较模型的预测

结果和真实标签，计算损失函数的梯度。然后，梯度从输出层向输入层传播，根据链式法则更新每个神经元的权重，以最小化损失函数。

全连接神经网络在许多领域广泛应用，包括图像分类、语音识别、自然语言处理等。然而，全连接神经网络的参数量随着网络层数和神经元数量的增加而增加，容易导致过拟合问题。为了应对这个问题，可以使用正则化技术、Dropout、批归一化等方法来提高模型的泛化能力。

### 1.4.2 代码实现

（1）sklearn 实现

如图 8 所示，通过两层循环的设置，使用 sklearn 库提供的多层感知机模型 MLPClassifier 构建不同学习率和隐藏层神经元个数的全连接神经网络。此外需要注意，本次试验采用随机梯度下降优化器、固定学习率的模式，而非 Adam 优化器、自适应调整学习率的模式。激活函数选择使用 sigmoid 型函数，最大迭代轮次为 2000 轮。使用训练数据对模型进行训练，将训练得到的模型用于测试数据的测试，计算模型的分类准确率记录。

```
1 个用法  zllwhu *
@get_time
def train_and_predict():
    for j in range(len(learning_rate)):
        print("当前学习率: " + str(learning_rate[j]))
        for i in range(len(hidden_layer_neural_unit)):
            print("当前隐层神经元数量: " + str(hidden_layer_neural_unit[i]))
            model = MLPClassifier(hidden_layer_sizes=hidden_layer_neural_unit[i],
                                  activation='logistic',
                                  solver='sgd',
                                  random_state=None,
                                  learning_rate='constant',
                                  learning_rate_init=learning_rate[j],
                                  max_iter=2000,
                                  early_stopping=False,
                                  shuffle=False)
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            accuracy = accuracy_score(y_test, y_pred)
            result_accuracy[j].append(accuracy)
            print("当前精度: " + str(accuracy))
```

图 8 全连接神经网络 sklearn 实现代码图

设置学习率为 0.1、0.01、0.001、0.0001，隐藏层神经元个数为 500、1000、1500、2000，重复实验，得到分类准确率结果如图 9 所示。

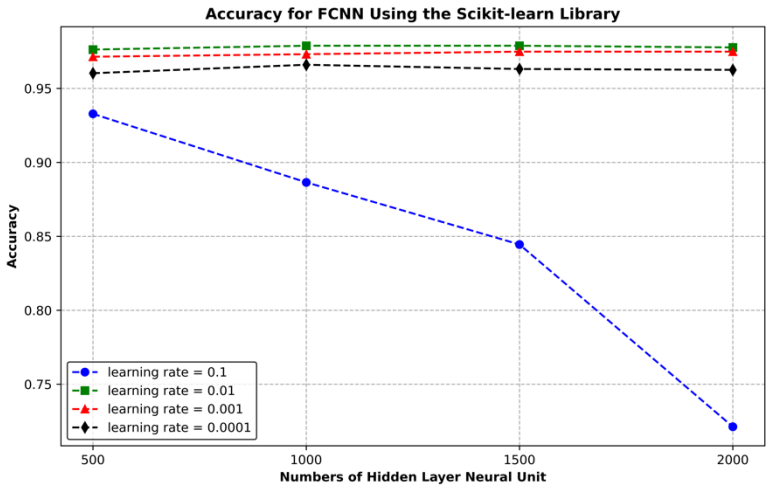


图 9 全连接神经网络 sklearn 实现结果图

（2）手动实现

如图 10 所示，定义全连接神经网络类 NeuralNetwork，定义成员变量输入尺寸、隐藏层尺寸、输出尺寸、学习率、最大迭代轮数、分组大小。

```
zllwhu *
class NeuralNetwork:
    zllwhu *
    def __init__(self, input_size, hidden_size, output_size,
                 learning_rate, max_iter, batch_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.batch_size = batch_size
        self.weights1 = None
        self.biases1 = None
        self.weights2 = None
        self.biases2 = None
```

图 10 全连接神经网络手动实现类定义代码图

如图 11 所示，定义前向传播过程，依次计算输入数据的加权和与非线性变换得到的激活值，计算第二层的加权和与非线性变换得到的激活值，最终将两层激活值返回。

zllwhu
```
def forward_propagation(self, X):
    Z1 = np.dot(X, self.weights1) + self.biases1
    A1 = self.sigmoid(Z1)
    Z2 = np.dot(A1, self.weights2) + self.biases2
    A2 = self.sigmoid(Z2)
    return A1, A2
```

图 11 全连接神经网络手动实现前向传播代码图

如图 12 所示，定义反向传播过程，依次计算输出层权重和偏差的梯度，隐藏层权重和偏差的梯度，返回计算得到的梯度用于参数更新。

zllwhu
```
def backward_propagation(self, X, y, A1, A2):
    m = X.shape[0]
    dZ2 = A2 - y
    dW2 = np.dot(A1.T, dZ2) / m
    dB2 = np.mean(dZ2, axis=0, keepdims=True)
    dZ1 = np.dot(dZ2, self.weights2.T) * A1 * (1 - A1)
    dW1 = np.dot(X.T, dZ1) / m
    dB1 = np.mean(dZ1, axis=0, keepdims=True)
    return dW1, dB1, dW2, dB2
```

图 12 全连接神经网络手动实现反向传播代码图

如图 13 所示，定义参数更新过程，更新操作将根据梯度和学习率的乘积，以梯度下降的方式沿着损失函数的梯度方向调整神经网络的参数，以尽量减小损失。

zllwhu
```
def update_parameters(self, dW1, dB1, dW2, dB2):
    self.weights1 -= self.learning_rate * dW1
    self.biases1 -= self.learning_rate * dB1
    self.weights2 -= self.learning_rate * dW2
    self.biases2 -= self.learning_rate * dB2
```

图 13 全连接神经网络手动实现参数更新代码图

如图 14 所示，定义模型训练过程，包括参数初始化、迭代更新、前向传播、反向传播和参数更新过程，通过不断迭代重复训练模型。

```
zllwhu
def train(self, X, y):
    self.initialize_parameters()
    num_samples = X.shape[0]

    for _ in range(self.max_iter):
        for batch_start in range(0, num_samples, self.batch_size):
            batch_end = batch_start + self.batch_size
            X_batch = X[batch_start:batch_end]
            y_batch = y[batch_start:batch_end]

            A1, A2 = self.forward_propagation(X_batch)
            dW1, dB1, dW2, dB2 = self.backward_propagation(X_batch, y_batch, A1, A2)
            self.update_parameters(dW1, dB1, dW2, dB2)
```

图 14 全连接神经网络手动实现模型训练代码图

如图 15 所示，定义模型实例化、训练、预测和结果记录过程。

```
1 个用法  zllwhu
@get_time
def train_and_predict():
    for j in range(len(learning_rate)):
        print("当前学习率: " + str(learning_rate[j]))
        for i in range(len(hidden_layer_neural_unit)):
            print("当前隐层神经元数量: " + str(hidden_layer_neural_unit[i]))
            output_size = y_train_encoded.shape[1]
            model = NeuralNetwork(X_train.shape[1], hidden_layer_neural_unit[i], output_size,
                                  learning_rate=learning_rate[j], max_iter=2000, batch_size=200)
            model.train(X_train, y_train_encoded)
            y_pred = model.predict(X_test)
            accuracy = np.mean(y_pred == y_test)
            result_accuracy[j].append(accuracy)
            print("当前精度: " + str(accuracy))
```

图 15 全连接神经网络手动实现模型实例化代码图

设置学习率为 0.1、0.01、0.001、0.0001，隐藏层神经元个数为 500、1000、1500、2000，重复实验，得到分类准确率结果如图 16 所示。

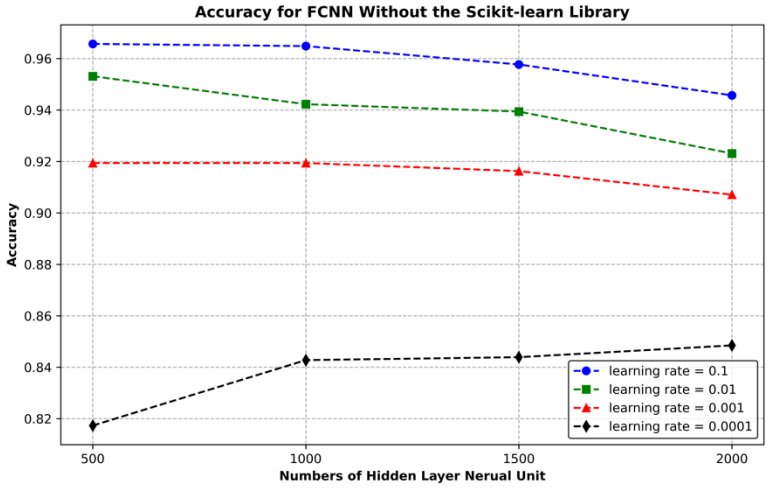


图 16 全连接神经网络手动实现结果图

### 1.4.3 实验结果

统计 sklearn 和手动实现的全连接神经网络准确率测试结果，如表 4 和表 5 所示。

表 4 全连接神经网络测试结果表（sklearn 实现）

| 学习率 | 隐藏层神经元数量 | | | |
|---|---|---|---|---|
| | 500 | 1000 | 1500 | 2000 |
| 0.1 | 0.9328 | 0.8865 | 0.8445 | 0.7213 |
| 0.01 | 0.9763 | 0.9788 | 0.9788 | 0.9777 |
| 0.001 | 0.9714 | 0.9731 | 0.9748 | 0.9748 |
| 0.0001 | 0.9603 | 0.9660 | 0.9631 | 0.9626 |

表 5 全连接神经网络测试结果表（手动实现）

| 学习率 | 隐藏层神经元数量 | | | |
|---|---|---|---|---|
| | 500 | 1000 | 1500 | 2000 |
| 0.1 | 0.9657 | 0.9648 | 0.9577 | 0.9457 |
| 0.01 | 0.9531 | 0.9423 | 0.9394 | 0.9231 |
| 0.001 | 0.9194 | 0.9194 | 0.9162 | 0.9071 |
| 0.0001 | 0.8173 | 0.8428 | 0.8439 | 0.8485 |

重复实验多次，统计并计算 sklearn 和手动实现的全连接神经网络对测试集样本进行预测的平均耗时，如表 6 所示。

**表 6 全连接神经网络测试平均耗时表**

| 实现方式 | 平均耗时/秒 |
|---|---|
| sklearn | 791.3919 |
| 手动 | 3960.6146 |

通过统计并分析实验结果，可以发现 sklearn 实现和手动实现的全连接神经网络，学习率较大时，在准确率指标上的表现差距较小；学习率较小时，在准确率指标上的表现差距较大。此外，手动实现的计算耗时远高于 sklearn 实现。原因分析如下：

（1）sklearn 库使用小批量随机梯度下降等优化方式加快了模型收敛速度。

（2）在计算性能上，sklearn 库经过高度优化提高运行效率，且使用并行化技术加速运算。

## 1.5 算法三：卷积神经网络

### 1.5.1 算法介绍

卷积神经网络（Convolutional Neural Networks，CNN）是一类主要用于处理具有网格结构数据，如图像和视频的深度学习模型。CNN 在计算机视觉领域取得了很大成功，其设计灵感来自生物学中对视觉系统的理解，尤其是视觉皮层中的神经元的工作方式。

如图 4 所示，为卷积神经网络的一般结构示意。

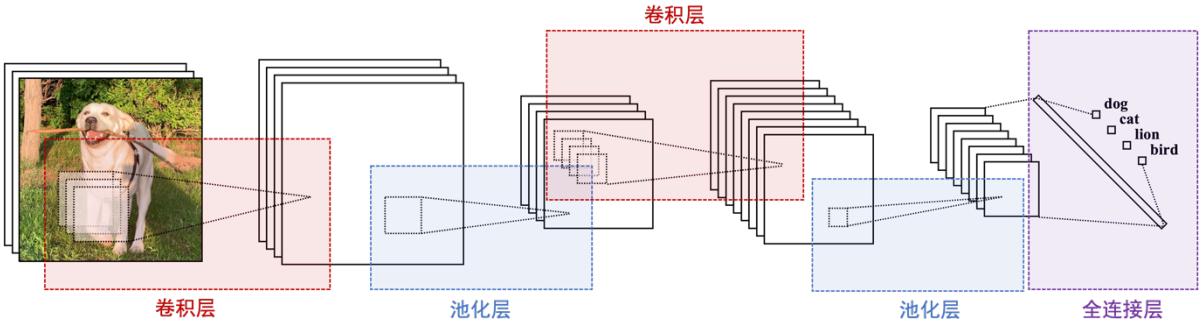


**图 17 卷积神经网络一般结构示意图**

（1）输入层（Input Layer）：用于接收输入数据，通常表示为图像。每个输入节点对应图像中的一个像素值或通道。

（2）卷积层（Convolutional Layer）：卷积层是 CNN 的核心组件。它通过在输入图像上滑动卷积核，学习图像中的特征。卷积操作可以有效地捕捉图像的局部模式，如边缘、纹理等。

（3）池化层（Pooling Layer）：池化层用于降低特征图的空间维度，减少计算复杂度，并提取图像的重要特征。常见的池化操作包括最大池化和平均池化。

（4）全连接层（Fully Connected Layer）：全连接层通常位于网络的顶部，将前面层的输出转换为最终的输出。全连接层的每个节点都与前一层的所有节点相连，通过权重连接。在图像分类任务中，全连接层通常用于输出类别概率。

（5）输出层（Output Layer）：输出层产生最终的预测结果。

如表 7 所示，是一些常见的卷积神经网络及其特点。

**表 7 常见卷积神经网络表**

| 网络名 | 网络特点 |
|---|---|
| LeNet-5 | LeNet-5 是早期的卷积神经网络，主要用于手写数字识别。包含卷积层、池化层和全连接层。 |
| AlexNet | AlexNet 是在 2012 年 ImageNet 图像分类比赛中获胜的模型，推动了深度学习在计算机视觉中的应用。具有 8 个卷积层和 3 个全连接层。 |
| VGGNet | VGGNet 的核心思想是使用多个 3x3 的小卷积核来代替一个大的卷积核，以增加网络的深度。VGGNet 有 16 层或者 19 层的深度。 |
| GoogLeNet | GoogLeNet 使用了 Inception 模块，该模块同时使用多个不同大小的卷积核，提供了不同尺度的信息。具有 22 层的深度。 |
| ResNet | ResNet 是使用残差块（Residual Block）的模型，通 |

| | |
|---|---|
| | 过引入残差连接，解决了深度网络中梯度消失问题。ResNet 赢得了 2015 年 ImageNet 图像分类比赛。 |
| MobileNet | MobileNet 是一种轻量级的卷积神经网络，专注于在资源受限的环境中运行，如移动设备。使用深度可分离卷积来减少参数数量。 |
| EfficientNet | EfficientNet 基于网络缩放的思想，通过在深度、宽度和分辨率上进行均衡调整，以在相对较少的参数下提高性能。 |

### 1.5.2 代码实现

如图 18 所示，使用 pytorch 库定义卷积神经网络类 CNN1D，构建网络结构为卷积层、激活层、池化层、全连接层。

其中，如图 19 所示，卷积层是尺寸为 3 的一维卷积核，步长为 1，零填充为 1；池化层是尺寸为 2 的池化核，步长为 2。

```
2 个用法  zllwhu
class CNN1D(nn.Module):
    zllwhu
    def __init__(self):
        super(CNN1D, self).__init__()

        self.conv1 = nn.Conv1d( in_channels: 1,  out_channels: 16, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool1d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(16 * 8,  out_features: 128)
        self.fc2 = nn.Linear( in_features: 128,  out_features: 10)
```

图 18 卷积神经网络类定义代码图

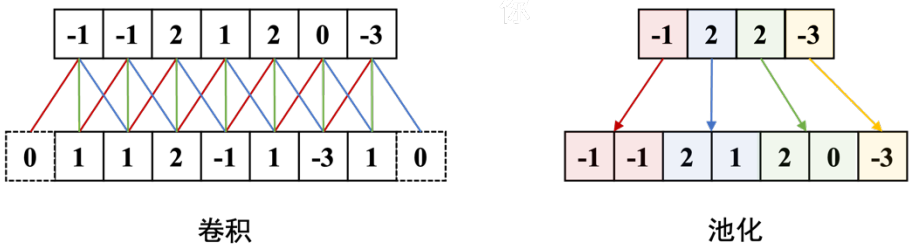


图 19 卷积和池化操作示意图

如图 20 所示，定义模型训练过程，包括选取批次、前向传播、反向传播和参数优化、计算平均损失并记录。

zllwhu
```
def forward(self, x):
    x = self.conv1(x)
    x = self.relu(x)
    x = self.maxpool(x)
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = self.relu(x)  # 添加ReLU激活函数
    x = self.fc2(x)
    return x
```

图 20 卷积神经网络模型训练代码图

如图 21 所示，定义模型评估过程，使用训练得到的模型对测试集数据进行预测，计算准确率并记录。

1 个用法 zllwhu *
```
@get_time
def train_and_predict():
    for epoch in tqdm(range(num_epochs)):
        model.train()  # 设置模型为训练模式
        total_loss = 0.0
        # 批量训练
        for i in range(0, len(X_train), batch_size):
            inputs = X_train[i:i + batch_size]
            labels = y_train[i:i + batch_size]
            # 梯度清零
            optimizer.zero_grad()
            # 前向传播
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            # 反向传播和优化
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        # 计算平均损失
        avg_loss = total_loss / (len(X_train) // batch_size)
        # 记录训练信息
        losses.append(avg_loss)
```

图 21 卷积神经网络模型评估代码图

### 1.5.3 实验结果

使用 Adam 优化器，批次大小 200，训练 30 轮后，平均交叉熵损失如图 21 所示。模型最终分类准确率为 0.9671，训练耗时 1.6274 秒。

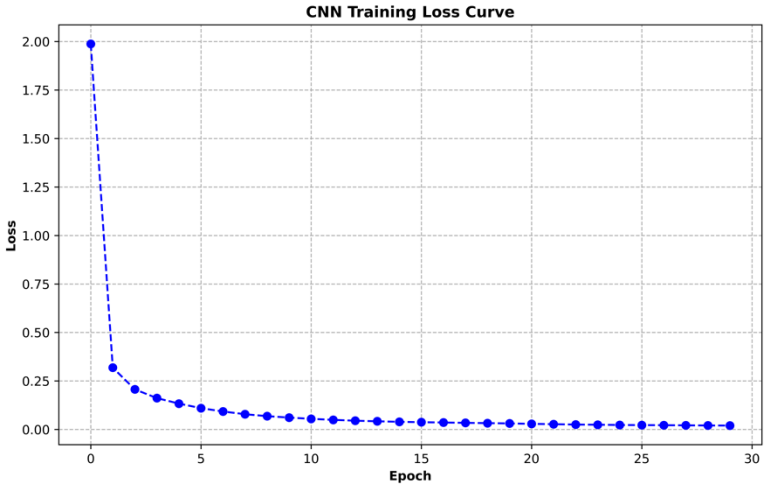


图 22 卷积神经网络 Loss 曲线图

# 2 小组论文分享报告

# 3 个人技术报告