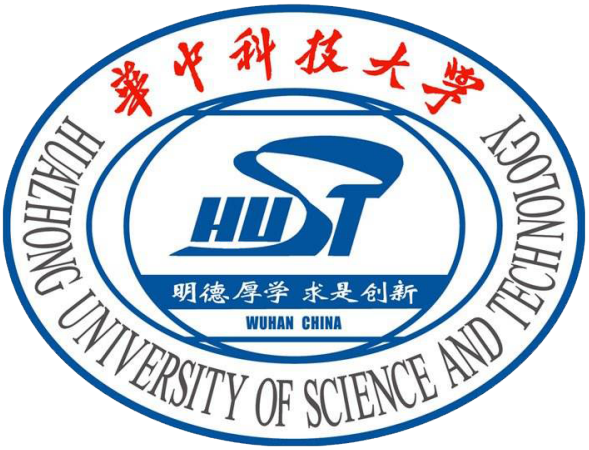
华中科技大学计算机科学与技术学院

《机器学习》结课报告



专　　业	计算机科学与技术
班　　级	CS2203
学　　号	U202115473
姓　　名	刘欣逸
成　　绩	
指导教师	何　琨
时　　间	2024 年 5 月 20 日

目录

1 实验要求

总体要求：

1. 模型训练需自己动手实现，严禁直接调用已经封装好的各类机器学习库（包括但不限于 *Sklearn*，功能性的可以使用，比如 *sklearn.model_selection.train_test_split*），但可以使用 *NumPy* 等数学运算库（实现后，可与已有库进行对比验证）；
2. 使用机器学习及相关知识对数据进行建模和训练，并进行相应参数调优和模型评估；
3. 鼓励使用多种模型或不同数据集进行实验，并给出相应的分析思考；
4. 鼓励自主拓展探索；
5. 严禁抄袭任何来源的代码或报告，一经发现大作业直接记 0 分处理；如需借鉴或引用，请标明出处；

2 引言

图 2.1: MNIST 数据集

手写数字识别问题是机器学习领域的经典问题之一，也是深度学习领域的入门问题，是一个多分类问题，主要应用于银行支票识别、邮政编码识别等，具有重要的现实意义。MNIST 数据集是一个在机器视觉领域的字符识别基准数据集，被广泛应用于训练和评估各种图像处理系统。如图2.1所示，由 LeCun 在 1998 年提出，由 60000 张图片的训练集和 10000 张测试图片的测试集组成。每张图片都是 28*28 的灰度手写数字图片，每张图片都有一个标签，标签是 0-9 的数字，表示图片上的数字。要对手写数字进行识别，实际上是将数据集的每个样本分为 0-9 十个类别，是一个经典的多分类问题。

3 算法设计与实现

3.1 k 近邻算法

k 近邻算法对于每一个测试集中的样本点，使用最为临近的 k 个测试集的点进行多数表决来决定这个样本点的标签。k 近邻算法的一大特点是没有显示的训练过程，

3.1.1 算法设计

在 k 近邻算法中，需要定义一个距离度量。我选择了 Minkowski 距作为距离度量，样本点 x_i, x_j 的 Minkowski 距离为：

$$L_p(x_i, x_j) = (\sum_{l=1}^{n} |x_i^{(l)} - x_i^{(l)}|^p)^{1/p}$$

其中，n 是特征空间的维数，$p > 1$. 当 $p = 2$ 时，就得到了欧式距离。k 邻近算法描述为：输入：有 N 个实例的训练数据集

$$T = \{(x_1, y_1), (x_2, y_2)..., (x_N, y_N)\}$$

其中 $x_i \in R^n$ 为实例的特征向量，$y \in \{c_1, c_2..., c_k\}$ 为实例的类别（标签）。在这个问题中，$x_i \in \{0, 1.., 255\}, y \in \{0, 1..., 9\}$。决定一个测试集实例点的算法如下：

Algorithm 1 k-NN

1: Input: instance x to be predicted.
2: Output: predicted y of instance x.
3: Calculate $L_p(x_i, x)$ for all $x_i \in T$.
4: Sort T on the ascending order of $L_p(x_i, x)$
5: $N_k(x) \leftarrow \{x_{k-NN} |$ where x_{k-NN} is k-nearest neighbors of x according to $L_p(x_i, x)\}$
6: $y \leftarrow \underset{c_j}{argmax} \sum_{x_i \in N_k(x)} I(y_i = c_j)$

3.1.2 算法实现

使用 *NumPy* 实现。代码关键部分如下：

```
1  def Minkowski(x, y, p = 2):
2      return np.sum(abs(x - y) ** p) ** (1 / p)
3
4  for point in test_set:
5      distances = np.array([[Minkowski(point, x[1:]), x[0]] for x in train_set])
6      distances = distances[distances[:, 0].argsort()][:K]
7      # 统计 label 0~9 在邻近点出现了几次
```

```
8       tmp = np.array([[i, np.sum(distances[..., 1] == i)] for i in range(0, 10)])
9       # 找出出现最多的 label
10      max_likelihood = tmp[:, 1].argmax()
11      res_send.append(max_likelihood)
```

3.1.3 改进效率——并行计算实现

k 近邻算法的计算量是非常大的。如果不对计算做并行化，预测全部的测试集将会耗费相当长的时间。我观察到算法运行时 CPU 多核利用率低，推测原因是受 MNIST 数据集的限制，*NumPy* 的调用的特点是计算量少（两个长度为 784 向量计算距离），调用次数大（42000×28000 次），导致 NumPy 库不会开启多线程计算（并行计算的收益小于线程启动的开销），从而算法的运行效率较低。

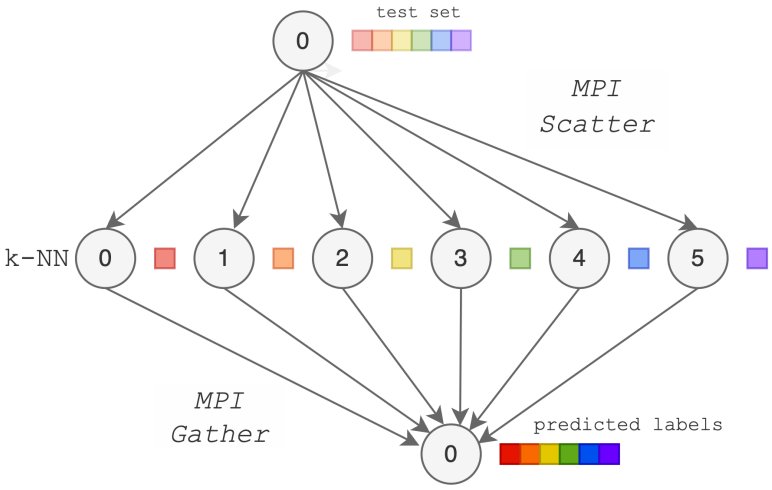


图 3.1: 基于 MPI 的并行 k-NN 算法

前面说过，k 邻近算法的一大特点是没有显式的学习过程（或者说，k 邻近是非参数模型），非常适合进行数据并行。高性能计算常用的消息传递接口（Message Passing Interface, MPI），是一种进程间通信的接口，一般用于 C/C++/Fortran 的程序，但是 Python 也实现了相应的 API。我使用 MPI4Py 的 *Scatter* 方法在主进程中把测试集划分给多个从进程并行处理，再用 *Gather* 方法将用数据收集回根进程，如图3.1所示，这种数据并行（Data Parallel）的方式加快了运行速度。同时，MPI 进程可以分布在集群上的多个节点运行，极大提高了程序的可拓展性（scalabilty）。

方案	使用核数	使用节点数	时间
串行计算	1	1	约 3 小时 50 分钟
MPI 并行计算	240	4	约 120 秒

表 3.1: 不同实现的计算效率比较

MPI 并行算法的代码过长，不在正文赘述。

3.1.4 参数调优与模型评估

参数调优 k 值的选择会对 k 邻近算法的表现产生重大影响，如果 k 过小，只有和输入实例相近的少数点会对结果起决定作用，如果这几个点恰巧是噪声，模型就会给出错误的结果。如果 k 值过大，距离输入相差较远的点也会对结果产生影响。在实践中一般使用交叉验证法取一个较小的 k 值 [2]。我按 9:1 比例的划分了训练集进行了多次测试，得到的测试数据如下：结果显示，最优的 k 值确实较小。

k	准确率（%）
1	96.38
2	**97.90**
3	97.00
4	96.74
5	96.71
7	96.71
9	96.38
15	96.09
20	95.59
40	94.93
80	93.50
160	91.52

表 3.2: k 值对模型表现的影响

模型评估 在 Kaggle 的测试集上达到了 96.7% 的准确率。

图 3.2: Kaggle 评测结果

3.2 多层感知机算法

3.2.1 算法设计

MLP，全称是多层感知机（Multi-Layer Perceptron），是一种前向结构的人工神经网络，映射一组输入向量到一组输出向量。MLP 可以被看作是一个有向图，由多个的节点层所组成，每一层都全连接到下一层。除了输入节点，每个节点都是一个带有非线性激活函数的神经元（或称为处理元）。一种被广泛使用的激活函数是 ReLU 函数。MLP 是深度学习网络的一种，可以用于分类或者回归问题。MLP 的一个主要优点是能够学习不可见的特征，这在很多机器学习任务中是非常有用的。

问题定义 有 N 个实例的训练数据集

$$T = \{(x_1, y_1), (x_2, y_2)..., (x_N, y_N)\}$$

其中 $x_i \in R^n$ 为实例的特征向量，$y \in \{c_1, c_2..., c_k\}$ 为实例的类别（标签）。

定义损失函数 采用交叉熵损失函数：

$$loss(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i log(\hat{y}_i)$$

其中标签用 one-hot 标签表示。

前向传播 前向传播的算法比较简单。记第 $l \in \{1, 2..., L\}$ 层的参数 $\mathbf{W}^{(l)}$ 和 $\mathbf{b}^{(l)}$，神经元的值为 $\mathbf{X}^{(l)}$，激活函数为 $\sigma(\circ)$, 则前向传播的公式为：

$$\mathbf{X}^{(l+1)} = \sigma(\mathbf{X}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}) \tag{3.1}$$

在输出层后接 $softmax(\circ)$ 层实现分类：

$$\hat{\mathbf{y}} = softmax(\mathbf{X}^{(L)})$$

反向传播 对交叉熵损失函数 $loss(\mathbf{y}, \hat{\mathbf{y}})$，可以证明：

$$\partial_{\mathbf{X}^{(L)}} loss = softmax(\mathbf{X}^{(L)}) - \mathbf{y}$$

对于隐层和输入层有

$$\begin{aligned}
\partial_{\mathbf{X}^{(l)}} loss &= \partial_{\mathbf{X}^{(l+1)}} loss \times \partial_{\mathbf{X}^{(l)}} \mathbf{X}^{(l+1)} \\
&= \partial_{\mathbf{X}^{(l+1)}} loss \times \partial_{\mathbf{X}^{(l)}} \sigma(\mathbf{X}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}) \\
&= \partial_{\mathbf{X}^{(l+1)}} loss \times \sigma'(\mathbf{X}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}) \times \mathbf{W}^{(l)\top}
\end{aligned} \tag{3.2}$$

$$\begin{aligned}\partial_{\mathbf{W}^{(l)}} loss &= \partial_{\mathbf{X}^{(l+1)}} loss \times \partial_{\mathbf{W}^{(l)}} \mathbf{X}^{(l+1)} \\ &= \partial_{\mathbf{X}^{(l+1)}} loss \times \partial_{\mathbf{W}^{(l)}} \sigma(\mathbf{X}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}) \\ &= \partial_{\mathbf{X}^{(l+1)}} loss \times \sigma'(\mathbf{X}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}) \times \mathbf{X}^{(l)}\end{aligned} \tag{3.3}$$

$$\begin{aligned}\partial_{\mathbf{b}^{(l)}} loss &= \partial_{\mathbf{X}^{(l+1)}} loss \times \partial_{\mathbf{b}^{(l)}} \mathbf{X}^{(l+1)} \\ &= \partial_{\mathbf{X}^{(l+1)}} loss \times \partial_{\mathbf{b}^{(l)}} \sigma(\mathbf{X}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}) \\ &= \partial_{\mathbf{X}^{(l+1)}} loss \times \sigma'(\mathbf{X}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)})\end{aligned} \tag{3.4}$$

3.2.2 算法实现

算法实现主要分为以下几个功能部分：

定义损失函数、激活函数等

```
def sigmoid(x):
        return 1 / (1 + np.exp(-x))
def grad_sigmoid(x):
        return sigmoid(x) * (1 - sigmoid(x))
def LeakyReLU(x):
        return np.where(x > 0, x, 0.00 * x)
def grad_LeakyReLU(x):
        return np.where(x > 0, 1, 0.00)
def softmax(x):
        return np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True)
def grad_softmax(x):
        return (1 - softmax(x)) * softmax(x)
def linear(X, w, b):
        return np.matmul(X, w) + b
```

数据归一化和参数初始化

```
def init_weights() -> None:
    w_input[...] = np.random.normal(size=(input_size, hidden_size))
    w_hidden_arr[...] = np.random.normal(size=(number_hidden_layers-1, hidden_size,
        hidden_size))
    w_output[...] = np.random.normal(size=(hidden_size, output_size))
    return None
......
train_pd = pd.read_csv("train.csv")
train_feature = train_pd.values.copy() [:, 1:] / 255.0
train_labels = np.eye(numclass)[ train_pd.values.copy() [:, 0] ].reshape(len(
    train_pd.values) // batch_size, batch_size, numclass)
```

```
test_pd = pd.read_csv("test.csv")
test_feature = test_pd.values.copy() / 255.0
```

前向传播 由公式3.1前向传播的代码实现：

```
def feed_forward() -> None:
    hidden_layers[0, ...] = linear(input_layer, w_input, b_input)
    for i in range(0, number_hidden_layers-1, 1):
        hidden_layers[i+1, ...] = activation(linear(hidden_layers[i], w_hidden_arr[
            i], b_hidden_arr[i]))
    output_layer[...] = linear(hidden_layers[number_hidden_layers-1], w_output,
        b_output)
    softmax_output[...] = softmax(output_layer)
    return None
```

反向传播 由公式3.2、3.3、3.4反向传播的代码实现：

```
def back_propagation() -> None:
    output_layer_grad = (softmax_output - train_label)
    w_output_grad = np.matmul(hidden_layers[number_hidden_layers-1].T,
        output_layer_grad)
    hidden_layers_grad[number_hidden_layers-1] = np.matmul(output_layer_grad,
        w_output.T)
    b_output_grad = output_layer_grad
    for i in range(number_hidden_layers-1, 0, -1):
        w_hidden_grad_arr[i-1] = np.matmul(hidden_layers[i-1].T, hidden_layers_grad
            [i] * grad_activation(linear(hidden_layers[i-1], w_hidden_arr[i-1],
            b_hidden_arr[i-1])))
        b_hidden_grad_arr[i-1] = hidden_layers_grad[i] * grad_activation(linear(
            hidden_layers[i-1], w_hidden_arr[i-1], b_hidden_arr[i-1]))
        hidden_layers_grad[i-1] = np.matmul(hidden_layers_grad[i] * grad_activation
            (linear(hidden_layers[i-1], w_hidden_arr[i-1], b_hidden_arr[i-1])),
            w_hidden_arr[i-1].T)
    w_input_grad = np.matmul(input_layer.T, hidden_layers_grad[0] )
    b_input_grad = hidden_layers_grad[0]
    return None
```

更新参数

```
def update_weights() -> None:
    w_input[...] -= learning_rate * w_input_grad.sum(axis=0) / batch_size
    b_input[...] -= learning_rate * b_input_grad.sum(axis=0) / batch_size
    for i in range(0, number_hidden_layers-1, 1):
        w_hidden_arr[i, ...] -= learning_rate * w_hidden_grad_arr[i].sum(axis=0) /
            batch_size
        b_hidden_arr[i, ...] -= learning_rate * b_hidden_grad_arr[i].sum(axis=0) /
            batch_size
    w_output[...] -= learning_rate * w_output_grad.sum(axis=0) / batch_size
```

```
8       b_output[...] -= learning_rate * b_output_grad.sum(axis=0) / batch_size
9       return None
```

以上代码中，隐层的数量、大小，激活函数的种类都是参数化的设计，为的是可以灵活调整，比较不同参数的效果。

3.2.3 参数调优与模型评估

对模型的隐层数量和神经元数量在自己划分的数据集上进行调优和测试，得到的数据如下：

Number of hidden layers	2	**3**	4	5	6
Acc on test set(%)	91.52	**93.21**	91.38	56.31	9.88
Loss on test set	4.32	**4.56**	5.60	41.19	39.72

表 3.3: Lr = 0.02, 20 Epochs, Size of hidden layers 256

Size of hidden layers	64	128	192	256	**320**	512
Acc on test set(%)	87.95	91.31	92.02	92.04	**92.90**	92.04
Loss on test set	8.55	4.67	4.17	3.65	**4.09**	4.08

表 3.4: Lr = 0.02, 20 30 Epochs, Number of hidden layers 2

根据上述结果，选择 Lr = 0.02, 50 Epochs, Size of hidden layers 320，Number of hidden layers 3，对 Kaggle 上的数据集进行分类，结果如下：

图 3.3: MLP 测试结果

4 实验环境与平台

4.1 硬件环境

由于本次试验采用 *numpy* 数学库，无法使用 GPU 进行计算，我选择了华科七边形超算队的服务器集群进行实验，大规模并行计算能搞保证训练的速度，与这次实验有关的关键硬件配置如下：

名称	型号	数量
Server	Inspur NF5280M6 2U rack	4
CPU	Intel(R) Xeon(R) Gold 6338	2
DRAM	Micron DDR4 3200MT/s 32G RECC	32
Infiniband	Mellanox ConnectX-4	1

表 4.1: 硬件环境

4.2 软件环境

名称	版本
OS Distribution	Ubuntu x86_64 Server 20.04.6 LTS
MPI4Py	3.1.4
MPI	OpenMPI 4.0.2
Python	CPython 3.11.5
NumPy	1.23.5
Slurm	23.11.4

表 4.2: 软件环境

5 结果与分析

每种模型的结果已经在第三章相应的小节中给出，这里不再赘述。

5.1 两种模型的比较

相同点：

- 都可以用于分类任务：MLP 和 k-NN 都是监督学习算法，可以用于诸如 MNIST 手写数字识别的分类任务;
- 都需要训练：MLP 和 k-NN 都需要在有标记的训练集上进行训练，以便在测试集上进行预测。

不同点：

- 学习方式不同：MLP 是参数模型，在训练前会预设模型参数（如隐层神经元的数目），而 k 近邻算法在预测时直接从训练集中学习。
- 预测时间和计算量不同：MLP 的预测时间通常比 k-NN 短，因为 MLP 只需要前向传播一次就可以得到预测结果，而 k-NN 需要计算测试样本与所有训练样本的距离。
- 性能不同：在 MNIST 数据集上，MLP 通常可以达到更高的准确率，因为它可以学习到数据的深层次特征。而 k-NN 的性能则较为依赖于选择的邻居数量 k 和距离度量方式。

理论上来说，MLP 算法的准确率还应该更高点，因为时间的原因，没有去尝试暂退法、Batch Normaliztion、更换激活函数等其他改进性能等方法。这是一个小小的遗憾。

6 个人体会

这次实验让我受益良多，主要有以下几点：

- k-NN 算法是一个没有显示学习过程的算法，计算量非常大。以前我只在 C 语言和 FORTRAN 中写过 MPI 程序。看到自己写的代码能在两分钟跑完原来三个多小时才能跑完的任务，不禁感概并行计算的强大。
- MLP 的反向传播是实验中的一个难点。从繁杂的公式推导，到用 *NumPy* 实现反向传播算法，每一步对我来说都是一个巨大的挑战。当然，在这个过程中我也迅速掌握了 MLP 的结构和计算过程。看到自己写的模型的损失函数逐步从 120 收敛 20，非常有成就感。
- 在撰写论文的过程中也掌握了很多相当有用的技能，如绘制 MPI 通信示意图的过程中，我学会了 Draw.io 这个软件的使用。编写 LaTeX 的时候学会了如何插入公式、算法、表格，并调整排版使之变得优美。
- 在实现算法的过程中，我体会到了 *NumPy* 数学库的强大，他自带的排序算法、计数算法、选择算法提供了极强的算法表达能力，为实现 k-NN 算法和 MLP 网络提供了很大的便利。

最后要感谢何琨老师在课堂上的认真负责细心的教学，让我深入理解了机器学习的各个领域，掌握了机器学习的必备技能。

参考文献

[1] 周志华. 机器学习: 第 3 章. 清华大学出版社, 2016.

[2] 李航. 统计学习方法: 第 3 章. 清华大学出版社, 2012.