数据结构与卷积神经网络代码实现

实验名称：数据结构大作业

姓名：陈祥蔚

学院：数学与统计学院

专业：统计学

学号：**320190930061**

2023 年 12 月 29 日

摘要

本次大作业主要完成了以下内容：编写两个数据结构、将待学习数据加载到数据结构中、通过神经网络学习、以及比较不同的数据结构的性能。

我选择的两个数据结构是线性表和链表。我编写了这两个数据结构，并且添加了不同的成员变量和成员函数，用来实现不同的功能。

由于无论是什么数据结构，其提供给神经网络用于学习的数据是一样的，而神经网络的训练轮次多，训练量大，因此我选择在训练前的数据处理部分比较不同的数据结构的优劣。评价大致分为三个任务上时间和空间共六个指标，每个指标均进行了 12 次实验，最后绘图比较。

第一个任务为将图片信息加载到数据结构中。在这个任务中，链表在时间和空间上都不占优，链表所花时间的均值为 0.002141 秒，所占用内存均值为 0.3197MB，线性表所花平均时间为 0.00052 秒，所占用内存均值为 0.0915MB，链表所花时间为线性表的 4 倍左右，所占用内存是线性表的 3 倍左右。

第二个任务为将训练数据和训练标签匹配。在这个任务中，链表在时间上表现优于线性表，所花时间均值为 20.782 秒，而线性表所花时间为 23.4787 秒，链表比线性表快了 12.98% 左右，但是这一优势可以看出是牺牲了空间所带来的，因为链表的占用内存均值为 115.4333MB，而线性表占用内存均值为 114.7891MB，不过这一差距并不明显，线性表只比链表少了 0.056% 左右，因此可以设想，如果针对更加大量的数据，链表在时间上的优势完全可以弥补空间上的劣势。

第三个任务为将数据转换为神经网络能够接受的类型。在这个任务中，链表依然都不占优，其花费时间的均值为 0.000583，占用内存的均值为 0.124MB，而线性表的所花时间均值为 0.000124，占用内存均值为 0.0169MB，链表花费时间约为线性表的 5 倍，占用的空间为线性表的 7 倍左右。

在神经网络中，我使用了 5 个卷积层，前 4 个卷积层每个后面跟随了一个最大池化层，第 2、4、5 个卷积层后增加了随机失活层，最后添加了 2 层全连接层。除了最后一层因为是分类任务所以使用了 Softmax 层外，其余均使用的是 ReLU 激活函数。在训练过程中，采用了 SGD 优化器，开始设置了 100 个训练轮次，而后经过测试发现在 175 个训练轮次后训练准确率趋于稳定，测试准确率在 72.5% 上下。

关键词：深度学习，人工智能

目录

1 任务描述

本次作业旨在实现以下目标：
1、选择两种数据结构，表示肺部 X 射线图像和相关的疾病标签
2、使用 MindSpore 构建 CNN 模型，进行图像分类和疾病检测任务
3、将肺部 X 射线图像加载到数据结构中，包括每个图像的标签，表示是否有特定的疾病。
4、训练模型，以自动检测疾病病灶的位置和类型
5、评估模型的性能
6、分析和比较不同数据结构对于模型性能和数据处理效率的影响

2 数据结构

针对图像文件在管理系统中的特性，以及考虑到图像在卷积神经网络中的加载方式，我们采取的管理并储存图像的方式是线性表和链表。

2.1 线性表

在线性表中，我们储存的是每个图片在计算机中的地址，由地址对于具体的图片进行管理。在使用 CNN 网络进行学习的时候，通过传入地址读取具体的图片，并且对于图片进行学习。因此，我们实现了以下的特殊方法：初始化、迭代、求长度；以及以下的自定义方法：返回匹配项下标、添加元素、删除元素、插入元素、查找元素。

LinearTable 类：表示线性表，包含以下主要功能：
__init__(self)：初始化一个空的线性表，使用列表 datalist 存储数据。
__iter__(self)：使线性表可迭代，可以直接迭代线性表中的元素。
__len__(self)：返回线性表中元素的数量。
index(self, data)：返回第一个匹配项的下标，如果不存在则返回 -1。
add_data(self, data)：向线性表末尾添加一个新的数据项。
find_data(self, data)：查找线性表中所有匹配指定数据项的下标，返回一个列表。
insert_data(self, location, data)：在指定位置插入新的数据项。
delete_data(self, data)：从线性表中删除第一个匹配指定数据项的元素。

2.2 双向链表

在 LinkedList.py 中，我们实现了一个基本的双向链表结构，用于管理图像文件的路径。以下是每个类的主要功能：

ImageNode 类：表示链表中的节点，每个节点包含一个图像文件的路径。节点还包括指向下一个节点和上一个节点的指针。
ImageLinkedList 类：表示双向链表，包含以下主要功能：
add_image(image_path)：向链表末尾添加一个新的图像节点。
delete_image(image_path)：从链表中删除包含指定图像路径的节点。
search(image_path)：在链表中查找包含指定图像路径的节点，返回该节点。
is_empty()：检查链表是否为空。
length()：返回链表中节点的数量。
display_images()：打印链表中所有图像节点的路径。
__getitem__(index)：通过索引获取链表中特定位置的图像路径。
to_list()：将链表中的元素输入到列表中并返回

2.3 数据加载与测量

与数据结构和数据加载相关的文件是 dshwwithpytorch 文件夹中的 datawithlt.py 和 datawithll.py 文件，前者是使用线性表，后者是使用链表。

在这两个文件中，对数据进行了加载和预处理，并在这个过程中调用了例如 time() 对于加载时间进行测量，以及导入 psutil 包对于内存进行测量。直接运行这两个文件夹即可得到初步的结果如下。

```
D:\anaconda\envs\myenv\python.exe D:\pythonstarter\dshwwithpytorch\datawithlt.py
将图片信息加载到数据结构中的加载时间: 0.0005609989166259766 秒
添加数据后内存占用增加: 0.05859375 MB
将训练数据和训练标签匹配的加载时间: 22.86709237098694 秒
添加数据后内存占用增加: 114.33203125 MB
将线性表中的列表转换成array的加载时间: 0.0 秒
添加数据后内存占用增加: 0.01953125 MB
保存完毕

Process finished with exit code 0
```

图 1: 线性表的运行时间和占用内存

```
D:\anaconda\envs\myenv\python.exe D:\pythonstarter\dshwwithpytorch\datawithll.py
将图片信息加载到数据结构中加载时间: 0.0020003318786621094 秒
添加数据后内存占用增加：0.31640625 MB
将训练数据和训练标签匹配的加载时间: 21.732478857040405 秒
添加数据后内存占用增加：115.24609375 MB
使用链表中的内置函数将链表转换成列表后转为array数组的时间: 0.0005145072937011719 秒
添加数据后内存占用增加：0.1171875 MB
保存完毕

Process finished with exit code 0
```

图 2: 链表的运行时间和占用内存

2.4 定量分析

2.4.1 方法和评价标准

在实验中，为了定量分析，准确起见，我将每个数据结构加载了十二次，比较不同的数据结构在不同的指标上的差别，并分析出现这样的结果的原因。与之相关的文件是 dshwwithpytorch 文件夹里面的 comparell.py，comparelt.py，myplot.py，time&memory 文件夹和 plots 文件夹。在 comparell.py，comparelt.py 两个文件中我分别用两种数据结构运行了 12 次，获得实验数据。而后我将数据以列表的形式储存到 time&memory 文件夹中。

最后，我在 myplot.py 文件中绘制了比较的图像，而后将图像导出保存至 plots 文件夹中。分析的指标分为三个进程，共有六个，三个为时间指标，三个为内存指标。它们是：(1) 将图片信息加载到数据结构中的加载时间和添加数据后内存占用。(2) 将训练数据和训练标签匹配的加载时间和内存占用。(3) 将各个数据结构中的数据转换成 array 数组的加载时间和内存占用。在实验时选取的数据集是训练集。

选择这三项进程的六个指标原因如下。

首先，由于创建数据结构的目的就是储存、管理数据，因此选择将图片信息加载到数据结构中的加载时间和添加数据后内存占用是非常自然的事情。

其次，由于分类问题需要将数据和标签匹配，因此数据结构在匹配的速度和占用内存大小的性能就成为了评价数据结构的重要指标。

最后，不管什么数据结构，由于神经网络只能接受特定的数据类型，因此我们都需要将数据结构中的数据转换数据类型后才能进行学习，因此我们选择了将数据结构中的数据转换成 array 数组的加载时间和内存占用作为两个评价指标。

2.4.2 将图片信息加载到数据结构中的加载时间和内存占用

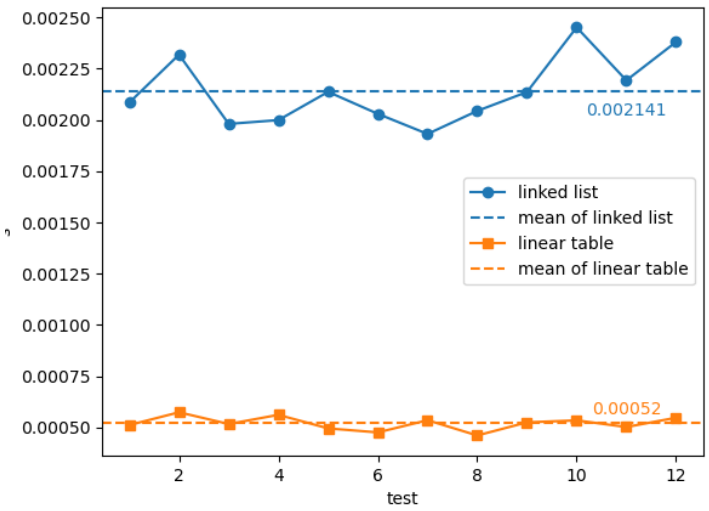

图 3: 将图片信息加载到数据结构中的加载时间

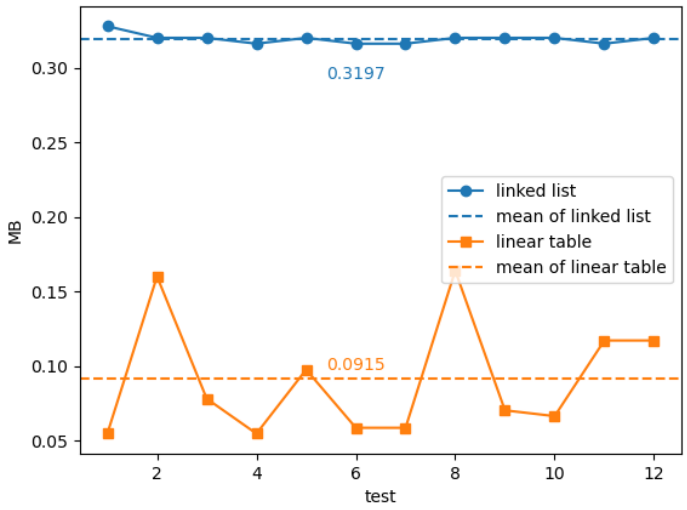

图 4: 将图片信息加载到数据结构中的内存占用

根据图表可以看出，从时间上来看，链表花费的时间是线性表的 4 倍左右，链表占用的内存是线性表的 3 倍左右。据分析，这样的结果产生的原因是在加载阶段，链表的每一个节点要比线性表多创建了头指针和尾指针，因此不管是加载时间还是占用内存，链表都比线性表要花更多的时间。

2.4.3 将训练数据和训练标签匹配的加载时间和内存占用

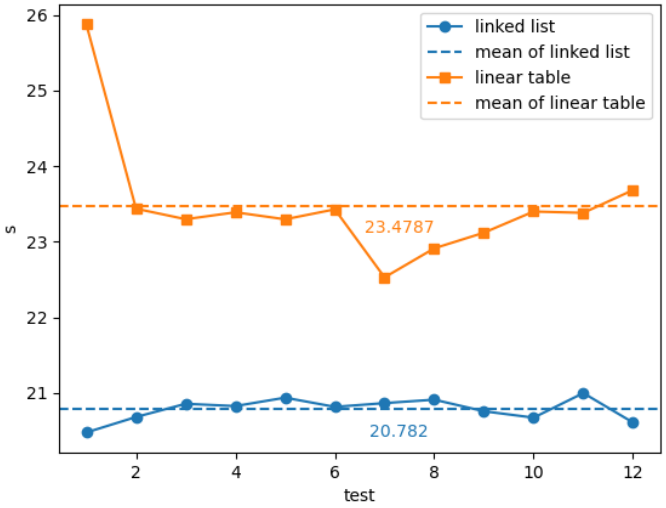

图 5: 将训练数据和训练标签匹配的加载时间

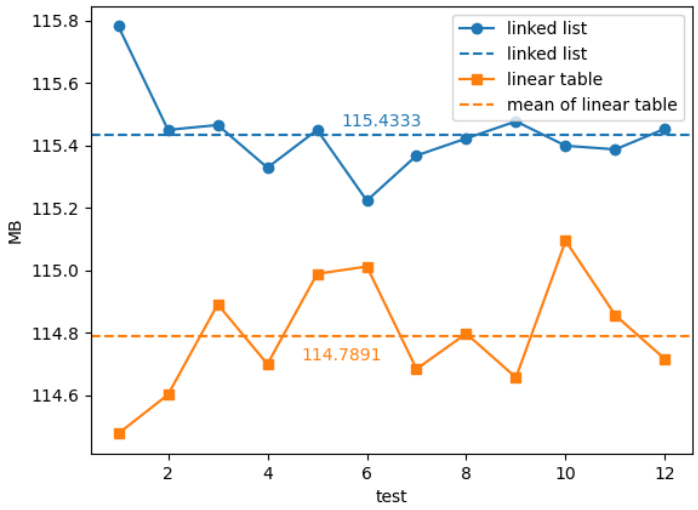

图 6: 将图片信息加载到数据结构中的内存占用

在这两个指标中，链表的加载时间普遍低于线性表，内存占用普遍高于线性表，但是差别并不明显。原因可能链表可以动态开辟存储空间，因此在匹配的过程中时间较短，但是由于每个节点都有指针，因此总体内存增大。

2.4.4 将各个数据结构中的数据转换成 array 数组的加载时间和内存占用

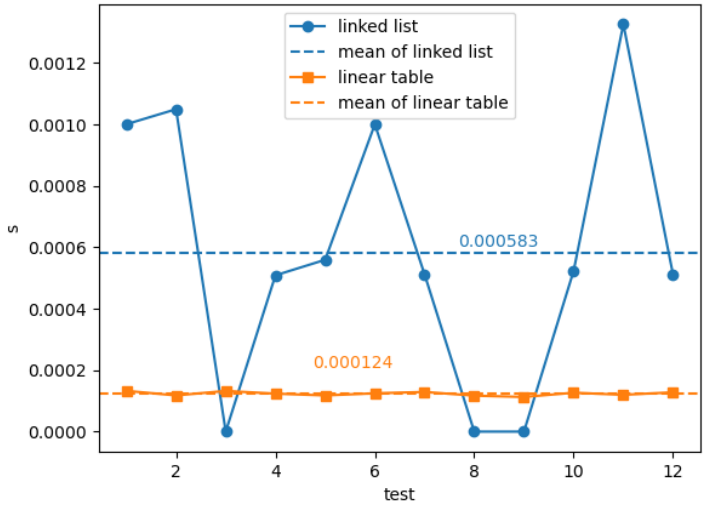

图 7: 将各个数据结构中的数据转换成 array 数组的加载时间

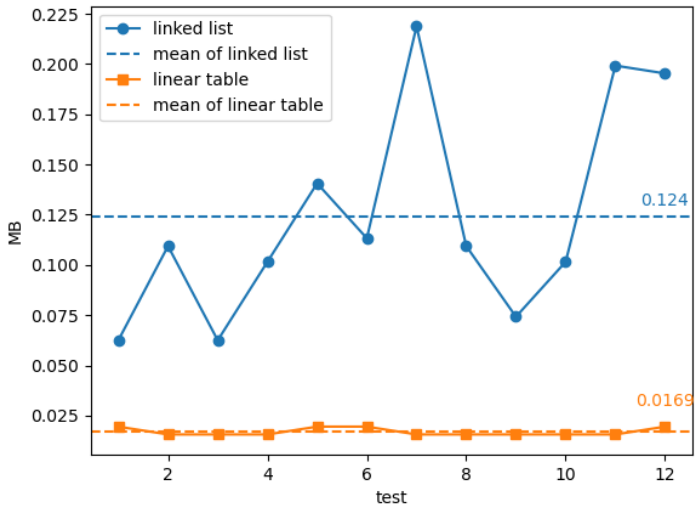

图 8: 将图片信息加载到数据结构中的内存占用

在转换数组的这两个指标中，链表的加载时间是线性表的 6 倍左右，占用内存是线性表的 7 倍。原因应当如下：线性表在储存信息的时候就是采取的列表形式，因此按照线性表的顺序逐一转换即可；而链表由于分散储存，因此需要在链表中写一个转换成线性表的函数，才能将链表中的元素先储存进一个列表中，而后才能进一步转换为数组。

2.5 误差

首先在实验的时候犯了一个错误，如图是将训练数据和训练标签匹配的加载时间的图像。从图中可以看出，第一次实验后内存便基本为零了。猜想的原因是，线性表的实验过程中是用 for 循环进行实验的，但是可能在这个过程中需要清空内存，但是没有清空，于是就出现了这样的情况。

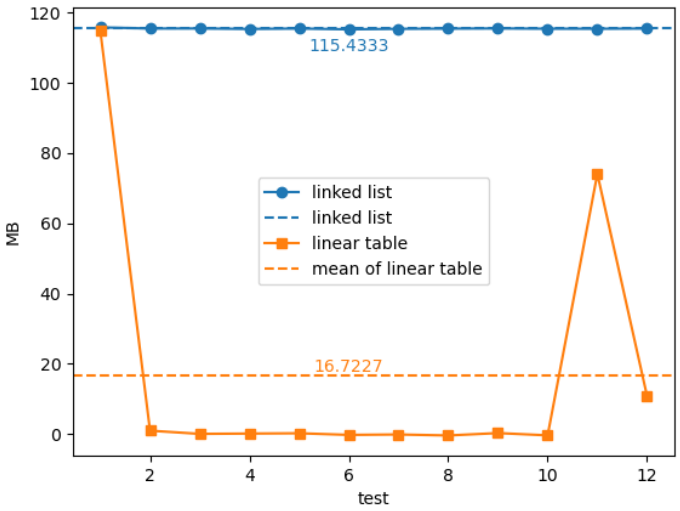

图 9: 错误图样

因此保险起见，我后来分别将线性表和链表的单次实验进行了 12 次，并手动将数据储存进线性表中。原始数据可以在 comparell.py 和 comparelt.py 中看到，也就是上面几节中图片展示的样子。但是明显可以看出，有的数据波动过大，例如 2.4.4 节中的图 7，图 8，目前并不是很清楚原因是什么。

另外，在实验过程中偶尔会出现时间或内存为 0.0 的状况，时间为 0.0 的原因应该是使用了 time 包中的 time() 函数进行计时，后来换成 perf_counter() 函数后就不会出现这样的状况了。

3 数据集

数据集来源于 https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia/code?datasetId=17810&sortBy=voteCount，该数据集分为 3 个文件夹（train、test、val），并包含每个图像类别（肺炎/正常）的子文件夹。有 5,863 张 X 射线图像 (JPEG) 和 2 个类别（肺炎/正常）。

胸部 X 射线图像（前后位）选自广州市妇女儿童医疗中心 1 至 5 岁儿童患者的回顾性队列。所有胸部 X 射线成像均作为患者常规临床护理的一部分进行。

为了分析胸部 X 射线图像，首先通过删除所有低质量或不可读的扫描来筛选所有胸部 X 射线照片以进行质量控制。然后，两位专家医生对图像的诊断进行了分级，然后才允许训练人工智能系统。为了解决任何评分错误，评估集还由第三位专家进行了检查。

4 预处理

为了能够进行 CNN 的训练，我们需要对于现有的数据进行一些预处理。预处理的文件是 mydata.py，处理后的数据存放在 data 文件夹中，该文件夹中包括了 test_data.npy，test_data.npy，train_label.npy，train_data.npy，val_data.npy，val_label.npy。

预处理包括了以下步骤：

1、定义获取图片路径的函数：get_photo_paths 函数接受一个文件夹路径作为参数，检查该文件夹是否存在，获取文件夹中的所有文件，并返回包含这些文件路径的列表。

2、定义加载训练数据的函数：get_training_data 函数接受两个线性表对象作为参数，用于存储数据和对应的标签。函数根据文件路径加载图像数据，将其调整大小，然后添加到数据线性表中，并根据文件路径中是否包含特定标签确定对应的标签。

3、初始化线性表和获取图片路径列表：创建三个线性表对象 train、test、val，并通过 get_photo_paths 函数获取训练、测试和验证集的图像文件路径。

4、将图片信息加载到线性表中：使用 add_data 方法将图像信息加载到训练、测试和验证集的线性表中。

5、建立用于存放标签的线性表：创建三个线性表对象 train_data_label、test_data_label、val_data_label，用于存放对应的标签信息。

6、获取训练数据：调用 get_training_data 函数获取训练、测试和验证集的图像数据，并将其归一化。

7、预处理：将图像数据的形状修改为在代码中定义的 img_size（150）。

8、转换数据类型：将图像数据和标签的数据类型转换为适用于 MindSpore 框架的数据类型。

9、修改标签的形状：使用 np.squeeze 函数将标签的形状进行修改。

10、保存文件：将处理后的图像数据和标签保存为.npy 文件，以备后续在 MindSpore 中使用。

5 CNN

1、卷积层（Convolutional Layers）：

使用了五个卷积层，每一层包括卷积操作（nn.Conv2d）、批量归一化（nn.BatchNorm2d）、激活函数（ReLU）和最大池化（nn.MaxPool2d）。卷积核大小为 (3, 3)，步长（stride）为 1，padding 模式为'same'，保证了卷积后特征图大小不变。

2、Dropout 层：

在第二、四、五个卷积层后使用了 Dropout 操作，有助于防止过拟合。

3、全连接层（Fully Connected Layers）：

包括两个全连接层，每一层包括全连接操作（nn.Dense）、激活函数（ReLU）和 Dropout 操作。第一个全连接层的输出大小为 128，输入大小为 6400（由 nn.Flatten 层展平得到）。第二个全连接层的输出大小为 3，对应于分类的类别数量。

4、激活函数：

使用了 ReLU 激活函数，除了最后一层，最后一层使用了 Softmax 激活函数进行多类别分类。打印操作：

在网络中使用了 P.Print() 操作，该操作用于在计算过程中打印张量的形状信息。网络的整体结构是卷积层的堆叠，通过不断降低特征图的大小，最终通过全连接层输出分类结果。Dropout 层有助于提高模型的泛化能力。

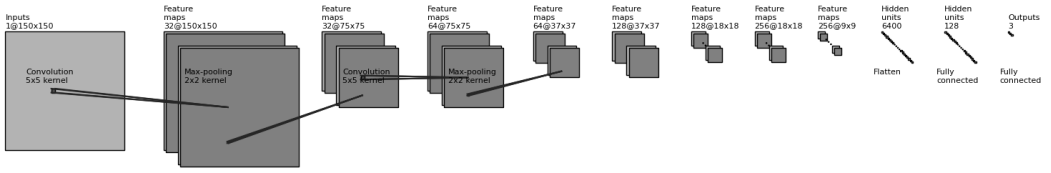

图 10: CNN 网络图像

6 训练

1、优化器和损失函数的设置：使用随机梯度下降 (SGD) 作为优化器，学习率为 0.01。使用交叉熵损失函数 (CrossEntropyLoss)。

2、训练循环：设置了总共 100 个训练周期 (epochs)。在每个周期内，遍历训练数据集，并进行前向传播、反向传播和优化。输出每个步骤的训练进度信息，包括当前周期和步骤。

3、模型保存：在训练完成后，将模型保存到文件’model/model.pth’。

4、损失和准确率的记录：记录每个训练周期的总损失和准确率。绘制训练损失和准确率随着训练周期变化的图表。

5、100 个轮次训练结果为：总损失 803.9553998112679，准确率 0.9351993865030674

图像如下：

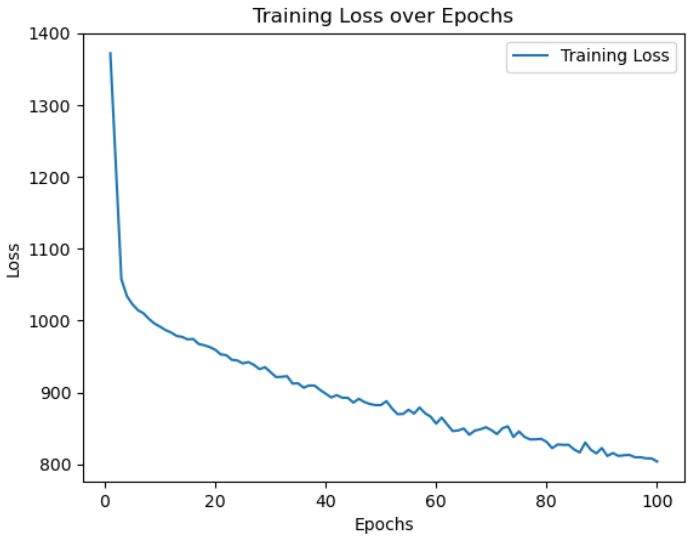

图 11: 损失值随训练轮次变化

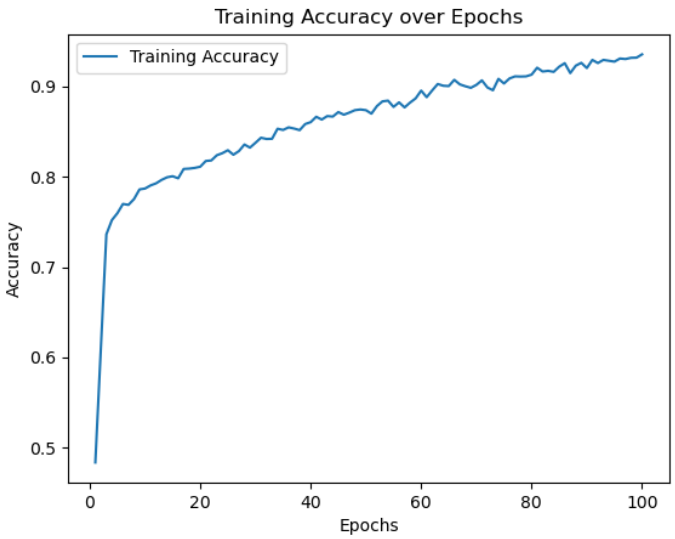

图 12: 准确率随训练轮次变化

7 测试

在测试中，我首先比较了在 50、100、150、200、250、300 轮次中损失值和准确率的变化，可以发现在 150-200 轮次，损失值和准确率就开始不再变化。

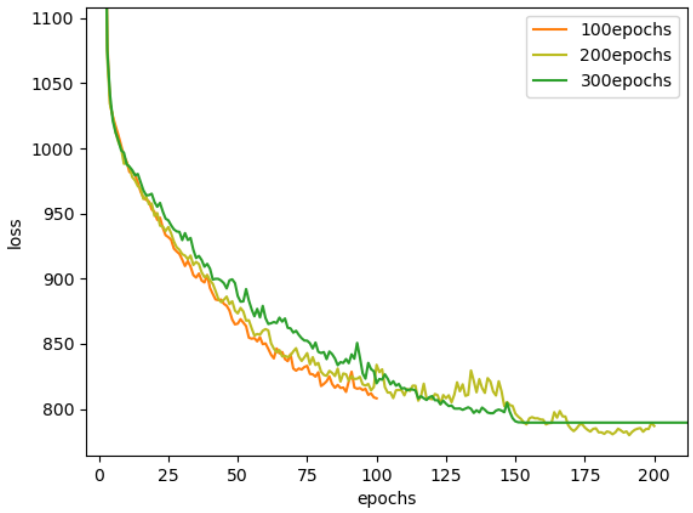

图 13: 损失值随训练轮次变化

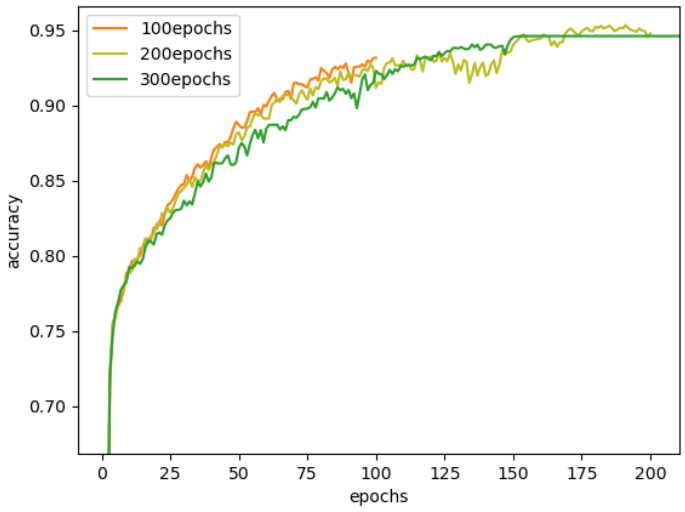

图 14: 准确率随训练轮次变化

因此，我们将准确率的测试放在 175 次进行。结果如图所示。

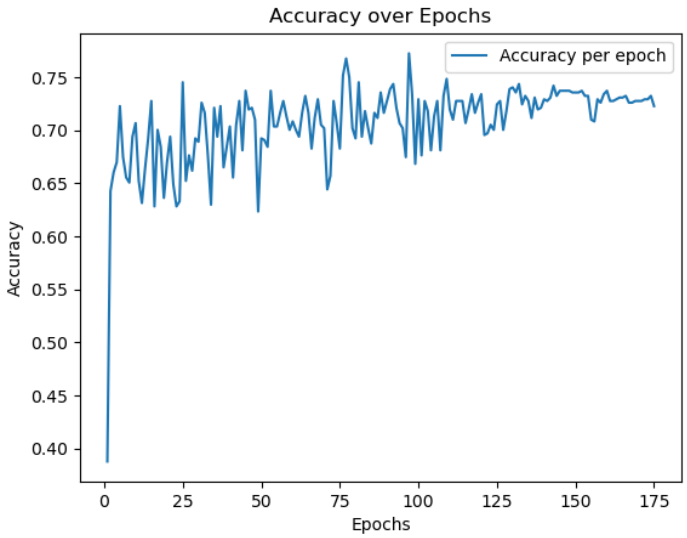

图 15: 175 轮次准确率随轮次变化

从图片中可以看出，准确率在 70% 上下徘徊，并不稳定，最后可能是收敛到 72.5% 左右，相对来说并不高。模型还有很大的进步空间。

8 讨论

本次作业中仍然存在着各种不足。

1、在运行 mindspore 的时候并不能十分完美地运行出来。神经网络的结构没有问题，在训练的时候发现出了一些问题。在训练的时候 loss 值一直不收敛。后来发现标签不同，loss 值收敛到不同的值，并且最终的测试准确率一直在 38%，保持不变。排查原因时，我认为数据集和预处理没有问题，因为用 pytorch 写的网络是可以运行出来的，并且 MindSpore 中的训练方法是导入 Model 包，而后直接调用里面的 model_train 函数进行训练，因此不太清楚是哪里出现问题了。另外一个重要的原因是 MindSpore 在 Windows 上不支持 GPU 计算，用 CPU 计算会很慢。

2、代码的组织不好。很多时候直接在作业的文件夹下新建 py 文件来写一个程序，因此一方面看起来比较乱；并且很多时候导入文件或者保存文件选择的是绝对地址，因此可能不便于在别的计算机上测试。不过还好不管是数据集还是程序文件都基本上是直接在 D 盘中的，因此就算是绝对地址的话导入也比较方便。

3、实验代码太过冗长，符号不简洁。因为实验中涉及到两种数据结构和多个指标，因此如何合理地组织符号也是一个问题。上一个问题提出的组织架构不好，一方面也是代码太过冗长的原因，有的时候写的时候没有规划好，写完了一个数据结构的相关部分后，发现另外一个不能在同一个文件中运行，因此只好另写一个文件，导致整个文件夹比较繁杂。

9 总结

在本次作业中，研究了线性表和链表就加载数据等方面的性能，综合比较了其在时间和空间上在各个任务中的表现优劣：将数据加载到数据结构中时由于链表要额外创建指针，因此从时间和空间上来说均比线性表表现差一些；将数据彼此匹配时，链表在时间上表现比线性表优异，分析应该是用空间换时间的原因，其相应的空间占用得更多一些；针对数据转换任务，链表在时间和空间上都并不如线性表，原因应该是链表一方面储存了指针，一方面比链表要额外经过一个转换成列表的步骤。

最后，我设计了一个神经网络，并且让神经网络学习，而后进行了不同轮次的训练，比较出训练准确度不再增加的轮次后对于准确度进行测试，完成了神经网络的学习，效果有待提升，可能是网络架构的问题，也可能是优化器等各个方面的问题。