哈尔滨工业大学

模式识别与深度学习

实验一：深度学习框架熟悉

姓　　名：刘天瑞

院　（系）：未来技术学院

专　　业：视听觉信息处理

学　　号：7203610121

指导教师：左旺孟

提交日期：2023.5.2

摘 要

本次实验为模式识别与深度学习课程中深度学习模块部分的实验一：即对深度学习框架熟悉，其主要任务是搭建一个多层感知机模型 MLP，并在 MNIST 手写体数字数据集上对其进行训练与测试评价。除此之外，在本次实验中，我搭建了一个包含 2 个全连接层与 1 个 softmax 层的 MLP，并在 MNIST 手写体数字数据集上取得了 98.44%的较为不错的准确率。

关键词： 模式识别与深度学习，MLP，MNIST

目 录

一、深度学习框架与实验环境.....4

二、主要研究内容以及实验背景知识.....5

2.1 MLP 算法简介.....5

2.2 MINIST 数据集介绍.....5

三、实验详细过程.....6

3.1 加载并读取数据集.....6

3.2 搭建 MLP 网络结构.....6

3.3 定义损失函数.....7

3.4 定义优化器.....7

3.5 MLP 模型训练过程.....8

四、实验结果与分析.....8

一、 深度学习框架与实验环境

本次实验采用的深度学习框架是 Python 中强大的深度学习库 Pytorch，整个实验是在 Visual Studio + Pip 环境下完成的。Pip 是一个开源的 Python 发行版本，可以很方便地安装许多深度学习所需要的模块包，而 Visual Studio 则是一个功能强大的 IDE，可以在其中完成 python 代码编写深度学习过程、进行训练测试等深度学习环节。由于我在大二时参加过美赛有接触到一些深度学习相关的工具知识，因此在本次实验中的配置环境过程相对得心应手。配置环境的具体流程比较繁琐，查看系统 Pytorch 库以及 cuda 版本如下图 1 所示（英伟达表示 cuda 可升级到的最高版本）：

```
C:\Users\刘天瑞>python
Python 3.9.7 (tags/v3.9.7:1016ef3, Aug 30 2021, 20:19:38) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> print(torch.__version__)
2.0.0+cu118
>>> ^Z
```

```
C:\Users\刘天瑞>nvidia-smi
Sat Apr 29 18:24:48 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 531.41                 Driver Version: 531.41       CUDA Version: 12.1     |
|-------------------------------------+----------------------+----------------------+
| GPU  Name                      TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                     |                      |               MIG M. |
|=====================================+======================+======================|
|   0  NVIDIA GeForce RTX 3060 L...  WDDM | 00000000:01:00.0  On |                  N/A |
| N/A   39C    P8              15W /  N/A|   1491MiB /  6144MiB |     21%      Default |
|                                     |                      |                  N/A |
+-------------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      2884    C+G   ...on\112.0.1722.58\msedgewebview2.exe    N/A      |
|    0   N/A  N/A      4944    C+G   ...crosoft\Edge\Application\msedge.exe    N/A      |
|    0   N/A  N/A      5324    C+G   ...\cef\cef.win7x64\steamwebhelper.exe    N/A      |
|    0   N/A  N/A      9704    C+G   ...7.0_x64__w2gh52qy24etm\Nahimic3.exe    N/A      |
|    0   N/A  N/A      9900    C+G   C:\Windows\explorer.exe                   N/A      |
|    0   N/A  N/A     10908    C+G   ...nt.CBS_cw5n1h2txyewy\SearchHost.exe    N/A      |
|    0   N/A  N/A     11620    C+G   ...5n1h2txyewy\ShellExperienceHost.exe    N/A      |
|    0   N/A  N/A     13316    C+G   ...2txyewy\StartMenuExperienceHost.exe    N/A      |
|    0   N/A  N/A     16712    C+G   ...CBS_cw5n1h2txyewy\TextInputHost.exe    N/A      |
|    0   N/A  N/A     17748    C+G   ...GeForce Experience\NVIDIA Share.exe    N/A      |
|    0   N/A  N/A     19364    C+G   ...GeForce Experience\NVIDIA Share.exe    N/A      |
|    0   N/A  N/A     20712    C+G   ...1.0_x64__8wekyb3d8bbwe\Video.UI.exe    N/A      |
|    0   N/A  N/A     22552    C+G   ...t.LockApp_cw5n1h2txyewy\LockApp.exe    N/A      |
|    0   N/A  N/A     25060    C+G   ...4036\office6\promecefpluginhost.exe    N/A      |
|    0   N/A  N/A     25076    C+G   ...__8wekyb3d8bbwe\WindowsTerminal.exe    N/A      |
|    0   N/A  N/A     25856    C+G   ...siveControlPanel\SystemSettings.exe    N/A      |
|    0   N/A  N/A     26908      C   ...Programs\Python\Python39\python.exe    N/A      |
|    0   N/A  N/A     26948    C+G   ...__8wekyb3d8bbwe\WindowsTerminal.exe    N/A      |
|    0   N/A  N/A     30532    C+G   ...22\Community\Common7\IDE\devenv.exe    N/A      |
|    0   N/A  N/A     31092    C+G   ...ft Office\root\Office16\WINWORD.EXE    N/A      |
|    0   N/A  N/A     31864    C+G   ...4036\office6\promecefpluginhost.exe    N/A      |
+-----------------------------------------------------------------------------+
```

图 1

二、 主要研究内容以及实验背景知识

2.1 MLP 算法简介

多层感知机（MLP，Multilayer Perceptron），也叫做人工神经网络（ANN，Artificial Neural Network），它是一种前向结构的人工神经网络，其中包含输入层、输出层以及多个隐藏层。MLP 神经网络的不同层之间是全连接的，即上一层的任何一个神经元与下一层的所有神经元都有连接。如下图 2 所示举例即为具有一层隐藏层的 MLP 网络结构：

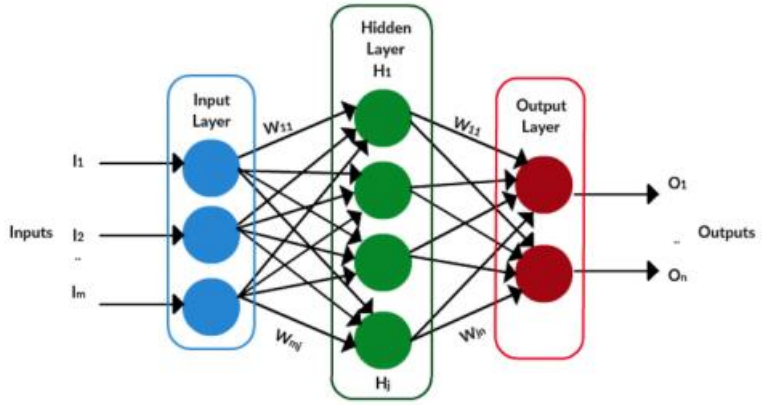

图 2

2.2 MNIST 数据集介绍

MNIST 是一个内容为手写体数字的图片数据集，该数据集由美国国家标准与技术研究所（National Institute of Standards and Technology, NIST）发起整理，一共统计了来自 250 个不同的人其手写数字图片，其中 50%是高中生，50%来自人口普查局的工作人员。该数据集的收集目的是希望通过具体算法来实现对手写体数字的识别。在该 MNIST 数据集中，训练集一共包含了 60000 张图像和标签，而测试集则一共包含了 10000 张图像和标签。该数据集自 1998 年起，就被广泛地应用于机器学习和深度学习领域，通常是被用来测试算法的效果。如下图 3 所示即为 MNIST 数据集中的部分手写体数字图片：

图 3

三、实验详细过程

3.1 加载并读取数据集

因为 MNIST 数据集是一个比较经典的机器学习数据集，其已经被 Pytorch 库收录于其数据库中，所以可以直接通过 torchvision.datasets.MNIST()函数直接进行调用，从而省去了较为繁琐的预处理步骤。在本次深度学习实验中，数据加载读取的部分代码如下图 4 所示：

```
# 加载读取MNIST手写体数字数据集
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(), torchvision.transforms.Normalize((0.1307,), (0.3081,))])
train_data = torchvision.datasets.MNIST(root = './Data', train = True, download = True, transform = transform)
test_data = torchvision.datasets.MNIST(root = './Data', train = False, download = True, transform = transform)
train_loader = DataLoader(dataset = train_data, batch_size = batch_size, shuffle = True)
test_loader = DataLoader(dataset = test_data, batch_size = batch_size, shuffle = True)
```

图 4

在这其中，torchvision.transforms.Compose()函数的功能是通过 Compose 把一些对图像处理的方法集中整合起来。此处该行代码完成了数据转换为张量，以及 MNIST 数据集标准化的操作（0.1307 和 0.3081 分别是 MNIST 数据集的均值和标准差，这些参数由 MINIST 数据集提供方给出）。而通过 Pytorch 库所提供的 dataloader 方法，则可以自动实现一个迭代器，每次返回一组 batch_size 个样本和标签。经过这些操作以后，MINIST 数据集就以样本和标签的形式保存在了 train_loader 和 test_loader 中。

3.2 搭建 MLP 网络结构

定义类 MLP 表示网络模型，在网络结构方面，我采用了两个隐藏层和一个输出层，提前设定好每层的神经元数量为 512，随即丢弃概率 dropout 为 0.2。前两层的激活函数我选择使用 ReLU()，而在输出层我便加了一个 softmax 进行归一化处理。网络结构的具体代码如下图 5 所示：

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()#  调用父类的构造函数
        #  神经元数量
        layer_1 = 512
        layer_2 = 512
        self.fc_1 = nn.Linear(28 * 28, layer_1)
        self.fc_2 = nn.Linear(layer_1, layer_2)
        self.fc_3 = nn.Linear(layer_2, 10)
        self.dropout = nn.Dropout(0.2)#  随机丢弃概率

    def forward(self, out):
        out = out.view(-1, 28 * 28)

        out = Function.relu(self.fc_1(out))
        out = self.dropout(out)

        out = Function.relu(self.fc_2(out))
        out = self.dropout(out)

        out = Function.log_softmax(self.fc_3(out), dim = 1)
        return out
```

图 5

3.3 定义损失函数

本次深度学习实验中的损失函数我选择使用交叉熵损失函数。通过如下图 6 所示的代码定义损失函数：

```
loss_function = torch.nn.CrossEntropyLoss().cuda()
```

图 6

通过.cuda()方法使损失函数的计算迁移到了 GPU 上，便能够一定程度上来提升训练的速度。

3.4 定义优化器

本次深度学习实验中我选择采用 Adam 优化器，其学习率不妨设置为 0.001。通过如下图 7 所示的代码定义优化器：

```
optimizer = torch.optim.Adam(params = model.parameters(), lr = 0.001)#  学习率设置为0.001
```

图 7

3.5 MLP 模型训练过程

一轮 epoch 模型迭代训练的具体运行过程大体上可以分为如下所示的几个步骤：

1. 将具体数据输入到模型中并计算输出。可以选择通过.to(device)的方法将计算过程迁移到 GPU 上，从而进一步使计算过程能不断加速；

2. 根据交叉熵损失函数来计算损失；

3. 通过.backward()完成反向传播预测；

4. 注意更新参数：optimizer.step()；

5. 每完成一遍训练后，再用测试集测试模型效果，得到准确率。

在整个训练过程中，设置 batch_size 为 128，总共进行 50 轮 epoch 迭代训练，在所有训练完成后，保存模型参数为 BEST_MLP.ckpt 文件。

四、实验结果与分析

在我这台计算机的 GPU 为 Nvidia GeForce RTX 3060 的环境下，训练 1 轮 epoch 大致需要 10s。而训练完 50 轮 epoch 后，得到的实验命令行运行结果如下图 8 所示：

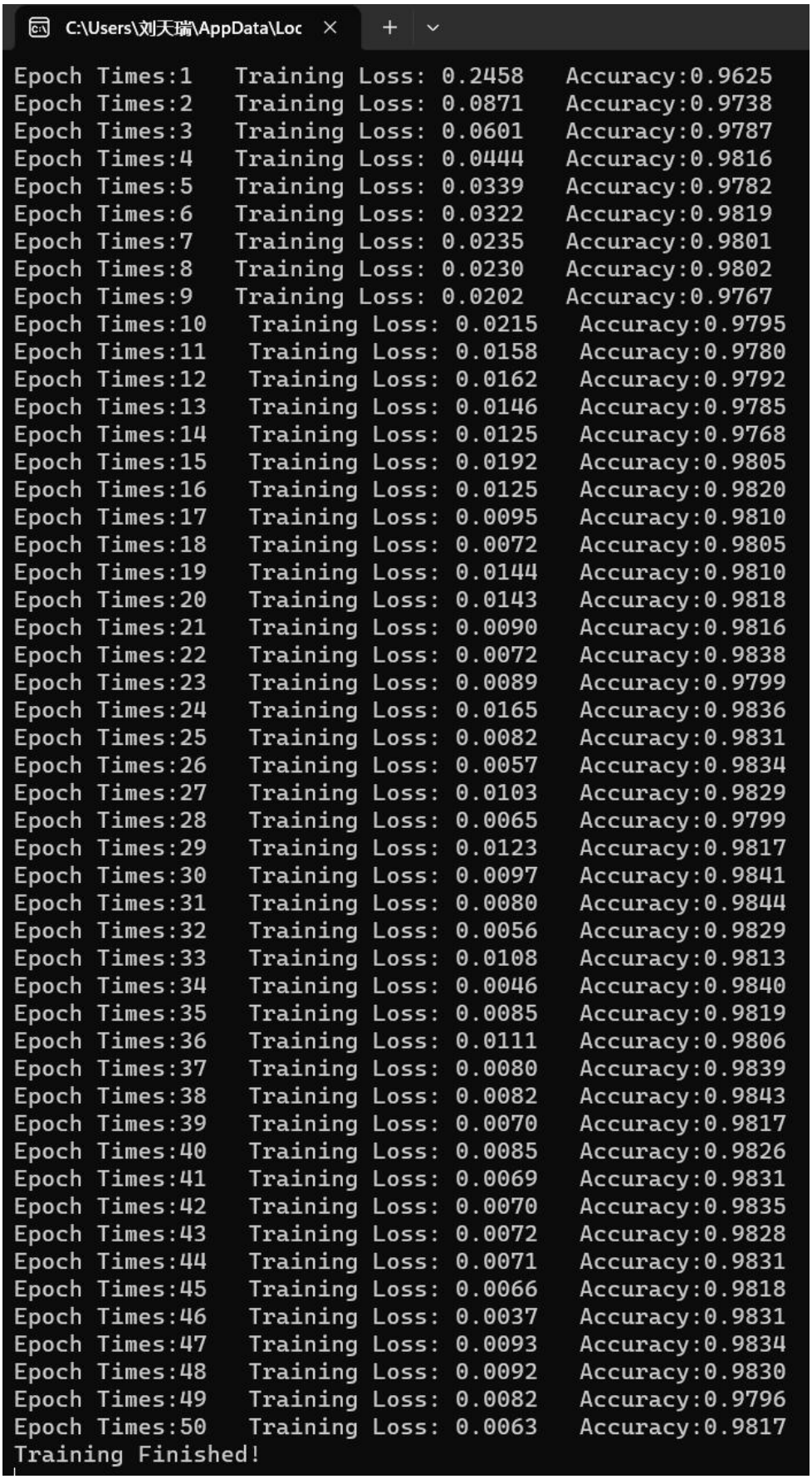
C:\Users\刘天瑞\AppData\Loc
Epoch Times:1 Training Loss: 0.2458 Accuracy:0.9625
Epoch Times:2 Training Loss: 0.0871 Accuracy:0.9738
Epoch Times:3 Training Loss: 0.0601 Accuracy:0.9787
Epoch Times:4 Training Loss: 0.0444 Accuracy:0.9816
Epoch Times:5 Training Loss: 0.0339 Accuracy:0.9782
Epoch Times:6 Training Loss: 0.0322 Accuracy:0.9819
Epoch Times:7 Training Loss: 0.0235 Accuracy:0.9801
Epoch Times:8 Training Loss: 0.0230 Accuracy:0.9802
Epoch Times:9 Training Loss: 0.0202 Accuracy:0.9767
Epoch Times:10 Training Loss: 0.0215 Accuracy:0.9795
Epoch Times:11 Training Loss: 0.0158 Accuracy:0.9780
Epoch Times:12 Training Loss: 0.0162 Accuracy:0.9792
Epoch Times:13 Training Loss: 0.0146 Accuracy:0.9785
Epoch Times:14 Training Loss: 0.0125 Accuracy:0.9768
Epoch Times:15 Training Loss: 0.0192 Accuracy:0.9805
Epoch Times:16 Training Loss: 0.0125 Accuracy:0.9820
Epoch Times:17 Training Loss: 0.0095 Accuracy:0.9810
Epoch Times:18 Training Loss: 0.0072 Accuracy:0.9805
Epoch Times:19 Training Loss: 0.0144 Accuracy:0.9810
Epoch Times:20 Training Loss: 0.0143 Accuracy:0.9818
Epoch Times:21 Training Loss: 0.0090 Accuracy:0.9816
Epoch Times:22 Training Loss: 0.0072 Accuracy:0.9838
Epoch Times:23 Training Loss: 0.0089 Accuracy:0.9799
Epoch Times:24 Training Loss: 0.0165 Accuracy:0.9836
Epoch Times:25 Training Loss: 0.0082 Accuracy:0.9831
Epoch Times:26 Training Loss: 0.0057 Accuracy:0.9834
Epoch Times:27 Training Loss: 0.0103 Accuracy:0.9829
Epoch Times:28 Training Loss: 0.0065 Accuracy:0.9799
Epoch Times:29 Training Loss: 0.0123 Accuracy:0.9817
Epoch Times:30 Training Loss: 0.0097 Accuracy:0.9841
Epoch Times:31 Training Loss: 0.0080 Accuracy:0.9844
Epoch Times:32 Training Loss: 0.0056 Accuracy:0.9829
Epoch Times:33 Training Loss: 0.0108 Accuracy:0.9813
Epoch Times:34 Training Loss: 0.0046 Accuracy:0.9840
Epoch Times:35 Training Loss: 0.0085 Accuracy:0.9819
Epoch Times:36 Training Loss: 0.0111 Accuracy:0.9806
Epoch Times:37 Training Loss: 0.0080 Accuracy:0.9839
Epoch Times:38 Training Loss: 0.0082 Accuracy:0.9843
Epoch Times:39 Training Loss: 0.0070 Accuracy:0.9817
Epoch Times:40 Training Loss: 0.0085 Accuracy:0.9826
Epoch Times:41 Training Loss: 0.0069 Accuracy:0.9831
Epoch Times:42 Training Loss: 0.0070 Accuracy:0.9835
Epoch Times:43 Training Loss: 0.0072 Accuracy:0.9828
Epoch Times:44 Training Loss: 0.0071 Accuracy:0.9831
Epoch Times:45 Training Loss: 0.0066 Accuracy:0.9818
Epoch Times:46 Training Loss: 0.0037 Accuracy:0.9831
Epoch Times:47 Training Loss: 0.0093 Accuracy:0.9834
Epoch Times:48 Training Loss: 0.0092 Accuracy:0.9830
Epoch Times:49 Training Loss: 0.0082 Accuracy:0.9796
Epoch Times:50 Training Loss: 0.0063 Accuracy:0.9817
Training Finished!

图 8

损失值和准确度曲线图如下图 9 所示：

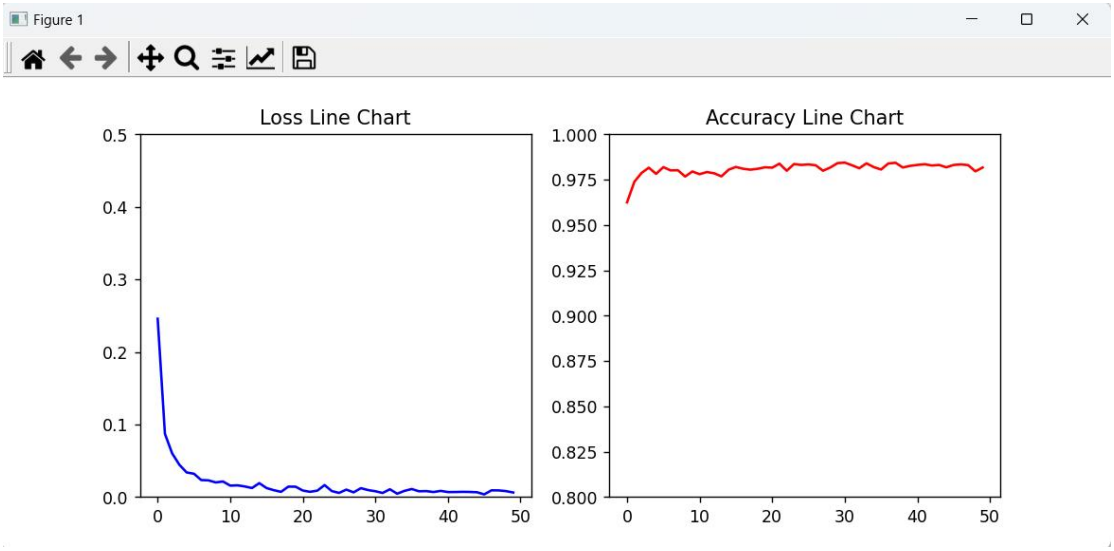

图 9

可以明显看出，在经过了仅仅几轮的 epoch 后，构建好的 MLP 模型在 MNIST 数据集上便已经有了比较优秀的识别效果，最终的准确率达到了 98%左右。而损失也在一定数量的 epoch 迭代轮次后保持在了 0.01 以下，并没有出现过拟合的现象。

最后我说明一下模型和数据文档归类。在实验的当前文件夹中，MLP.py 是程序文件。./Data 中是从网站上下载下来的 MNIST 手写体数字数据集，而./Model 中则保存着已经训练好的最佳模型。在 MLP.py 的 main 函数中如果只保留着 train_model()函数，则可以进行对 MLP 模型的训练过程，最后训练完成后的最佳模型就会保存在./Model/BEST_MLP.ckpt 路径文件中；而如果只保留 load_model()函数，则会载入训练好的模型，并且在测试集上验证其效果，其具体准确度效果如下图 10 所示：

图 10

源代码以及注释见附件。