第 2 次实验报告

2022111663　刘博飞

2024 年 10 月 8 日

1　环境介绍

1. 操作系统:Linux 系统

图 1: 操作系统信息

2. GPU 型号:

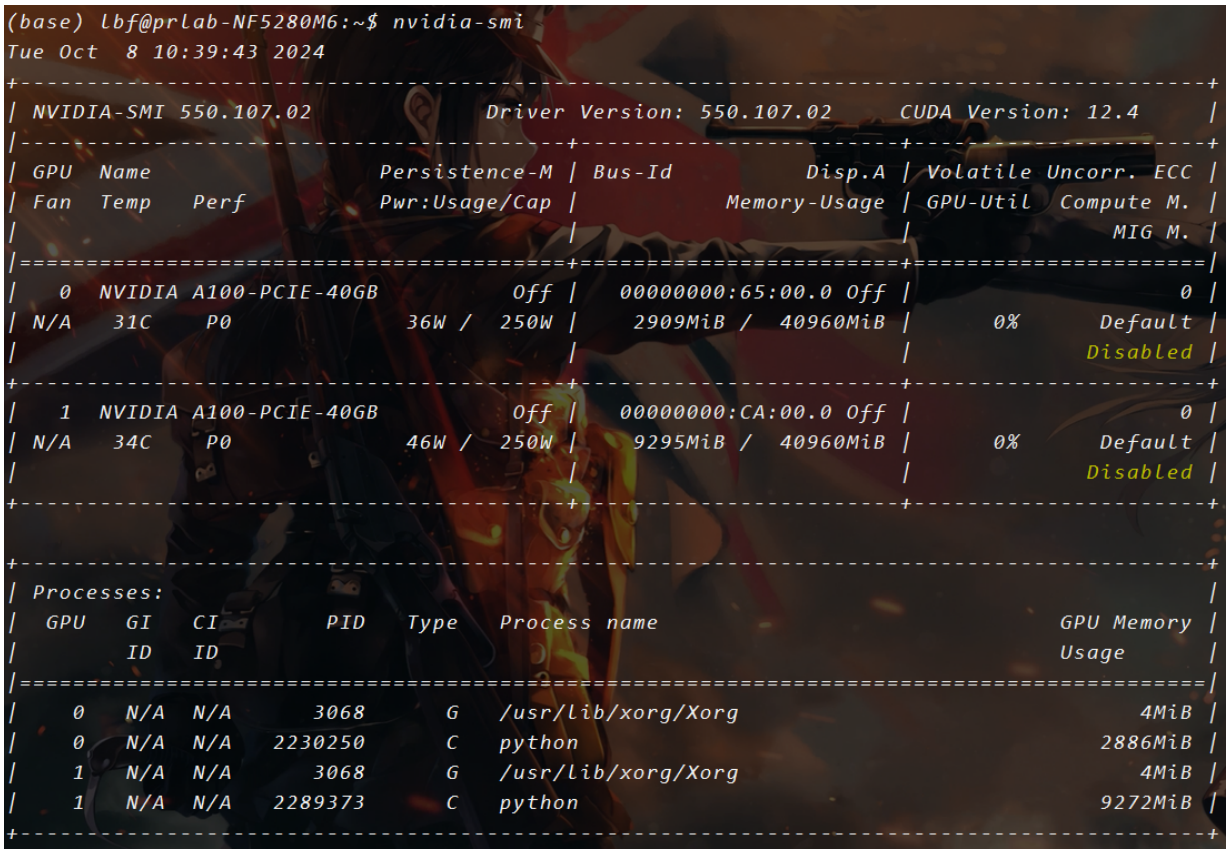


图 2: GPU 型号

3. torch 版本:

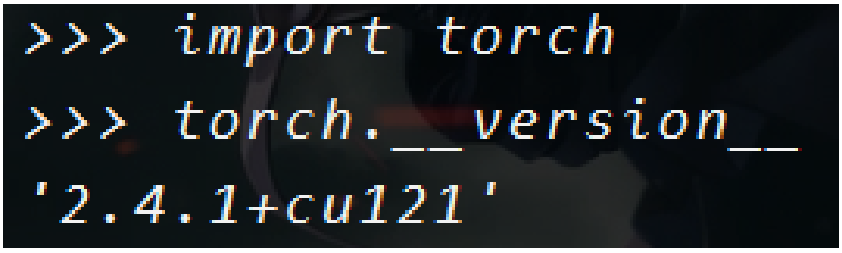


图 3: torch 版本

2 任务 1: 使用 pytorch 框架训练一个 MLP（多层感知机）

2.1 数据集下载与配置

从官网上下载 MNIST 数据集, 得到四个 ubyte 文件, 分别存储了训练集和测试集的 figure 和 label，但是发现不能直接从其中提取数据，因此根据原论文说明配置了数据集文件夹，结构如下：

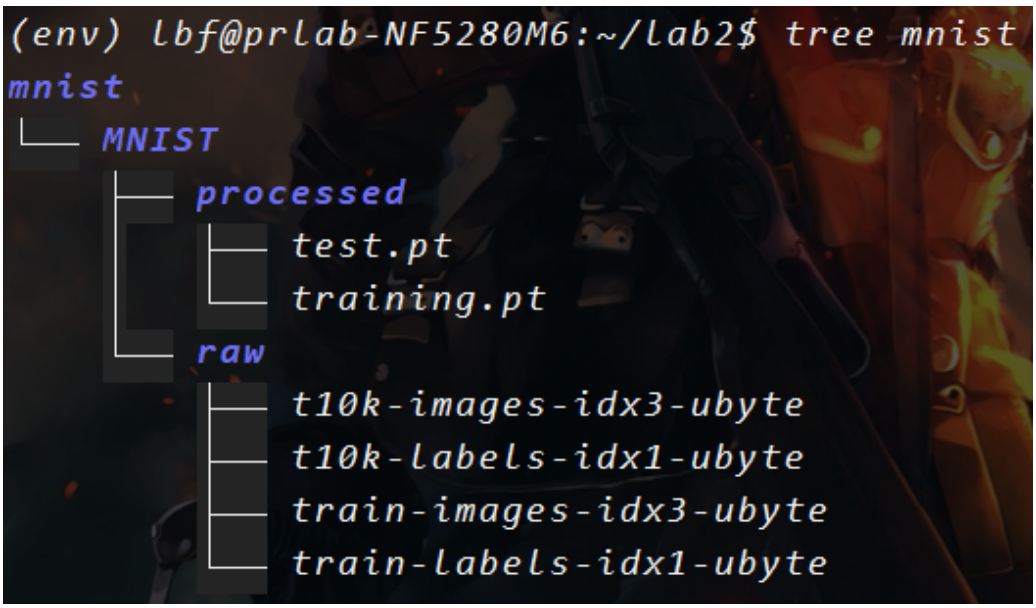


图 4: 数据集文件夹结构

2.2 数据集加载

使用 dataset 和 Dataloader 定义并加载数据集，发现 pytorch 中已经集成了 MNIST 数据集的个性化加载方法，其路径为: torchvision.datasets.MNIST(root,train,transform,download).

然后使用 torch.utils.data.Dataloader(dataset,batch_size,shuffle) 加载 MNIST 数据集, 在本实验中，Dataloader 有着如下的结构:[[(batch_size,28,28),(batch_size)]*batch_num], 在一个 batch 中，前者是图像的灰度矩阵，后者是每张图的标签。

2.3 网络框架搭建

2.3.1 单一神经层构造

机器学习的网络一般采用继承 torch.nn.Module 的方式构建，本实验中由于仅仅采用了线性神经网络，因此每个神经元只需要维护一个权重矩阵和一个偏置矩阵即可，二者的维数由输入输出的个数决定。之后基于权重矩阵和偏置矩阵构造前向传播函数即可。

2.3.2 多层神经网络模型

与单层神经网络相同，整体的多层神经网络也继承自 Module 类，不同的是，多层神经网络需要维护的是多个线性层方法以及 dropout、relu、softmax 等性能优化组件。

之后基于 self 函数的内容构造前向传播函数，为了方便计算，将灰度矩阵展平为如下结构：(batch_size,28x28). 在经过每层神经网络后，通过 dropout 函数、relu 函数或 softmax 函数处理优化。最终得到了期望概率矩阵，在本实验中的结构如下：(batch_size,10)

2.4 损失函数和优化器初始化

本实验中选择了 nn.NLLLoss 和 torch.optim.Adam 作为损失函数和优化器。

2.5 基于训练集训练 MLP

根据上述神经网络结构的构建，我们可以很容易的对 MLP 进行性训练，前向传播、反向传播以及优化的语句如下：

```
    # 前向传播
    outputs = model(images)
    loss = criterion(outputs, labels)

    # 反向传播和优化
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

3 任务 2: 使用自己构建的 MLP 来对 minist 手写数据集进行分类任务

使用测试集对 MLP 进行测试的思路就很简单了，可以陈述为：将测试集数据投入已经训练好的模型，得到训练结果并与标签对比，计算准确率。

4 任务 3: 通过 tensorboard 可视化网络的训练过程（loss 函数的变化）

4.1 tensorboard 的使用

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter('runs/hs1_128_hs2_128_lr_0.001_drop_0.5')
# 日志目录
writer.add_scalar('Loss/train', loss.item(), step)
```

4.2 实验结果分析

在本实验中影响模型性能的因素有很多，我选择了以下几个因素作为变量来探究较优秀的模型结构与参数：隐层神经元个数、隐层层数、学习率、dropout 比例。我使用 tensorboard 绘制了训练集的损失以判断模型的训练程度、绘制了验证集损失和准确率以监督模型的训练过程 (每训练 100 个 batch 将测试集作为验证集来验证模型的训练效果)。

4.2.1 隐层神经元个数

我分别在一层、两层、三层隐层层数的基础上改变了不同的隐层神经元个数进行试验。

a. 一层隐层的神经网络

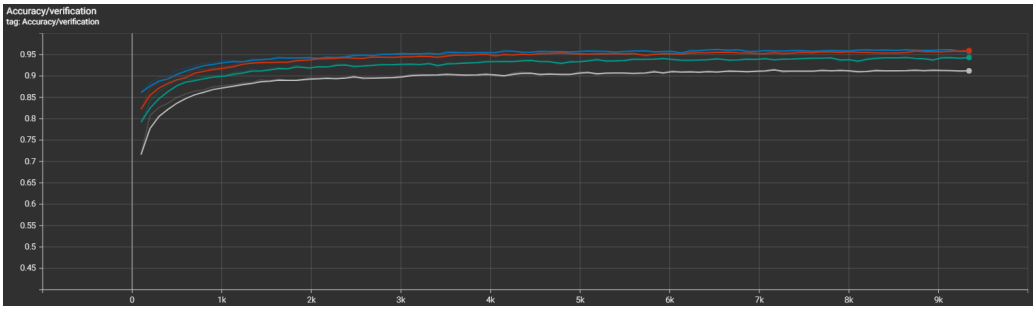
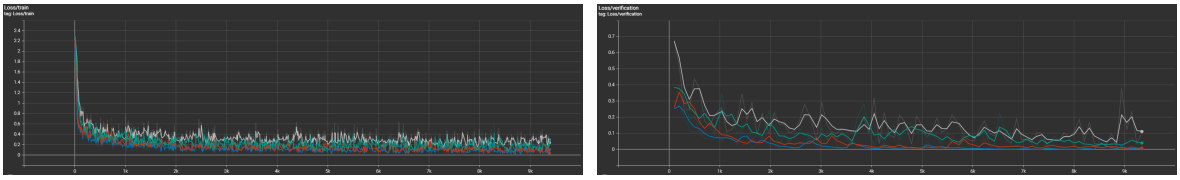


图 5: 验证集准确率



(a) 训练集损失　　(b) 验证集损失

图 6: 损失曲线

根据实验数据可以看出，红色（256 个神经元）和蓝色（512 个神经元）曲线的训练效果较好，并且两者相差不多，因此一层隐层的条件下可以采用 256 个神经元。

b. 两层隐层的神经网络

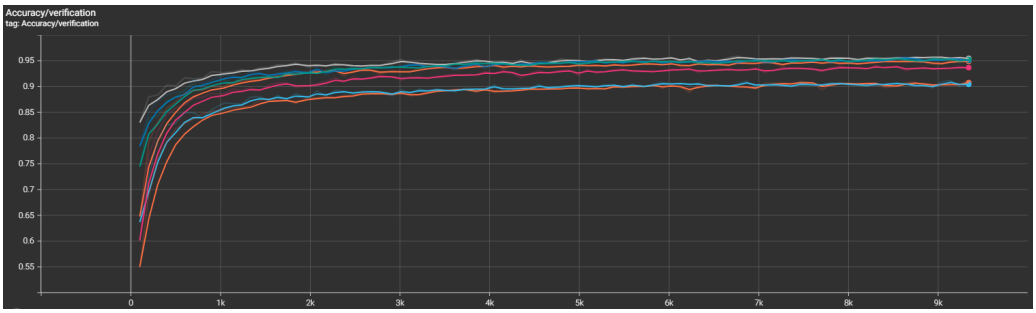
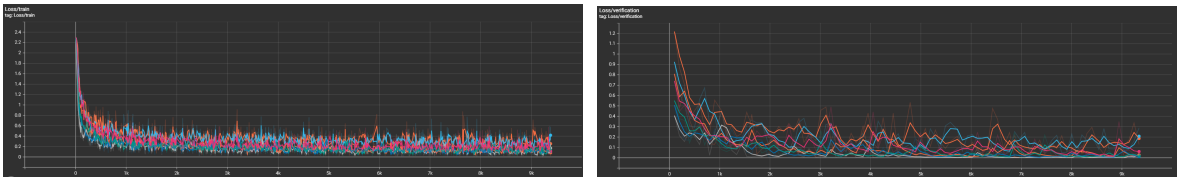


图 7: 验证集准确率



(a) 训练集损失　　(b) 验证集损失

图 8: 损失曲线

根据实验数据可以看出，灰色（hs1=512,hs2=512）、蓝色（hs1=256,hs2=256）、绿色（hs1=256,hs2=128）、橙色（hs1=256,hs2=64）曲线的训练效果较好，并且相差不多，因此两层隐层的条件下可以采用 hs1=256,hs2=128 个神经元。

此外还可以从实验结果中看出隐层神经元数目递减的效果多数情况下表现良好，因此在三层隐层的情况下的设计时可以借鉴这一规律。

c. 三层隐层的神经网络

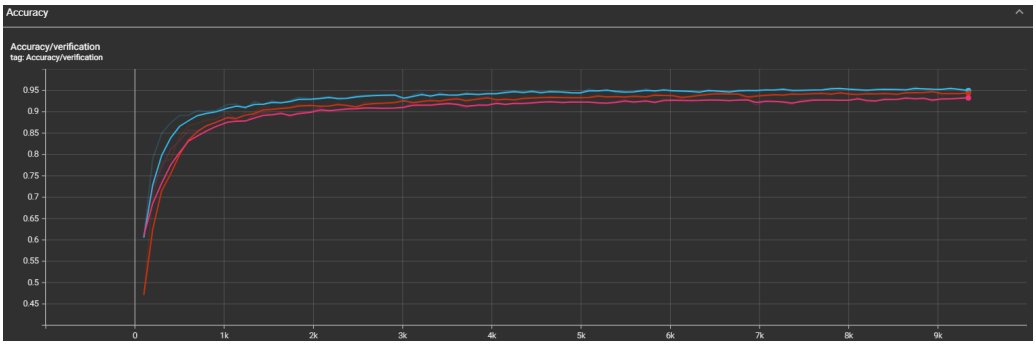
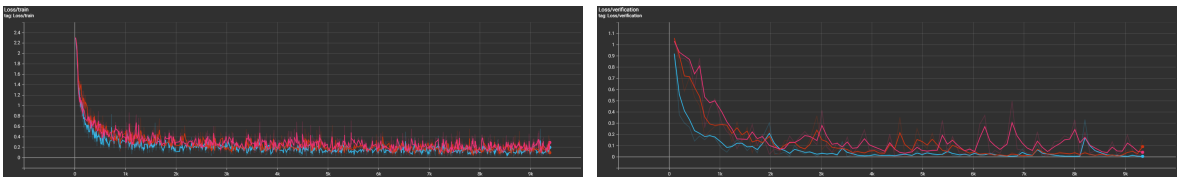


图 9: 验证集准确率



(a) 训练集损失　　(b) 验证集损失

图 10: 损失曲线

根据实验数据可以看出，蓝色（hs1=512,hs2=256,hs3=128）曲线效果明显好于另外两个。因此三层隐层的情况可以选择 hs1=512,hs2=256,hs3=128 个神经元。

4.2.2 隐层层数

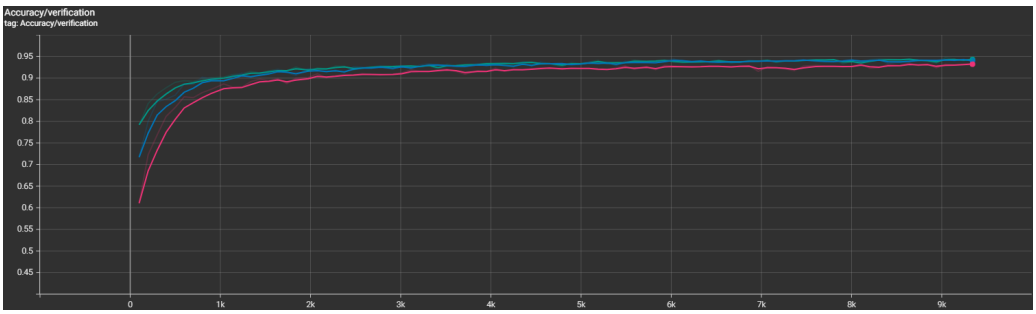
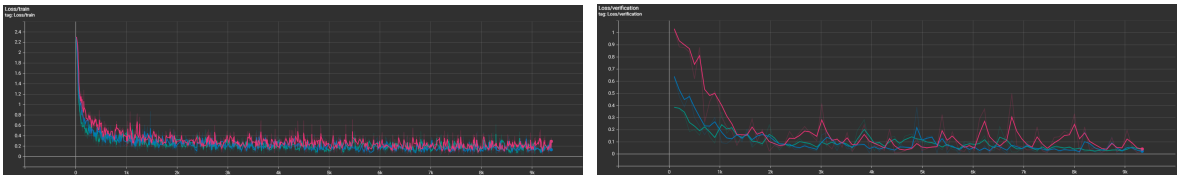


图 11: 验证集准确率



(a) 训练集损失　　(b) 验证集损失

图 12: 损失曲线

根据实验数据可以看出，绿色（1 层）和蓝色（2 层）曲线的训练效果较好，并且两者相差不多，因此建议选择的隐层层数不超过两层。

4.2.3 学习率

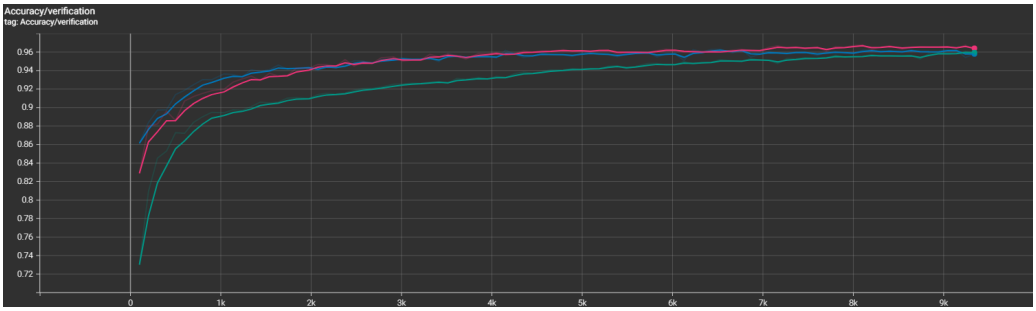
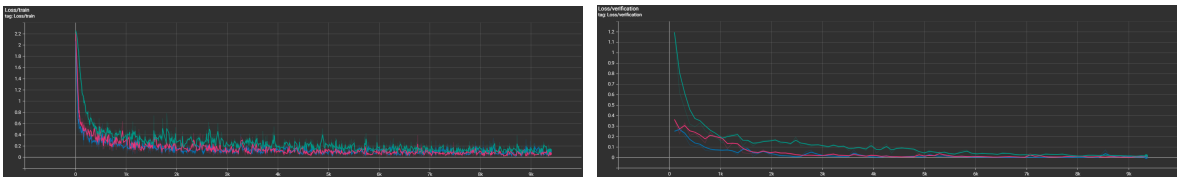


图 13: 验证集准确率



(a) 训练集损失　　(b) 验证集损失

图 14: 损失曲线

根据实验数据可以看出，粉红色（lr=0.0005）曲线的训练效果较好，因此建议选择的学习率为 0.0005。

4.2.4 dropout 比例

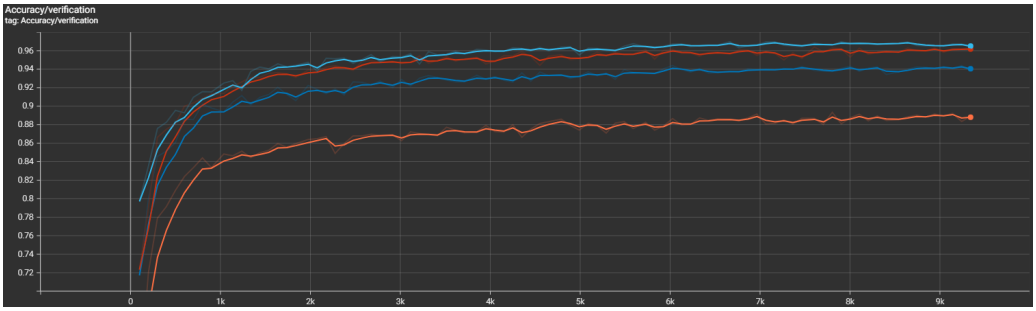
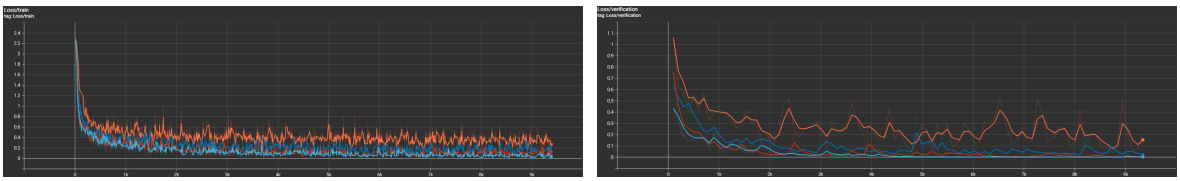


图 15: 验证集准确率



(a) 训练集损失　　(b) 验证集损失

图 16: 损失曲线

根据实验数据可以看出，dropout 的比例越小，模型评估的准确率越高，但是考虑到 dropout 过小可能导致过拟合，因此采用 dropout 的比例为 0.1。

4.2.5 最优模型结构与参数

实验全部的准确率如下图所示，根据以上的描述与合理推测可以得到最优模型的结构与参数为:hs1=256,lr=0.0005,dropout=0.1。

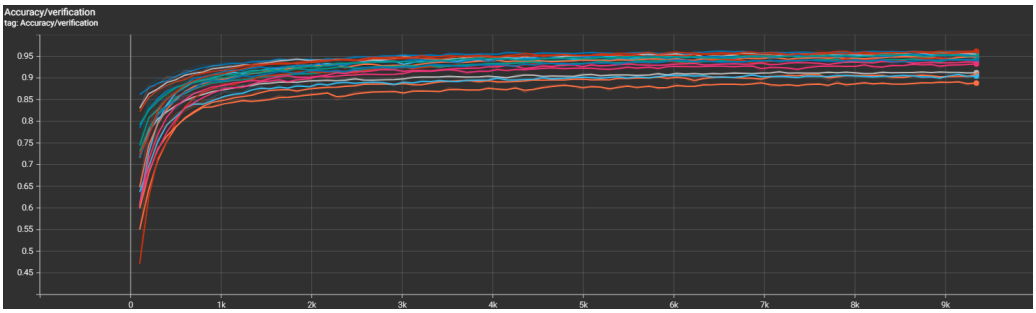


图 17: 验证集准确率

绘制出最优模型的准确率图像如下所示，可以看到明显高于其他结构，因此符合之前的猜测，最终准确率为 97.44%。

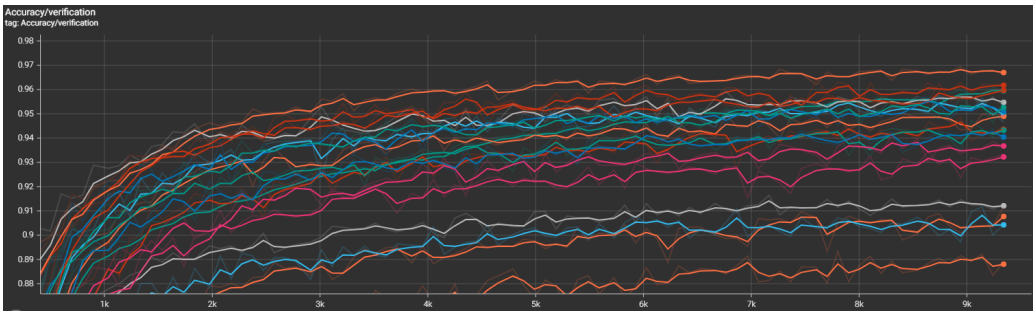


图 18: 验证集准确率

A 自定义线性层

```
class MyLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super(MyLinear, self).__init__()
        self.weights = nn.Parameter(torch.randn(out_features,
            in_features) * 0.01)  # (out,in)
        self.bias = nn.Parameter(torch.zeros(out_features))  # (out)

    def forward(self, x):
        return x @ self.weights.T + self.bias  # 矩阵乘法加偏置
```

B 自定义 Dropout 层

```
class MyDropout(nn.Module):
    def __init__(self, p=0.5):
        super(MyDropout, self).__init__()
        self.p = p

    def forward(self, x):
        if self.training:
            mask = (torch.rand(x.size(0), x.size(1),
                device=x.device) > self.p).float()
            return x * mask / (1 - self.p)
        return x
```

C 定义 MLP 模型

```
    class MLP(nn.Module):
        def __init__(self):
            super(MLP, self).__init__()
            self.fc1 = MyLinear(input_size, hidden_size1)
            self.dropout1 = MyDropout(0.5)
            self.fc2 = MyLinear(hidden_size1, hidden_size2)
            self.dropout2 = MyDropout(0.5)
            self.fc3 = MyLinear(hidden_size2, num_classes)
            self.relu = nn.ReLU()
            self.softmax = nn.LogSoftmax(dim=1)

        def forward(self, x):
            x = x.view(-1, input_size)
            # 将28x28图像展平,即(batch_size,28x28)
            x = self.relu(self.fc1(x))
            x = self.dropout1(x)
            x = self.relu(self.fc2(x))
            x = self.dropout2(x)
            x = self.softmax(self.fc3(x))
            return x  # (batch_size,10)
```