ML:Homework3——FNN_mnist_OCR

人工智能 91 卢佳源 2191121196

一、实验目的：

1、利用前馈全连接网络实现手写体识别；

2、利用 MNIST database 的训练和测试数据进行实验；

3、计算手写体识别的正确率，记录训练和测试过程；

4、网络结构、编程语言不限。

二、实验环境：

OS：Linux 操作系统的 ubuntu20.04 版本

IDE：VScode,anaconda3(‘base’:conda)

Terminal：gnome-terminal -t $TITLE -x

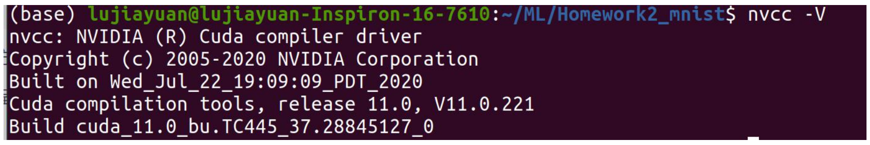
Language：python(3.8.8 64-bit)

PATH：/home/lujiayuan/ML/Homework2_mnist/mnist_basic.py

DataPath:/home/lujiayuan/ML/Homework2_mnist/data/MNIST/processed & raw

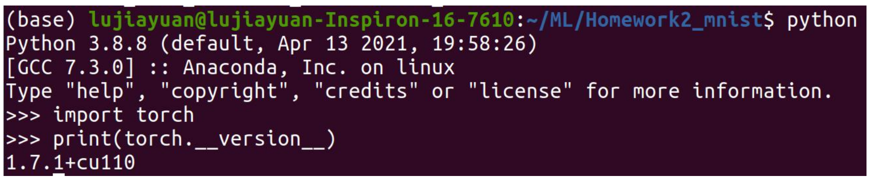
（torch 自带 MNIST 数据集并在调用时进行了预处理）

CUDA 版本：



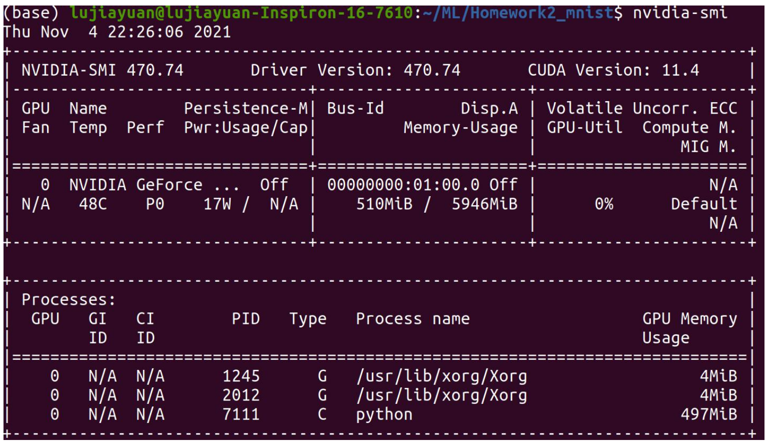
```
(base) lujiayuan@lujiayuan-Inspiron-16-7610:~/ML/Homework2_mnist$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed_Jul_22_19:09:09_PDT_2020
Cuda compilation tools, release 11.0, V11.0.221
Build cuda_11.0_bu.TC445_37.28845127_0
```

pytorch 版本：



```
(base) lujiayuan@lujiayuan-Inspiron-16-7610:~/ML/Homework2_mnist$ python
Python 3.8.8 (default, Apr 13 2021, 19:58:26)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> print(torch.__version__)
1.7.1+cu110
```

显卡信息：Name: GA106M [GeForce RTX 3060 Mobile / Max-Q]



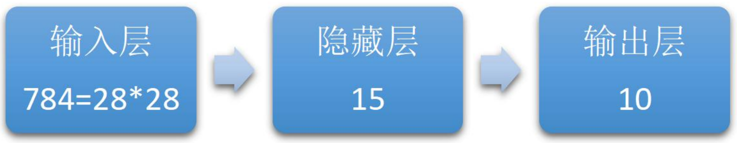
```
(base) lujiayuan@lujiayuan-Inspiron-16-7610:~/ML/Homework2_mnist$ nvidia-smi
Thu Nov  4 22:26:06 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.74       Driver Version: 470.74       CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   48C    P0    17W /  N/A |    510MiB /  5946MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1245      G   /usr/lib/xorg/Xorg                  4MiB |
|    0   N/A  N/A      2012      G   /usr/lib/xorg/Xorg                  4MiB |
|    0   N/A  N/A      7111      C   python                            497MiB |
+-----------------------------------------------------------------------------+
```

三、实验方法：

1、实验思路：

1）定义 FNN 网络（class Net）：

在 FNN 幻灯片的例子中，实现手写数字体识别时，用到了三层神经网络（输入层、隐藏层、输出层）：



我仿照上图进行网络设计，首先 in_hid 代表由输入层到隐藏层为 784 维到 15 维的张量线性运算，隐藏层使用 Leak ReLU 激活函数，最后进行隐藏层到输出层 10 个神经元的线性映射，完成前馈过程。

2）训练数据：

利用梯度下降法（交叉熵）计算训练数据的损失函数，并将输出概率最大值的数据当作预测值，然后计算训练正确率，将准确率函数和损失函数曲线画入 tensorboard。

3）测试数据：

利用上面训练好的模型进行对测试数据的测试，并和 2）相同方法计算损失函数和准确率。

4）输出运行过程中的 CPU 性能：

利用 pytorch 的 profiler 性能分析工具计算 CPU 端 Op 执行时间（即 CPU 负责 Op 的调度时间），有利于以后进行模型优化时发现性能瓶颈。

5）在 main 函数中定义各种训练参数（包括 batch_size，epoch，设备，学习率等），并选择合适的损失函数和优化函数，最后保存模型。

6）利用 NNI 进行模型自动超参数调优和手动参数调优。

2、实验代码：

```
from __future__ import print_function
import argparse
import torch
# torch.cuda.set_device(0)
import nni
import logging
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR
criterion = nn.CrossEntropyLoss()
import numpy as np
import torchvision.models as models
from torch.utils.tensorboard import SummaryWriter
from nni.utils import merge_parameter
writer = SummaryWriter('logs/mnist_experiment_1')
logger = logging.getLogger('mnist_AutoML')
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # prelu=nn.PReLU(num_parameters=1)
        # self.dropout1 = nn.Dropout2d(0.25)
        self.in_hid_1= nn.Linear(784, 512)
        self.hid1=nn.LeakyReLU()
        self.in_hid_2= nn.Linear(512, 256)
        self.hid2=nn.LeakyReLU()
        self.in_hid_3= nn.Linear(256, 128)
        self.hid3=nn.LeakyReLU()
        self.hid_out=nn.Linear(128,10)

    def forward(self, data):
        x = data.view(-1, 784)
        output=self.in_hid_1(x)
        # output=self.dropout1(output)
        output=self.hid1(output)
        output=self.in_hid_2(output)
        output=self.hid2(output)
        output=self.in_hid_3(output)
        output=self.hid3(output)
        output=self.hid_out(output)
        output = F.log_softmax(output, dim=1)
        return output


def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    running_loss = 0.0
    correct = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()
        if batch_idx % args['log_interval'] == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            if batch_idx != 0:
                global_step = (epoch - 1) * len(train_loader) + batch_idx
```

```
                    writer.add_scalar('Loss/train', running_loss / (args['batch_size'] * args['log_interval']), global_step)
                    writer.add_scalar('Accuracy/train', 100. * correct / (args['batch_size'] * args['log_interval']), global_step)
                running_loss = 0.0
                correct = 0.0

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))

def profile(model, device, train_loader):
    dataiter = iter(train_loader)
    data, target = dataiter.next()
    data, target = data.to(device), target.to(device)
    with torch.autograd.profiler.profile(use_cuda=False) as prof:
        model(data[0].reshape(1,1,28,28))
    print(prof)

def main():
    torch.backends.cudnn.enabled = False ###
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=14, metavar='N',
                        help='number of epochs to train (default: 14)')
    parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                        help='learning rate (default: 1.0)')
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training
status')

    parser.add_argument('--save-model', action='store_true', default=True,
                        help='For Saving the current Model')
    args = parser.parse_args()
    # use_cuda = not args.no_cuda and torch.cuda.is_available()

    # torch.manual_seed(args.seed)

    # device = torch.device("cuda" if use_cuda else "cpu")
    # kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
    # train_loader = torch.utils.data.DataLoader(
    #     datasets.MNIST('data', train=True, download=True,
    #                    transform=transforms.Compose([
    #                        transforms.ToTensor(),
    #                        transforms.Normalize((0.1307,), (0.3081,))
    #                    ])),
    #     batch_size=args.batch_size, shuffle=True, **kwargs)
    # test_loader = torch.utils.data.DataLoader(
    #     datasets.MNIST('data', train=False, transform=transforms.Compose([
    #                        transforms.ToTensor(),
    #                        transforms.Normalize((0.1307,), (0.3081,))
    #                    ])),
    #     batch_size=args.test_batch_size, shuffle=True, **kwargs)
    # dataiter = iter(train_loader)
    # images, labels = dataiter.next()

    # grid = torchvision.utils.make_grid(images)
    # writer.add_image('images', grid, 0)

    # model = Net().to(device)
    # optimizer = optim.Adadelta(model.parameters(), lr=args.lr)
    # images=images.to(device)
    # writer.add_graph(model, images)
    # scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    # print("Start profiling...")
```

```
        # profile(model, device, train_loader)
        # print("Finished profiling.")
        # for epoch in range(1, args.epochs + 1):
        #           train(args, model, device, train_loader, optimizer, epoch)
        #           test_acc=test(model, device, test_loader)
        #           scheduler.step()
        #           # report intermediate result
        #           nni.report_intermediate_result(test_acc)
        #           logger.debug('test accuracy %g', test_acc)
        #           logger.debug('Pipe send intermediate result done.')

        # # report final result
        # nni.report_final_result(test_acc)



        # if args.save_model:
        #           print("Our model: \n\n", model, '\n')
        #           print("The state dict keys: \n\n", model.state_dict().keys())
        #           torch.save(model.state_dict(), "mnist.pt")
        #           state_dict = torch.load('mnist.pt')
        #           print(state_dict.keys())
        # writer.close()

        return args

def NNI(args):
        # use_cuda = not args.no_cuda and torch.cuda.is_available()
        use_cuda = not args['no_cuda'] and torch.cuda.is_available()
        # torch.manual_seed(args.seed)
        torch.manual_seed(args['seed'])

        device = torch.device("cuda" if use_cuda else "cpu")
        kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
        train_loader = torch.utils.data.DataLoader(
                datasets.MNIST('data', train=True, download=True,
                                transform=transforms.Compose([
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.1307,), (0.3081,))
                                ])),
                batch_size=args['batch_size'], shuffle=True, **kwargs)
        test_loader = torch.utils.data.DataLoader(
                datasets.MNIST('data', train=False, transform=transforms.Compose([
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.1307,), (0.3081,))
```

```
                        ])),
            batch_size=args['test_batch_size'], shuffle=True, **kwargs)
        dataiter = iter(train_loader)
        images, labels = dataiter.next()

        grid = torchvision.utils.make_grid(images)
        writer.add_image('images', grid, 0)

        model = Net().to(device)
        optimizer = optim.Adadelta(model.parameters(), lr=args['lr'])
        images=images.to(device)
        writer.add_graph(model, images)
        scheduler = StepLR(optimizer, step_size=1, gamma=args['gamma'])#等间隔调整学习率
StepLR，  将学习率调整为 lr*gamma
        print("Start profiling...")
        profile(model, device, train_loader)
        print("Finished profiling.")
        for epoch in range(1, args['epochs'] + 1):
            train(args, model, device, train_loader, optimizer, epoch)
            test_acc=test(model, device, test_loader)
            scheduler.step()
            nni.report_intermediate_result(test_acc)
            logger.debug('test accuracy %g', test_acc)
            logger.debug('Pipe send intermediate result done.')
        nni.report_final_result(test_acc)


        if args['save_model']:
            print("Our model: \n\n", model, '\n')
            print("The state dict keys: \n\n", model.state_dict().keys())
            torch.save(model.state_dict(), "mnist.pt")
            state_dict = torch.load('mnist.pt')
            print(state_dict.keys())
        writer.close()


    if __name__ == '__main__':
        tuner_params = nni.get_next_parameter()
        logger.debug(tuner_params)
        params = vars(merge_parameter(main(), tuner_params))
        print(params)
        NNI(params)
```

四、实验结果：

1、实验结果：

三层 FNN（hid_size=15）见 epoch.txt，train_profile.txt，accuracy.csv，loss.csv

三层 FNN（hid_size=225）见 train_profile_225.txt

三层 FNN（hid_size=650）见 train_profile_650.txt

两层隐藏层见 train_profile_2hid.txt

三层隐藏层见 train_profile_3hid.txt，1_acc.csv，3_loss.csv

graph 模型结构图见 graph.png

2、实验结果分析：

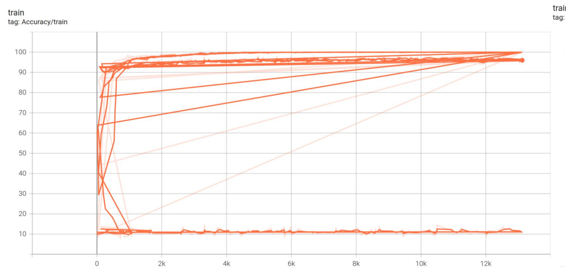
利用 tensorboard 进行可视化模型结构，并计算正确率和损失率。由下图可知，随着训练次数的增加，模型的训练准确率迅速增加，最终趋于稳定，大约在 95%，同时训练的损失函数也在快速下降，最终趋于 0.1 左右。

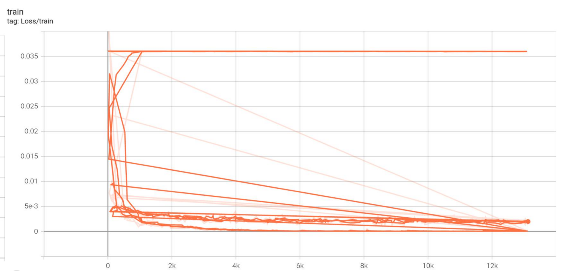
从终端输出结果可以看到测试集在每一次训练后的平均损失率和模型预测准确率，同训练集趋势相近，平均损失率逐渐下降，趋于 0.0002，准确率逐渐上升，趋于 0.95。

由 Graph 图可以将我设计的 FNN 结构可视化。（原图在附件中）

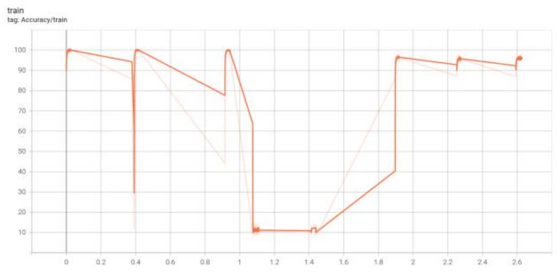
- tensorboard 结果可视化：

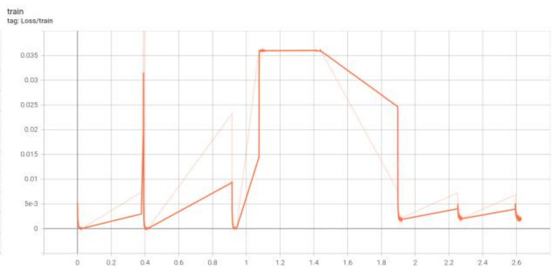
1）STEP：



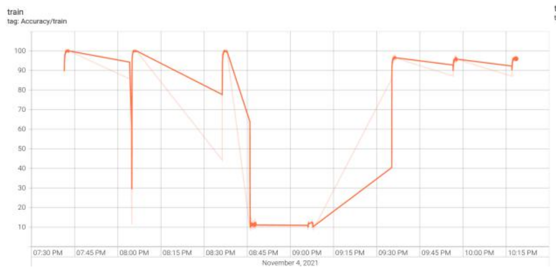


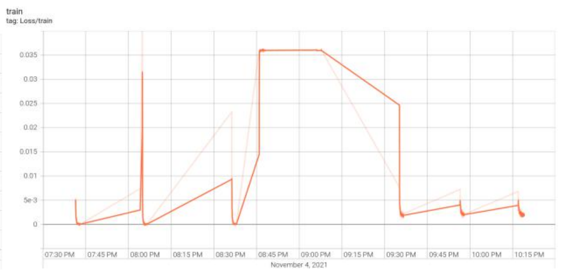
2）RELATIVE：



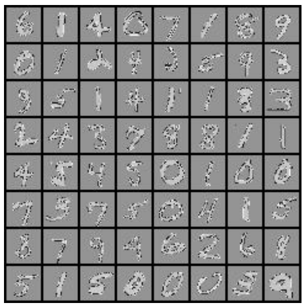


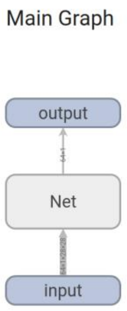
3）WALL：

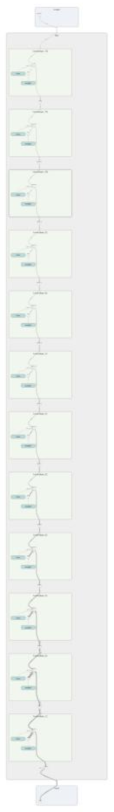




- imageData：　　　　　　　　　　Graphs:







- profile:见 train_profile.txt

五、遇到问题及解决思路：

问题 1：激励函数、损失函数和优化函数的选择会使模型的损失率和正确率有很大的改变，即存在不收敛的情况，比如我最初选择了 nul_loss()这个损失函数，输出的准确率仅有 11%。

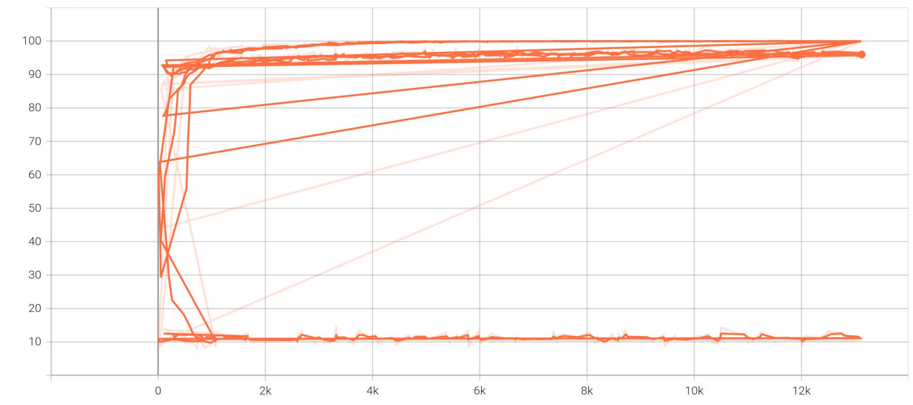
解决方法：最简单的解决方法就是换用不同的损失函数来进行训练，找到最适合该模型的损失函数。但是更应该从本质了解这三种函数的数学性质，比如其线性性质、非饱和性质、怎样缓解梯度消失等，选择既适合该模型，同时三者搭配合理的三个函数。

问题 2：隐藏层神经元的数目对识别性能的影响？

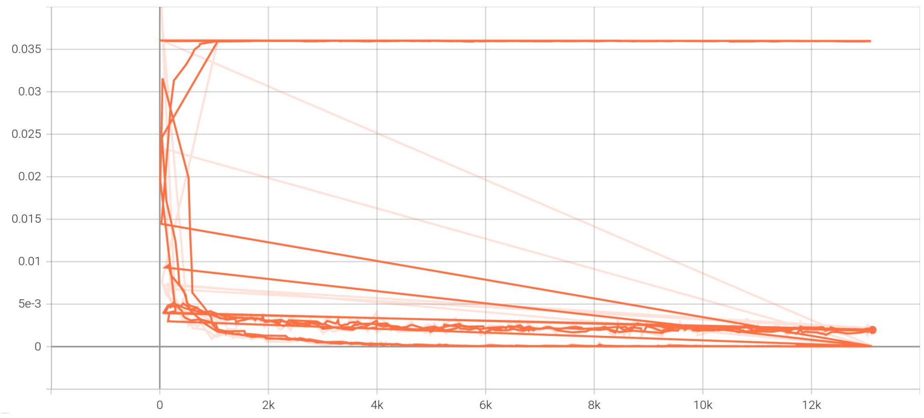
解决方法：尝试用不同的隐藏层大小来进行测试。同样，这也是一个依模型而定的参数，在我的试验看来，当隐藏层神经元个数选择为 15 个时，准确率为 95%， 而当选为 225 时，准确率为 98%（9827/10000），当选为 650 时，准确率为 98%（9836/10000）。

hid_size=225:

train
tag: Accuracy/train

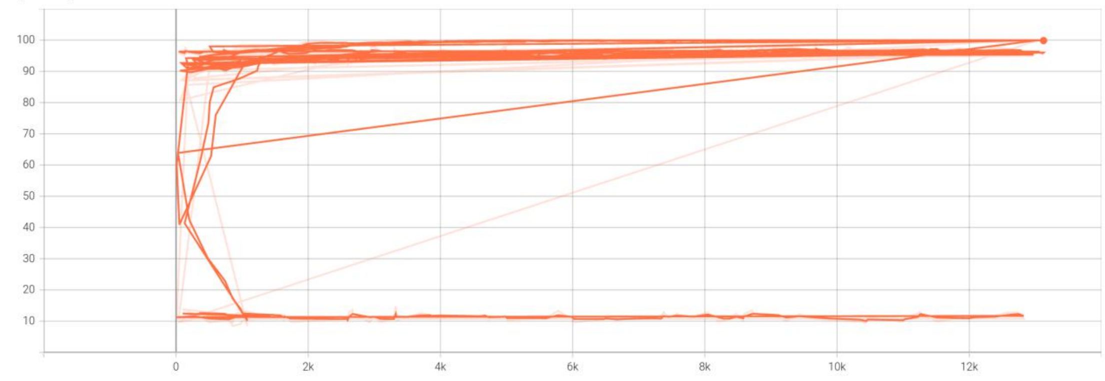


train
tag: Loss/train

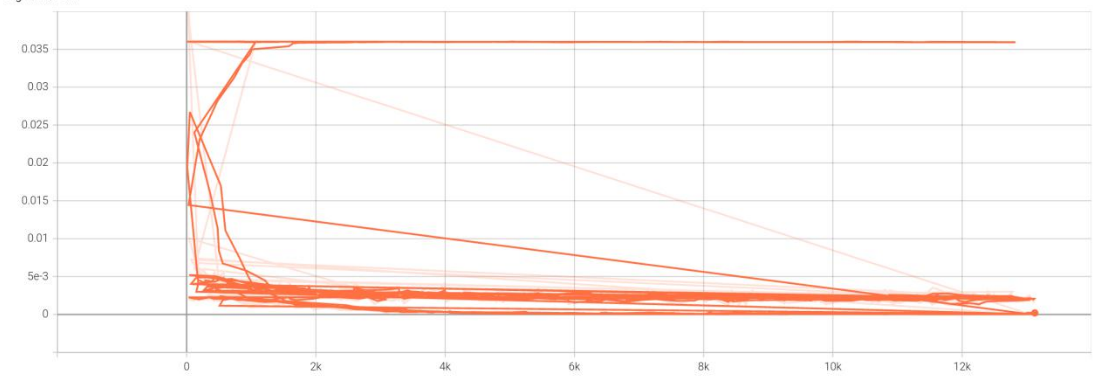


hid_size=225:

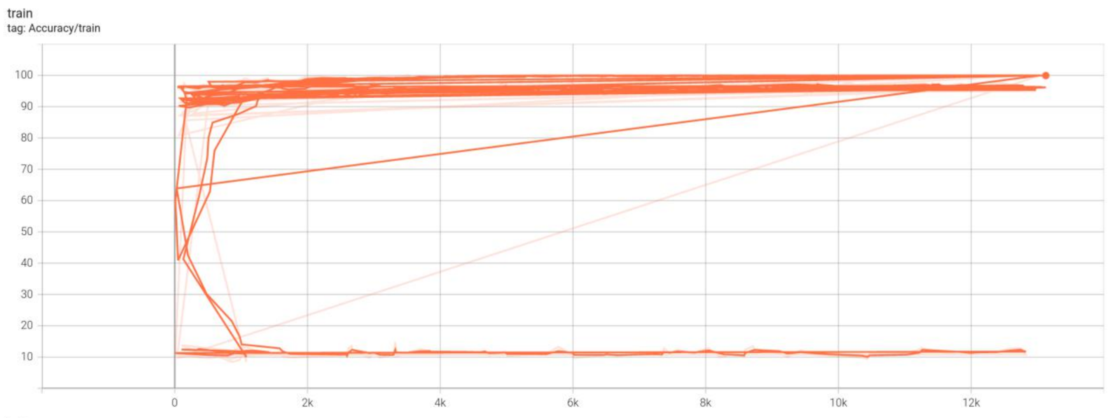
train
tag: Accuracy/train

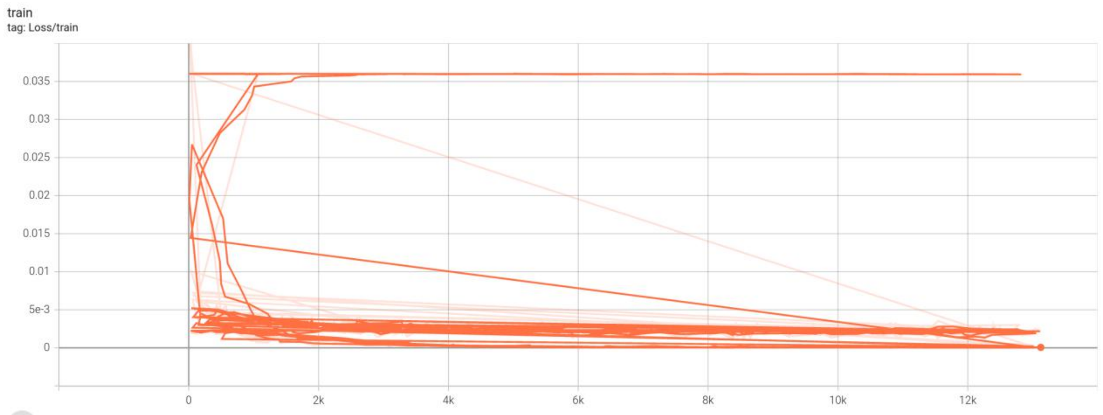


train
tag: Loss/train



hid_size=650:

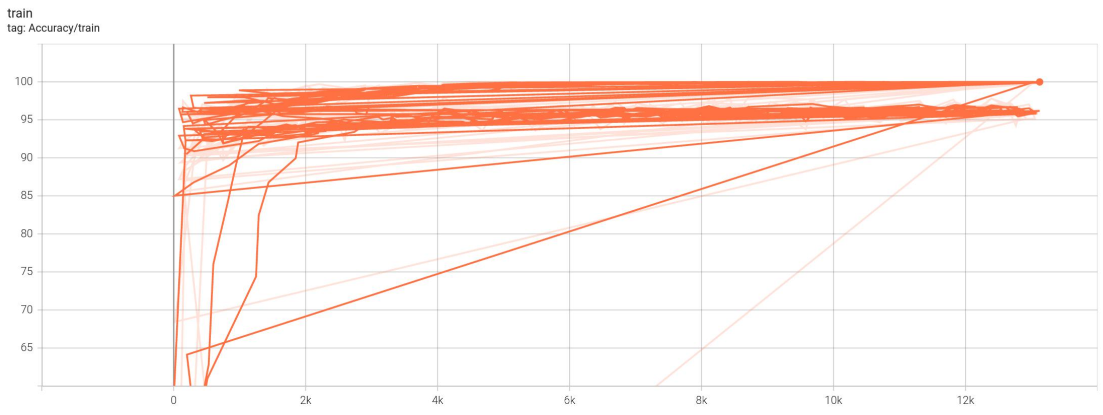




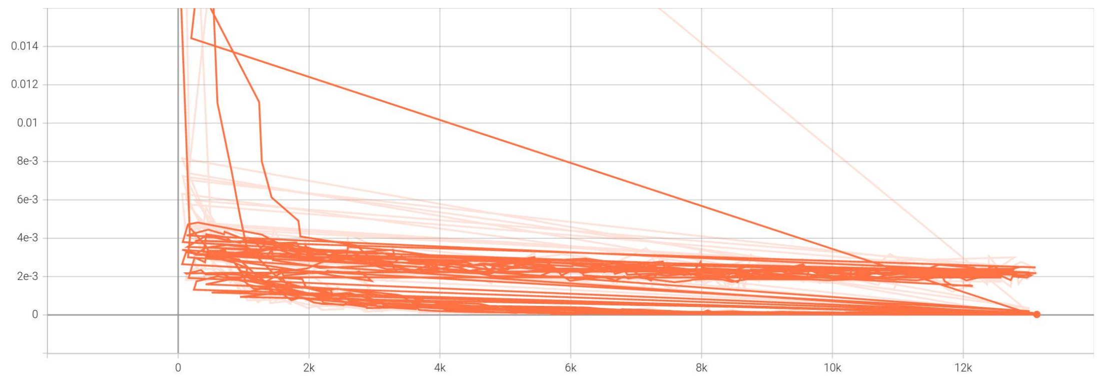
问题 3：增加隐藏层层数对模型准确率的影响？（hid_size=256）

解决方法：一定程度上地增加隐藏层层数可以一定程度上提升准确率，但是当层数过多时，会出现过拟合情况，造成准确率不收敛的问题。因此隐藏层层数的选取要适合当前模型。

两层隐藏层：Test set: Average loss: 0.0001, Accuracy: 9842/10000 (98%)

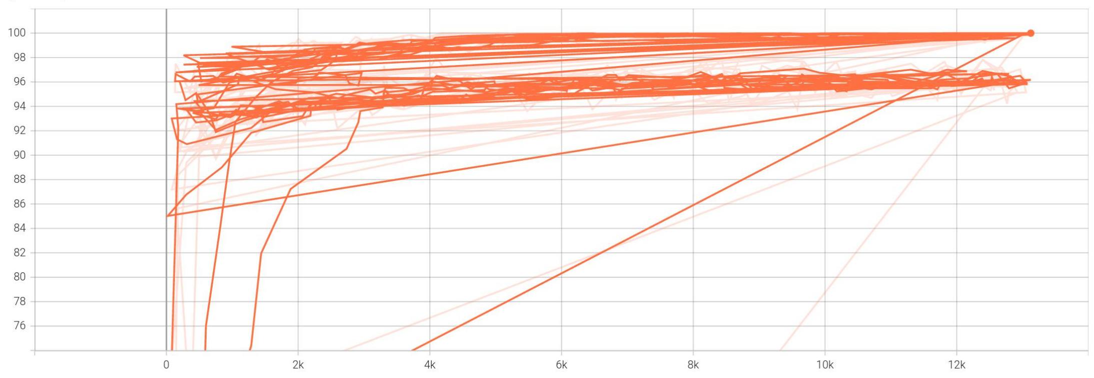


train
tag: Loss/train

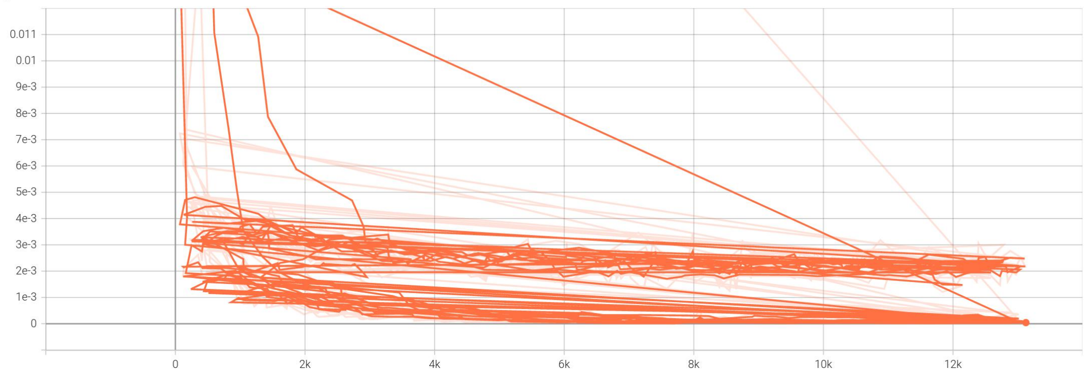


三层隐藏层：Test set: Average loss: 0.0001, Accuracy: 9850/10000 (98%)

train
tag: Accuracy/train



train
tag: Loss/train


0.011
0.01
9e-3
8e-3
7e-3
6e-3
5e-3
4e-3
3e-3
2e-3
1e-3
0
0
2k
4k
6k
8k
10k
12k