

卷积神经网络实验报告

姓名：杨鑫

学号：2011028

一、实验要求

- 掌握前馈神经网络（FFN）的基本原理
- 学会使用 PyTorch 搭建简单的 FFN 实现 MNIST 数据集分类
- 掌握如何改进网络结构、调试参数以提升网络识别性能

二、报告内容

- 运行原始版本 MLP，查看网络结构、损失和准确度曲线
- 尝试调节 MLP 的全连接层参数（深度、宽度等）、优化器参数等，以提高准确度
- 分析与总结格式不限
- 挑选 MLP-Mixer，ResMLP，Vision Permutator 中的一种进行实现（加分项）

三、实验步骤与心得

1. 实验环境配置

初次使用 pytorch 环境进行神经网络搭建，首先需要搭建 pytorch 的实验环境，由于电脑配置太低，因此本次实验配置了 CPU 的实验环境。

首先我们需要下载 anconda 工具，下载完成后使用 `conda --version` 查看是否下载成功(如果显示没有命令则可能需要配置环境变量)：

```
PS D:\学习资料\课程\深度学习\Deep-Learning> conda --version
conda 23.3.1
PS D:\学习资料\课程\深度学习\Deep-Learning> _
```

然后我们在 anconda 中新建虚拟环境，在 wind10 系统 CPU 中新建 pytorch 的虚拟环境：

```

C:\>conda create -n pytorch python=3.8
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: D:\Anaconda\envs\pytorch

  added / updated specs:
    - python=3.8

The following NEW packages will be INSTALLED:

ca-certificates      pkgs/main/win-64::ca-certificates-2023.01.10-haa9553
libffi               pkgs/main/win-64::libffi-3.4.2-hd77b12b_6
openssl              pkgs/main/win-64::openssl-1.1.1t-h2bbff1b_0
pip                  pkgs/main/win-64::pip-23.0.1-py38haa95532_0
python               pkgs/main/win-64::python-3.8.16-h6244533_3
setuptools           pkgs/main/win-64::setuptools-66.0.0-py38haa95532_0
sqlite               pkgs/main/win-64::sqlite-3.41.2-h2bbff1b_0

```

创建完成后可以使用命令 `conda info --envs` 查看虚拟环境是否创建成功,

然后使用 `activate pytorch` 激活 pytorch 的虚拟环境。

```

C:\>conda info --envs
# conda environments:
#
base                        D:\Anaconda
jittor                      D:\Anaconda\envs\jittor
pytorch                     D:\Anaconda\envs\pytorch

C:\Users\neverquit>activate pytorch

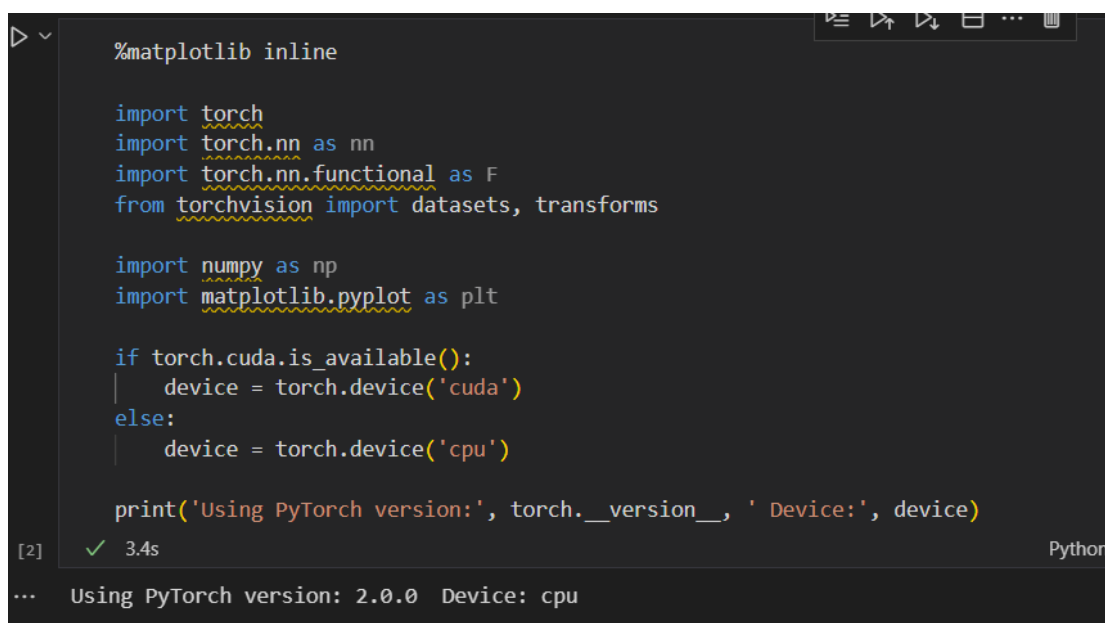
(pytorch) C:\Users\neverquit>a

```

然后在 pytorch 官网赋值安装 CPU 版本 pytorch 的命令:

PyTorch Build	Stable (2.0.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.7	CUDA 11.8	ROCm 5.4.2	CPU
Run this Command:	conda install pytorch torchvision torchaudio cpuonly -c pytorch			

安装完成后在 vscode 打开 jupyter notebook(插件，需要使用命令 **conda install -n pytorch ipykernel --update-deps --force-reinstall** 进行安装)，并且连接 conda 的虚拟环境内核。成功进行环境配置后，运行如下代码可以得到显示的结果为：



```
%matplotlib inline

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms

import numpy as np
import matplotlib.pyplot as plt

if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

print('Using PyTorch version:', torch.__version__, ' Device:', device)
```

[2] ✓ 3.4s Python

... Using PyTorch version: 2.0.0 Device: cpu

可以看到 Pytorch version 的版本为 2.0.0，Device 为 cpu。在实际运行过程中由于是新建的虚拟环境，因此可能存在包缺失的情况可以在 pytorch 虚拟环境下执行 **conda install package** 命令来安装所依赖的包。

2. MNIST 数据集

MNIST 数据集是一个手写数字图像数据集，包含了 60,000 个训练图像和 10,000 个测试图像，每个图像都是 28x28 像素大小的灰度图像。这个数据集通常被用来测试机器学习算法在图像识别任务上的性能。

在使用 pytorch 进行学习时，可以使用 pytorch 的处理图像视频的 torchvision 工具集直接下载 MNIST 的训练和测试图片，torchvision 包含了一些常用的数据集、模型和转换函数等等，比如图片分类、语义切分、目标识别、实例分割、关键点检测、视频分类等工具。

```
batch_size = 32

train_dataset = datasets.MNIST('./data',
                                train=True,
                                download=True,
                                transform=transforms.ToTensor())

validation_dataset = datasets.MNIST('./data',
                                     train=False,
                                     transform=transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                             batch_size=batch_size,
                                             shuffle=True)

validation_loader = torch.utils.data.DataLoader(dataset=validation_dataset,
                                                  batch_size=batch_size,
                                                  shuffle=False)
```

✓ 3.2s Python

Downloading <http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz>
Downloading <http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz> to ./data\MNI
100.0%
Extracting ./data\MNIST\raw\train-images-idx3-ubyte.gz to ./data\MNIST\raw

- **batch_size**: 每个批次的样本数量，这里设置为 32。
- **train_dataset**: MNIST 训练集数据集对象，通过 `datasets.MNIST` 函数创建，数据存储在 `./data` 目录下，通过 `transform=transforms.ToTensor()` 将图像转换为张量。
- **validation_dataset**: MNIST 测试集数据集对象，同样通过 `datasets.MNIST` 函数创建，数据存储在 `./data` 目录下，通过 `transform=transforms.ToTensor()` 将图像转换为张量。
- **train_loader**: 训练集数据加载器对象，通过 `torch.utils.data.DataLoader` 函数创建，将 `train_dataset` 数据集对象作为输入，设置批次大小为 `batch_size`，打乱数据顺序 (`shuffle=True`) 以增加模型的泛化能力。
- **validation_loader**: 测试集数据加载器对象，同样通过 `torch.utils.data.DataLoader` 函数创建，将 `validation_dataset` 数据集对象作为输入，设置批次大小为 `batch_size`，不打乱数据顺序 (`shuffle=False`)，

因为测试时需要按照原始顺序进行验证。

3. 原始 MLP 结构理解和运行

在手写字体的识别流程中加载完数据集后首先我们需要构建网络模型，在具体介绍网络之前这里介绍一下自己学到的专业名词以及解释：

- (1) 参数与超参数：模型 $f(x, \theta)$ 中的 θ 称为模型的参数，可以通过优化算法进行学习；而超参数是用来定义模型结构或优化策略的。
- (2) batch_size 批处理：每次处理的数据数量，这里小批量梯度下降方法。
- (3) epoch 轮次：把一个数据集，循环迭代几轮，迭代的过程就是梯度下降的过程。

构建网络模型的 Net 部分的代码如下：

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 100) # weight: [28*28,
50] bias: [50, ]
        self.fc1_drop = nn.Dropout(0.2)
        self.fc2 = nn.Linear(100, 80)
        self.fc2_drop = nn.Dropout(0.2)
        self.fc3 = nn.Linear(80, 10)

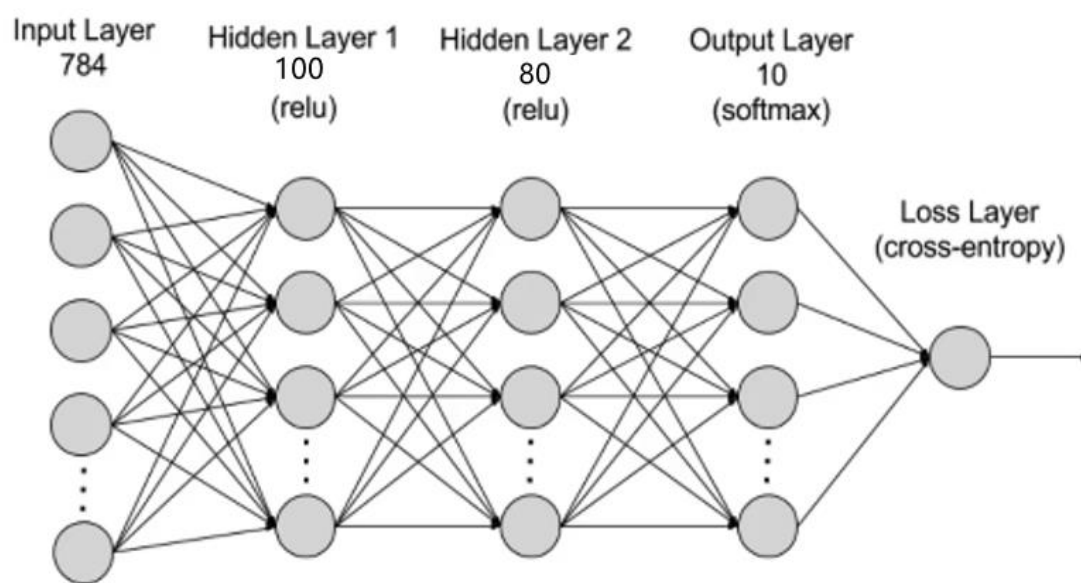
        self.relu1 = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 28*28) # [32, 28*28]
        x = F.relu(self.fc1(x))
        x = self.fc1_drop(x)
        x = F.relu(self.fc2(x))
        x = self.fc2_drop(x) # [32, 10]
        return F.log_softmax(self.fc3(x), dim=1)
```

首先需要在 Net 中定义 `__init__` 和 `forward` 两个方法，在 `__init__` 方法中，`super` 为调用父类的构造方法，即需要继承父类的一些方法，然后定义了两个全连接层，在每一个线性全连接层之后连接了一个 Dropout 层，随机失活 20% 的神经元，以防止过拟合问题。然后最后定义了第三个全连接层，输入大小为 80，输出大小为 10，即将输入特征映射到 10 个类别上。在 `forward` 方法中定义了模型前向传播的过程，`x.view` 将输入数据展开成一维向量，通过全连

接层并使用 ReLU 激活函数进行非线性变换，然后使用 fc1_drop 函数对第一个全连接层的输出进行随机失活。最后通过第三个全连接层，并使用 log_softmax 函数进行分类。

根据 Net 的构建，可以得到如下的网络结构图：



接下来是模型初始化和训练配置部分：

```
model = Net().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01,
momentum=0.5)
criterion = nn.CrossEntropyLoss()
```

首先创建了一个 Net 模型对象移到指定设备，然后定义了一个随机梯度下降 (SGD) 优化器对象，lr 为学习率，momentum 表示动量参数。最后定义交叉熵损失函数对象，用于计算模型预测值与真实标签之间的差距。

ReLU 激活函数：是一种常用的非线性激活函数，将所有负数输入变为 0，而正数输入则保持不变。它的公式为： $f(x) = \max(0, x)$ 。

Softmax 函数：一种将多维向量映射到概率分布上的函数，可以将神经网络输出的原始得分转换成概率值。

SGD 随机梯度下降：是一种常用的优化算法，用于训练神经网络。它通过计算损失函数对模型参数的梯度来更新参数，从而使损失函数最小化。每次更新时，SGD 只使用一个样本或一小批样本来计算梯度，而不是使用整个训练集。

交叉熵损失函数：一种常用的分类损失函数，用于衡量模型预测值与真实标签之间的差距。公式为 $H(p, q) = -\sum_{i=1}^n p_i \log(q_i)$ ，其中 p 表示真实标签的概率分布， q 表示模型预测的概率分布。交

叉熵损失函数越小，表示模型预测越准确。

然后需要去定义训练方法，

```
def train(epoch, log_interval=200):
    # Set model to training mode
    model.train()

    # Loop over each batch from the training set
    for batch_idx, (data, target) in enumerate(train_loader):
        # Copy data to GPU if needed
        data = data.to(device)
        target = target.to(device)

        # Zero gradient buffers
        optimizer.zero_grad()

        # Pass data through the network
        output = model(data)

        # Calculate loss
        loss = criterion(output, target)

        # Backpropagate
        loss.backward()

        # Update weights
        optimizer.step()      #  $w = w - \alpha * dL / dw$ 

        if batch_idx % 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.data.item()))
```

首先将模型设置为训练模式，然后遍历训练集中的每个批次，每个批次包含多个样本，然后将数据和标签拷贝到设备上，将梯度缓冲区清零，以避免梯度累积，将数据通过神经网络进行前向传播，得到预测输出，然后计算损失函数，用于衡量预测输出与真实标签之间的差距，然后对损失函数进行反向传播，计算参数的梯度，使用优化器更新模型参数，以最小化损失函数，最

后如果当前批次是 `log_interval` 的倍数，就输出训练进度信息，包括当前 epoch、处理的样本数量、总样本数量、完成进度百分比以及当前批次的损失值。

梯度累积问题是指在训练神经网络时，由于内存限制等原因，无法一次性将一个 batch 的所有样本都输入到模型中进行训练，需要将一个 batch 拆分成多个子 batch 进行训练。在这种情况下，每个子 batch 的梯度只是当前子 batch 的梯度，而不是整个 batch 的梯度。如果不对这些子 batch 的梯度进行处理，直接累加，会导致梯度累积问题。

验证过程包含了一下步骤：将模型设置为评估模式，然后遍历验证集中的每个批次，然后将数据通过神经网络进行前向传播得到预测输出，计算损失函数，统计预测正确的样本数得到平均损失和准确率，最后输出验证结果。

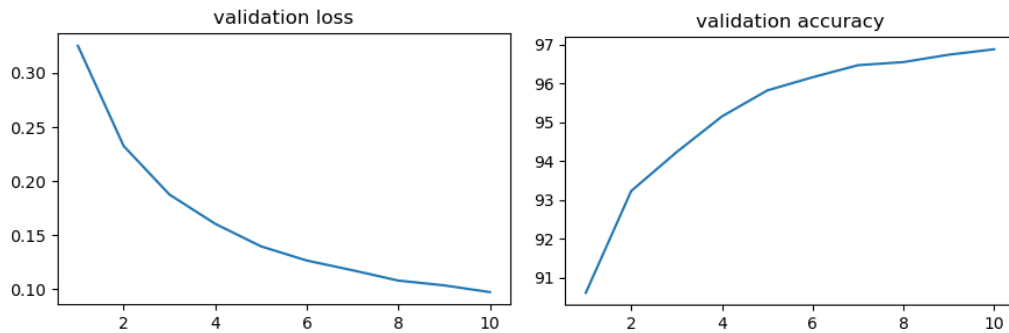
开始训练模型，输出预测结果，使用 CPU 来对 MLP 网络进行训练时间大致为 2min 左右，对于初始给定的 Net 结构训练得到的结构已经很好了，达到了 97%的正确率。

```
Train Epoch: 2 [0/60000 (0%)]    Loss: 0.226588
Train Epoch: 2 [6400/60000 (11%)]    Loss: 0.642302
Train Epoch: 2 [12800/60000 (21%)]    Loss: 0.206009
Train Epoch: 2 [19200/60000 (32%)]    Loss: 0.391120
Train Epoch: 2 [25600/60000 (43%)]    Loss: 0.403314
Train Epoch: 2 [32000/60000 (53%)]    Loss: 0.483281
Train Epoch: 2 [38400/60000 (64%)]    Loss: 0.213003
Train Epoch: 2 [44800/60000 (75%)]    Loss: 0.401656
Train Epoch: 2 [51200/60000 (85%)]    Loss: 0.180299
Train Epoch: 2 [57600/60000 (96%)]    Loss: 0.578514

Validation set: Average loss: 0.2324, Accuracy: 9323/10000 (93%)
...
Validation set: Average loss: 0.0974, Accuracy: 9688/10000 (97%)

CPU times: total: 8min 53s
Wall time: 2min 16s
```

可以查看训练过程中 Loss 曲线和 Accuracy 曲线如下：



4. 参数以及 MLP 网络结构优化

在我们 load 训练集的时候设定了 shuffle 参数为 True，也就是每次训练数据集的时候样本都是随机的，为了固定每次实验 load 训练集的顺序一致而不影响 Accuracy，因此这里需要将随机数种子进行固定。

```
import random
# 设置随机种子
# 固定 shuffle 随机数种子以及 cpu 等 backend 算法
seed = 10
random.seed(seed)
torch.manual_seed(seed) # 为 CPU 设置种子用于生成随机数，以使得结果是确定的
torch.backends.cudnn.deterministic = True
```

然后首先查看迭代次数对结果的影响，固定学习率和其他超参数不变：

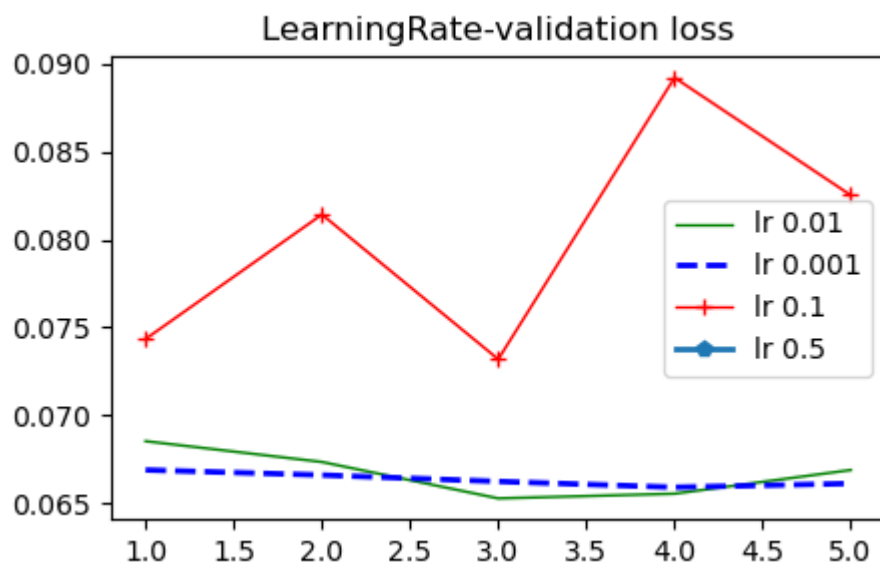
```
%%time
#查看迭代次数对于 Accuracy 的影响
plt.figure(figsize=(5,3))
for style, width, epochs in (('g-',1,1),('b--',1,5),('r--',1,10)):
    lossv, accv = [], []
    for epoch in range(1, epochs + 1):
        train(epoch)
        validate(lossv, accv)
    plt.plot(np.arange(1,epochs+1), accv,style,label='epochs'+str(epochs),linewidth=width)

plt.title('validation loss')
plt.legend()
plt.show()
```



可以看出过小的迭代次数可能使得 MLP 早停，造成较低的正确率(epochs=1 正确率低于 97.4%)，当迭代次数越高的时候，MLP 逐渐收敛得到较高的正确率(epochs=20 正确率为 98%)，但是迭代次数过高后 MLP 已经收敛因此正确率不会得到继续的提升。

然后固定迭代次数以及其他超参数不变查看学习率变换的情况下对于 Loss 结果的影响：



可以看出当学习率很低的时候，梯度下降的很慢，但学习率过高，梯度下降

的步子就很大，但很可能会跨过最优值，因此应该选择合适的学习率这里比如 0.01 在迭代次数很少的时候就可以达到很高的准确率。

然后尝试增加一层隐藏层(80,60)查看是否对结果有所优化：

```
Net(  
  (fc1): Linear(in_features=784, out_features=100, bias=True)  
  (fc1_drop): Dropout(p=0.2, inplace=False)  
  (fc2): Linear(in_features=100, out_features=80, bias=True)  
  (fc2_drop): Dropout(p=0.2, inplace=False)  
  (fc3): Linear(in_features=80, out_features=60, bias=True)  
  (fc3_drop): Dropout(p=0.2, inplace=False)  
  (f4): Linear(in_features=60, out_features=10, bias=True)  
)
```

然后再次训练，得到的准确率为 97%，可以看到对于模型的准确率和之前的网络结构训练得到的没有区别，但是对于更深的网络结构可以在 2-3 论训练就可以基本完成模型的训练，收敛的速度更快。

```
Validation set: Average loss: 0.1084, Accuracy: 9677/10000 (97%)
```

```
Train Epoch: 8 [0/60000 (0%)] Loss: 0.052358  
Train Epoch: 8 [6400/60000 (11%)] Loss: 0.029409  
Train Epoch: 8 [12800/60000 (21%)] Loss: 0.299065  
Train Epoch: 8 [19200/60000 (32%)] Loss: 0.330018  
Train Epoch: 8 [25600/60000 (43%)] Loss: 0.183410  
Train Epoch: 8 [32000/60000 (53%)] Loss: 0.144621  
Train Epoch: 8 [38400/60000 (64%)] Loss: 0.387120  
Train Epoch: 8 [44800/60000 (75%)] Loss: 0.017721  
Train Epoch: 8 [51200/60000 (85%)] Loss: 0.053807  
Train Epoch: 8 [57600/60000 (96%)] Loss: 0.091308
```

```
Validation set: Average loss: 0.1047, Accuracy: 9683/10000 (97%)
```

```
Train Epoch: 9 [0/60000 (0%)] Loss: 0.202926  
Train Epoch: 9 [6400/60000 (11%)] Loss: 0.113244  
Train Epoch: 9 [12800/60000 (21%)] Loss: 0.338725  
Train Epoch: 9 [19200/60000 (32%)] Loss: 0.090851  
Train Epoch: 9 [25600/60000 (43%)] Loss: 0.263933  
Train Epoch: 9 [32000/60000 (53%)] Loss: 0.199589  
Train Epoch: 9 [38400/60000 (64%)] Loss: 0.084359  
Train Epoch: 9 [44800/60000 (75%)] Loss: 0.250723  
Train Epoch: 9 [51200/60000 (85%)] Loss: 0.279225  
Train Epoch: 9 [57600/60000 (96%)] Loss: 0.017885
```

```
Validation set: Average loss: 0.0991, Accuracy: 9708/10000 (97%)
```

然后固定其他超参数，选择不同的 optimizer 和 loss 策略进行测试，在对 optimizer 优化策略中，在 torch 中可以选择的有很多：

Algorithms

Adadelata

Implements Adadelata algorithm.

Adagrad

Implements Adagrad algorithm.

Adam

Implements Adam algorithm.

AdamW

Implements AdamW algorithm.

SparseAdam

Implements lazy version of Adam algorithm suitable for sparse tensors.

在测试过程中我选择了 SGD、AdaGrad、RMSprop、Adam 的方法。

SGD (随机梯度下降): 每次更新时, 随机选择一个样本进行梯度计算, 并更新模型参数。SGD 算法简单高效, 但可能会受到噪声和局部最优解的影响。

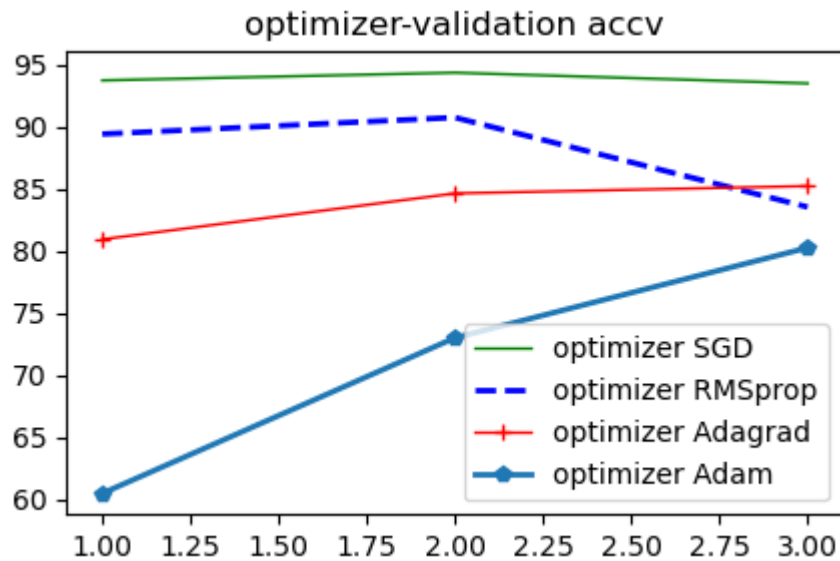
AdaGrad: 根据每个参数的历史梯度信息来调整学习率, 使得梯度较大的参数的学习率较小, 而梯度较小的参数的学习率较大。这种方式可以自适应地调整学习率, 但可能会导致学习率逐渐降低, 导致训练过程过早停止。

RMSprop: 与 AdaGrad 类似, 但使用了一个滑动平均来平衡历史梯度和当前梯度, 以避免学习率逐渐降低的问题。RMSprop 算法可以更好地适应非平稳目标函数。

Adam: 结合了动量梯度下降和 RMSprop 算法的优点。Adam 算法不仅可以自适应地调整学习率, 还可以在优化过程中保持梯度的稳定性和动量, 从而提高训练效果。Adam 算法是目前最流行的优化算法之一。

代码如下:

```
model = Net().to(device)
# 选择不同的优化策略
# optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01, lr_decay=0,
weight_decay=0,initial_accumulator_value=0)
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.01, alpha=0.5)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, betas=(0.9,0.99))
criterion = nn.CrossEntropyLoss()
```



由结果可以发现对于 MNIST 这种不是很大很复杂的数据集而言，使用更加复杂的优化策略，在分类精度上反而不如简单的优化策略了。

对于 loss 策略的优化中我们发现 MSE 损失函数并不适用于分类任务，而更加适用于回归任务，因此这里不再进行展示。

ROC-AUC 曲线: AUC-ROC 曲线是针对各种阈值设置下的分类问题的性能度量。ROC 是概率曲线，AUC 表示可分离的程度或测度，它告诉我们多少模型能够区分类别。AUC 越高，模型在将 0 预测为 0，将 1 预测为 1 时越好。

在给出的初始代码中，仅展示了每轮训练的损失值以及准确率吧，并没有给出常用的 ROC 曲线，因此在初始代码的基础之上对 ROC-AUC 曲线进行补充：

```
num_class = 10
score_array = np.array(score_list)
# 将 label 转换成 onehot 形式
label_tensor = torch.tensor(label_list)
label_tensor = label_tensor.reshape((label_tensor.shape[0],
1))
label_onehot = torch.zeros(label_tensor.shape[0], num_class)
label_onehot.scatter_(dim=1, index=label_tensor, value=1)
label_onehot = np.array(label_onehot)
print("score_array:", score_array.shape) # (batchsize,
classnum)
```

```

print("label_onehot:", label_onehot.shape) #
torch.Size([batchsize, classnum])

# 调用 sklearn 库, 计算每个类别对应的 fpr 和 tpr
fpr_dict = dict()
tpr_dict = dict()
roc_auc_dict = dict()
for i in range(num_class):
    fpr_dict[i], tpr_dict[i], _ = roc_curve(label_onehot[:,
i], score_array[:, i])
    roc_auc_dict[i] = auc(fpr_dict[i], tpr_dict[i])
# micro
fpr_dict["micro"], tpr_dict["micro"], _ =
roc_curve(label_onehot.ravel(), score_array.ravel())
roc_auc_dict["micro"] = auc(fpr_dict["micro"],
tpr_dict["micro"])

# macro
# First aggregate all false positive rates
all_fpr = np.unique(np.concatenate([fpr_dict[i] for i in
range(num_class)]))
# Then interpolate all ROC curves at this points
mean_tpr = np.zeros_like(all_fpr)
for i in range(num_class):
    mean_tpr += np.interp(all_fpr, fpr_dict[i], tpr_dict[i])
# Finally average it and compute AUC
mean_tpr /= num_class
fpr_dict["macro"] = all_fpr
tpr_dict["macro"] = mean_tpr
roc_auc_dict["macro"] = auc(fpr_dict["macro"],
tpr_dict["macro"])

plt.figure()
lw = 2
plt.plot(fpr_dict["micro"], tpr_dict["micro"],
        label='micro-average ROC curve (area = {0:0.2f})'
        ''.format(roc_auc_dict["micro"]),
        color='deeppink', linestyle=':', linewidth=4)

plt.plot(fpr_dict["macro"], tpr_dict["macro"],
        label='macro-average ROC curve (area = {0:0.2f})'
        ''.format(roc_auc_dict["macro"]),
        color='navy', linestyle=':', linewidth=4)

```

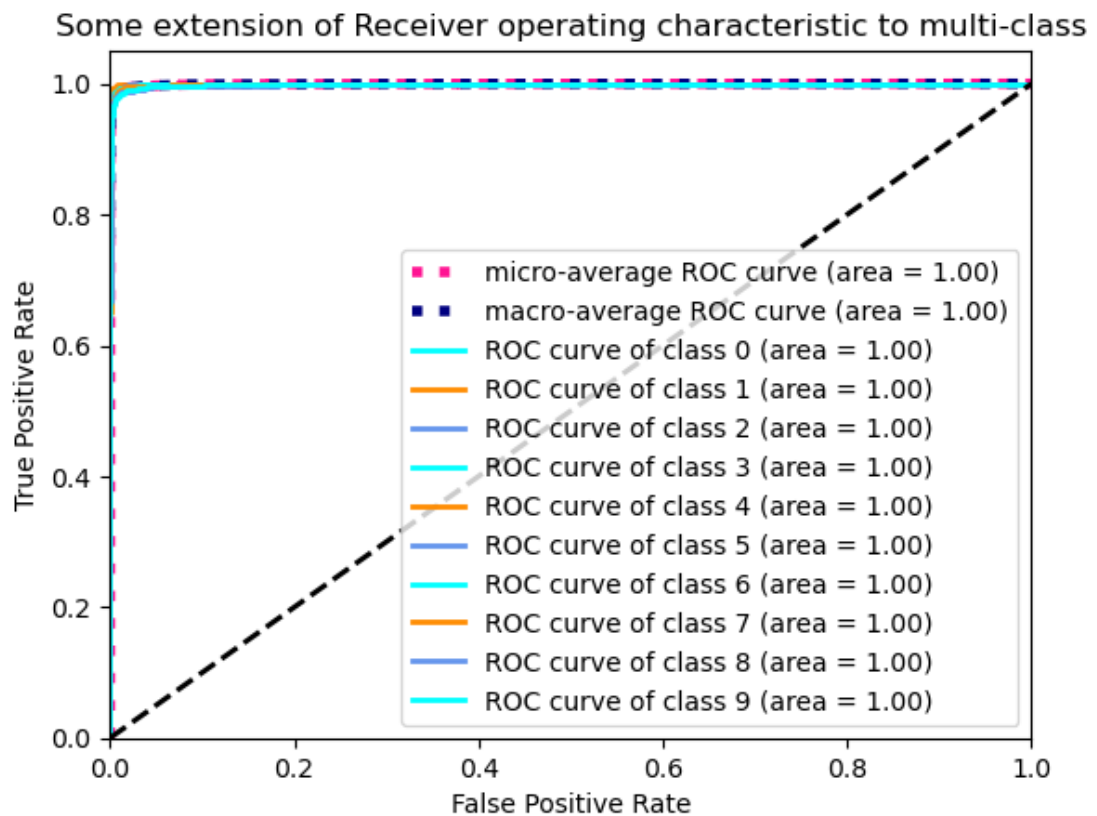
```

colors = cycle(['aqua', 'darkorange', 'cornflowerblue'])
for i, color in zip(range(num_class), colors):
    plt.plot(fpr_dict[i], tpr_dict[i], color=color, lw=lw,
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc_dict[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Some extension of Receiver operating characteristic
to multi-class')
plt.legend(loc="lower right")
# plt.savefig('set113_roc.jpg')
plt.show()

```

得到的 ROC-AUC 曲线：

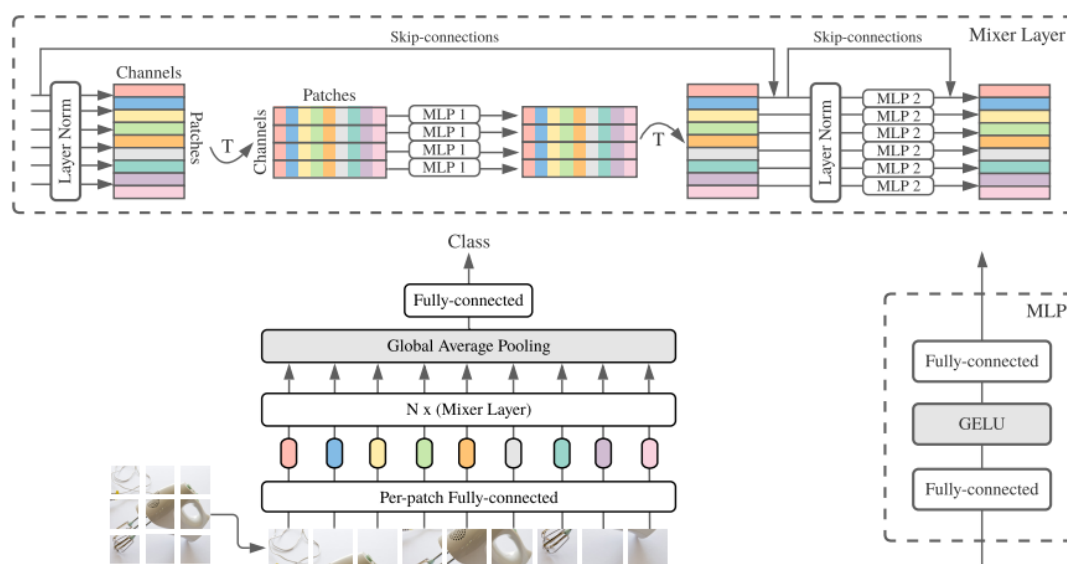


可以发现 MNIST 的不同类的分类效果基本相同都很接近 (0, 1)，都表现出了很好的分类效果。

5. MLP-Mixer 学习及实现

MLP-Mixer 是 Google 提出的一种新兴的神经网络架构，用于图像分类和其他视觉任务。与传统的卷积神经网络不同，MLP-Mixer 使用了一种称为 token mixing 的操作来实现卷积神经网络中的特征提取和全连接神经网络中的分类器。具体来说，MLP-Mixer 由多个基本块组成，每个基本块包含两个操作：channel mixing 和 token mixing。channel mixing 操作将输入特征图在通道维度上进行混合，以促进不同通道之间的信息交流；token mixing 操作将输入特征图在空间维度上进行混合，以捕捉局部和全局特征。与传统卷积神经网络相比，MLP-Mixer 具有更少的参数和更好的可解释性，同时在各种视觉任务上取得了与最先进方法相当的性能。

MLP-Mixer 的整体结构如下(图片来源于论文):



MLP-Mixer 主要包括三部分：Per-patch Fully-connected、Mixer Layer、分类器。

其中分类器部分采用传统的全局平均池化（GAP）+全连接层（FC）+Softmax 的方式构成。

对于 MNIST 手写体识别任务，可以使用 MLP-Mixer 模型来构建一个分类器。具

体来说，可以将 MNIST 图像转换为向量形式，并将其输入到 MLP-Mixer 模型中进行分类。在训练过程中，可以使用交叉熵损失函数来衡量模型在 MNIST 数据集上的分类性能，并使用优化算法对模型参数进行更新。在测试过程中，可以使用训练好的模型对新的手写数字图像进行分类。

通过上述学习，这里本人根据前面学习知识使用 PyTorch 框架实现 MLP-Mixer 进行手写体识别的代码。

首先实现 MLP-Mixer 代码，这里参考以下仓库进行实现：[lucidrains/mlp-mixer-pytorch: An All-MLP solution for Vision, from Google AI \(github.com\)](https://github.com/lucidrains/mlp-mixer-pytorch)

```
from torch import nn
from functools import partial
from einops.layers.torch import Rearrange, Reduce

pair = lambda x: x if isinstance(x, tuple) else (x, x)

class PreNormResidual(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.fn = fn
        self.norm = nn.LayerNorm(dim)

    def forward(self, x):
        return self.fn(self.norm(x)) + x

def FeedForward(dim, expansion_factor = 4, dropout = 0., dense =
nn.Linear):
    inner_dim = int(dim * expansion_factor)
    return nn.Sequential(
        dense(dim, inner_dim),
        nn.GELU(),
        nn.Dropout(dropout),
        dense(inner_dim, dim),
        nn.Dropout(dropout)
    )

def MLP Mixer(*, image_size, channels, patch_size, dim, depth,
num_classes, expansion_factor = 4, expansion_factor_token = 0.5,
dropout = 0.):
```

```

image_h, image_w = pair(image_size)
assert (image_h % patch_size) == 0 and (image_w % patch_size) ==
0, 'image must be divisible by patch size'
num_patches = (image_h // patch_size) * (image_w // patch_size)
chan_first, chan_last = partial(nn.Conv1d, kernel_size = 1),
nn.Linear

    return nn.Sequential(
        Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 =
patch_size, p2 = patch_size),
        nn.Linear((patch_size ** 2) * channels, dim),
        *[nn.Sequential(
            PreNormResidual(dim, FeedForward(num_patches,
expansion_factor, dropout, chan_first)),
            PreNormResidual(dim, FeedForward(dim,
expansion_factor_token, dropout, chan_last))
        ) for _ in range(depth)],
        nn.LayerNorm(dim),
        Reduce('b n c -> b c', 'mean'),
        nn.Linear(dim, num_classes)
    )

```

具体来说，该模型包含以下几个组件：

- **Rearrange 层**：用于将输入图像按照 `patch_size` 划分成多个 patch，并将每个 patch 展平为一个向量。
 - **nn.Linear 层**：用于将每个 patch 的向量映射到一个维度为 `dim` 的特征空间。
- 多个 **PreNormResidual** 块：每个块包含两个部分，分别是一个 MLP 和一个残差连接。其中，MLP 由两个全连接层组成，用于对每个 patch 的特征进行非线性变换；残差连接用于保留原始输入的信息。
- **nn.LayerNorm 层**：用于对整张图像的特征进行归一化。
 - **Reduce 层**：用于对所有 patch 的特征进行平均池化，得到整张图像的特征表示。
 - **nn.Linear 层**，用于将图像特征映射到类别概率。

使用定义的超参数去初始化模型，可见这里使用的 optimizer 和 loss 优化策略分别为 Adam 和 MSE：

```
model = MLP Mixer(  
    image_size = 28,  
    patch_size = 7,  
    dim = 14,  
    depth = 3,  
    num_classes = 10,  
    channels = 1  
)  
model.to(DEVICE)  
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)  
mse = nn.MSELoss()
```

然后测试迭代，生成结果并保存相关模型数据。

```
for epoch in range(n_epochs):  
    train(model, DEVICE, train_loader, optimizer, epoch)  
    validate(model, DEVICE, validation_loader)  
    torch.save(model.state_dict(), './model.pth')  
    torch.save(optimizer.state_dict(), './optimizer.pth')
```

得到最后的结果为：

```
Train Epoch: 2 [38568/60000 (64%)] tLoss: 0.003348  
Train Epoch: 2 [47968/60000 (80%)] tLoss: 0.008716  
Train Epoch: 2 [57568/60000 (96%)] tLoss: 0.004011  
...  
Train Epoch: 9 [57568/60000 (96%)] tLoss: 0.002345  
  
Validation set: Average loss: 0.0002, Accuracy: 9668/10000 (97%)
```

在经过 10 轮的迭代后精度可以达到 97%，这是因为 MNIST 数据集较小，也没有那么复杂，所以得到的分类精度与 MLP 相差不多。

四、实验总结

本次实验通过对 MLP 和 MLP-Mixer 两种深度学习模型的学习，应用到 MNIST 手写字体识别分类任务上，通过手动调参观察了超参数对于模型准确度的影响，并通过实现 MLP-Mixer 模型看出此模型在图像分类问题上面的优势。

五、参考资料

[1] <https://arxiv.org/pdf/2105.01601.pdf>

[2] [【论文+代码】额，MLP 有这么厉害？谷歌的 MLP-Mixer_哔哩哔哩_bilibili](#)

[3] [lucidrains/mlp-mixer-pytorch: An All-MLP solution for Vision, from Google AI \(github.com\)](#)