## 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**

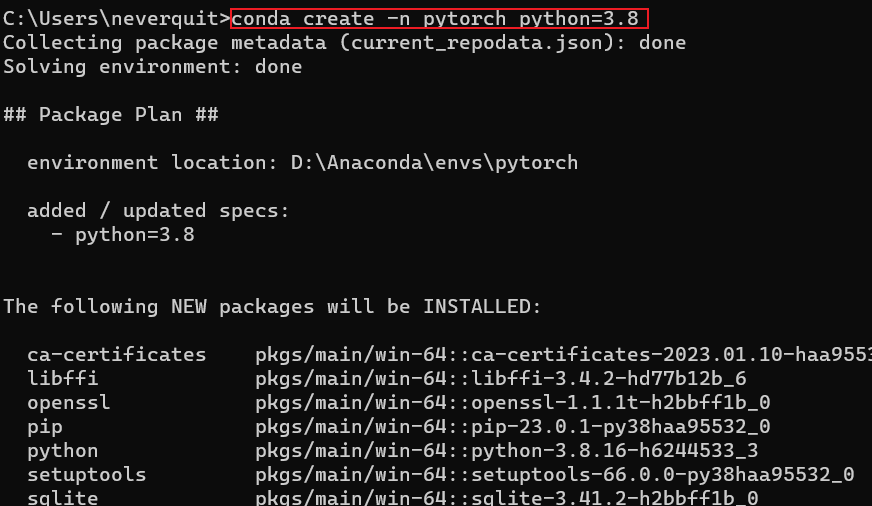
1. **实验环境配置**

|  |
| --- |
| 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)：

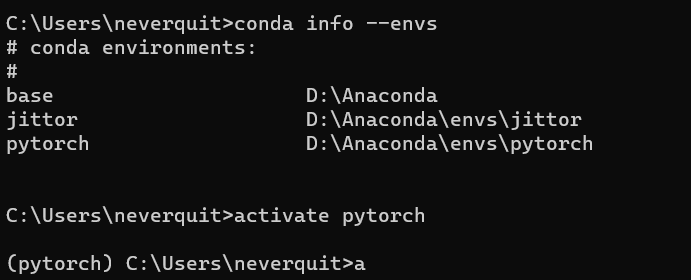


然后我门在anconda中新建虚拟环境，在wind10系统CPU中新建pytorch的虚拟环境：



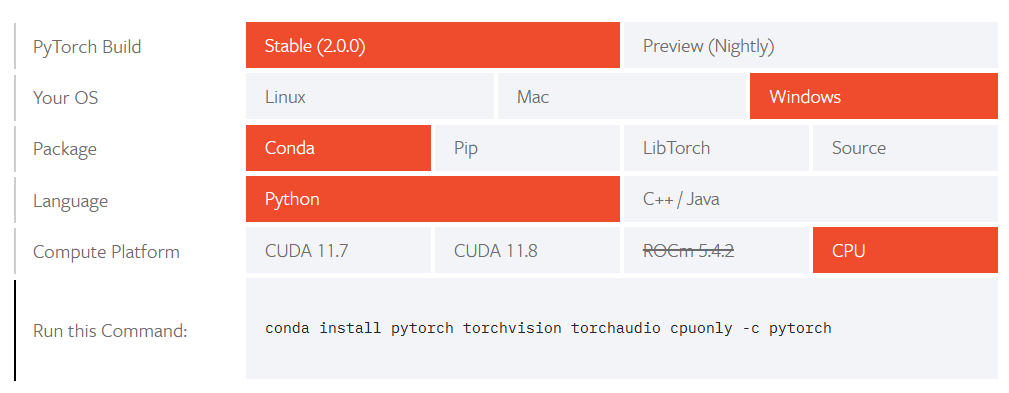


创建完成后可以使用命令**conda info –envs** 查看虚拟环境是否创建成功，然后使用 **activate pytorch** 激活pytorch的虚拟环境。

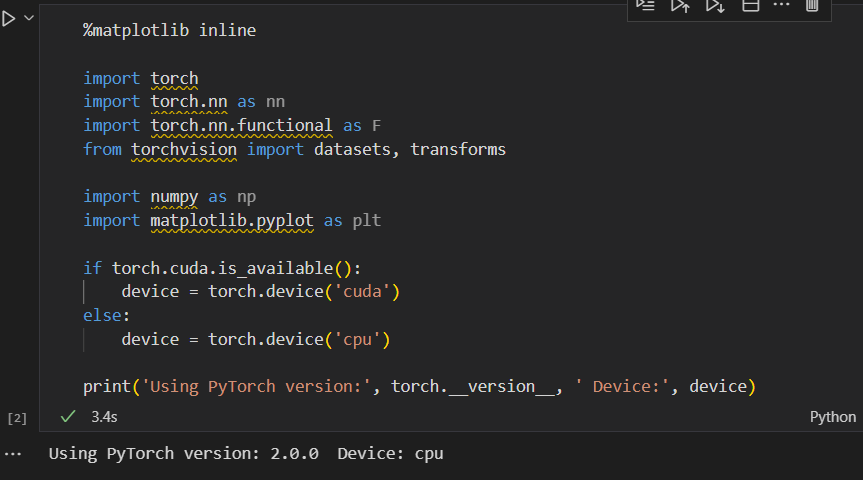




然后在pytorch官网赋值安装CPU版本pytorch的命令：



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：

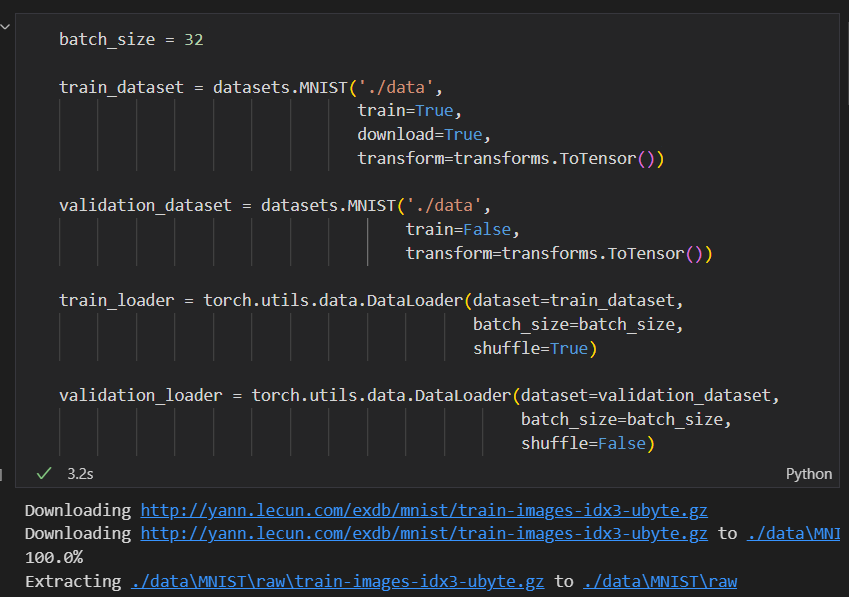


可以看到Pytorch version的版本为 2.0.0，Device为cpu。在实际运行过程中由于是新建的虚拟环境，因此可能存在包缺失的情况可以在pytorch虚拟环境下执行 **conda install package** 命令来安装所依赖的包。

1. **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**: 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\_datase**t: 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。

1. **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：

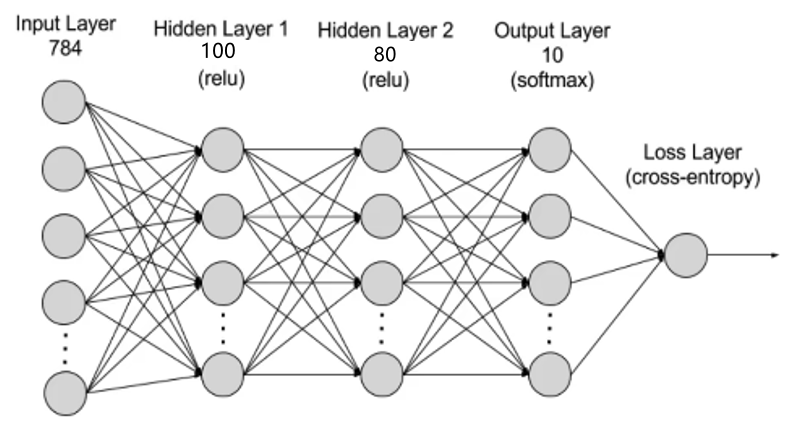
|  |
| --- |
| (1) 参数与超参数：模型f(x,θ)中的θ称为模型的参数，可以通过优化算法进行学习；而超参数是用来定义模型结构或优化策略的。  (2) batch\_size批处理：每次处理的数据数量，这里小批量梯度下降方法。  (3) epoch轮次：把一个数据集，循环迭代几轮，迭代的过程就是梯度下降的过程。 |

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, 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，

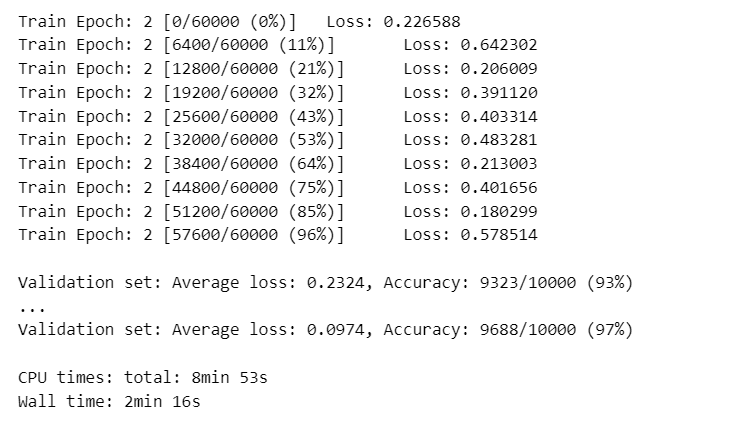
|  |
| --- |
| def train(epoch, log\_interval=200):      # Set model to training mode      model.train()        # Loop over each batch from the training set      for batch\_idx, (data, target) in enumerate(train\_loader):          # Copy data to GPU if needed          data = data.to(device)          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 - 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。

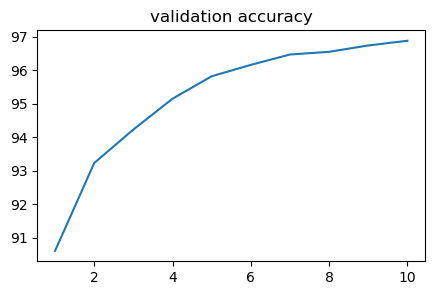
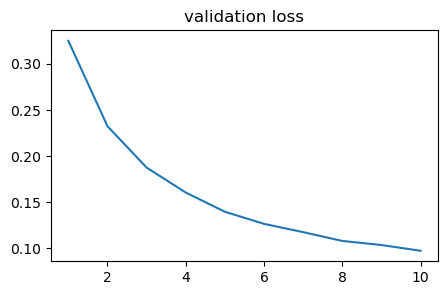
|  |
| --- |
| 梯度累积问题是指在训练神经网络时，由于内存限制等原因，无法一次性将一个batch的所有样本都输入到模型中进行训练，需要将一个batch拆分成多个子batch进行训练。在这种情况下，每个子batch的梯度只是当前子batch的梯度，而不是整个batch的梯度。如果不对这些子batch的梯度进行处理，直接累加，会导致梯度累积问题。 |

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。



可以查看训练过程中Loss曲线和Accuracy曲线如下：



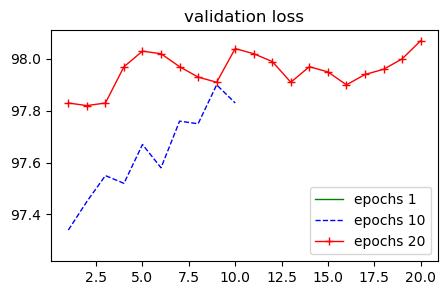
1. **参数以及MLP网络结构优化**

在我们load训练集的时候设定了shuffe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 |

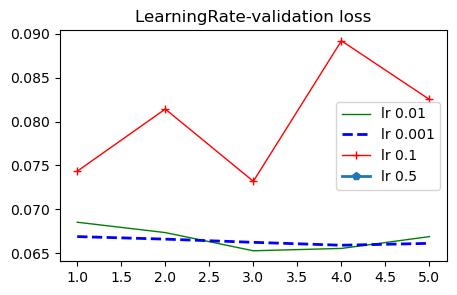
然后**首先查看迭代次数对结果的影响，固定学习率和其他超参数不变**：

|  |
| --- |
| %%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() |



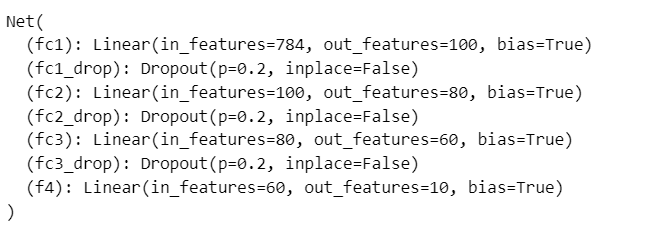
可以看出过小的迭代次数可能使得MLP早停，造成较低的正确率(epochs=1正确率低于97.4%)，当迭代次数越高的时候，MLP逐渐收敛得到较高的正确率(epochs=20正确率为98%)，但是迭代次数过高后MLP已经收敛因此正确不会得到继续的提升。

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**：

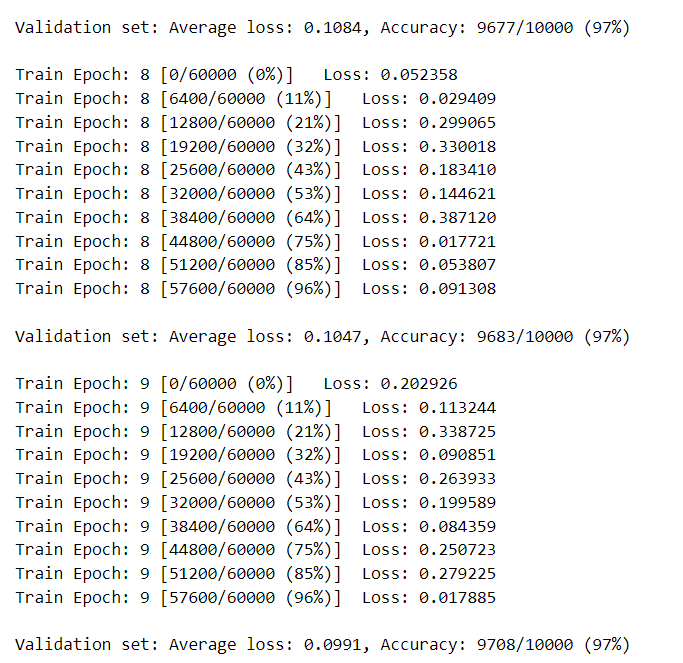
****

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：



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。



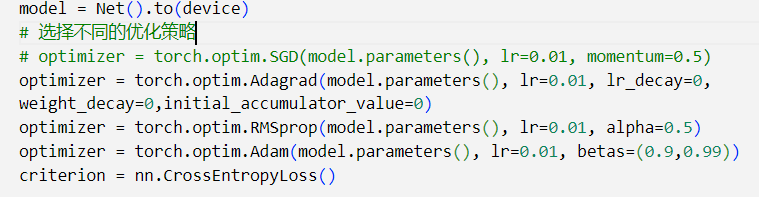
然后固定其他超参数，选择不同的optimizer和loss策略进行测试，在对optimizer优化策略中，在torch中可以选择的有很多：

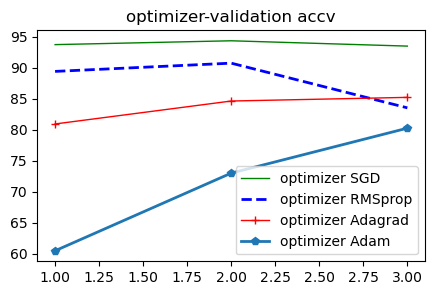


在测试过程中我选择了SGD、AdaGrad、RMSprop、Adam的方法。

|  |
| --- |
| 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：





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。

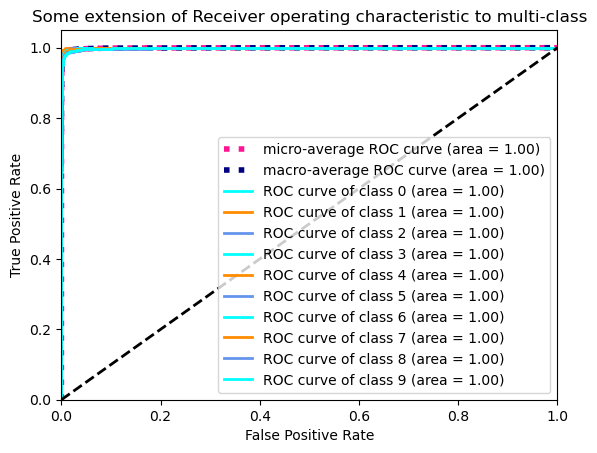
对于loss策略的优化中我们发现MSE损失函数并不适用于分类任务，而更加适用于回归任务，因此这里不再进行展示。

|  |
| --- |
| 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：

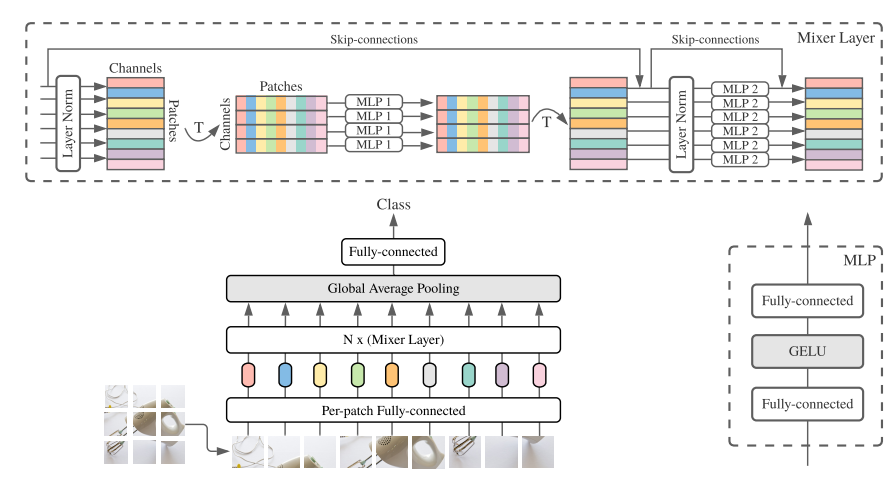


可以发现MNIST的不同类的分类效果基本相同都很接近（0，1），都表现出了很好的分类效果。

1. **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 MLPMixer(\*, 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。

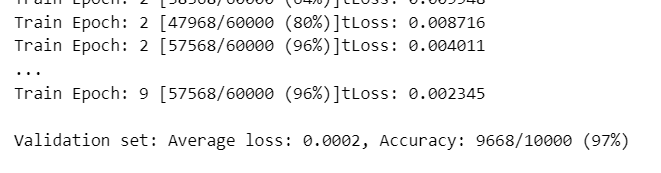
使用定义的超参数去初始化模型，可见这里使用的optimizer和loss优化策略分别为Adam和MSE：

|  |
| --- |
| model = MLPMixer(      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：



在经过10轮的迭代后精度可以达到97%，这是因为MINST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](https://www.bilibili.com/video/BV1Af4y1Y7fE/?spm_id_from=333.337.search-card.all.click&vd_source=2056100b00440c395c8d7455c3abe6ab)

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