MLP

> 2013551 雷贺奥

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

作者在最后实现了MLP_MIXER，收敛速度和准确率相较于MLP都有提升。

复现原始MLP

网络结构

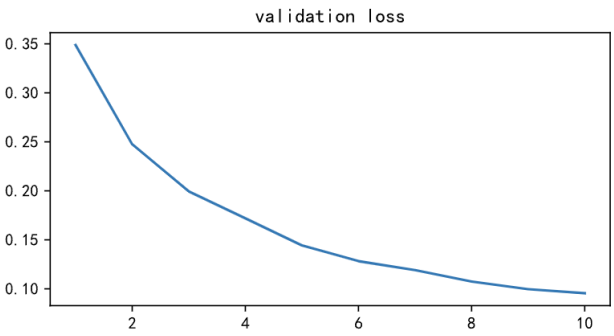
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        #全连接层
        # weight: [28*28, 50]   bias: [50, ]
        self.fc1 = nn.Linear(28*28, 100)
        #在训练过程的前向传播中，让每个神经元以一定概率p处于不激活的状态。以达到减少过拟合的
效果。
        self.fc1_drop = nn.Dropout(0.2)
        #全连接层
        self.fc2 = nn.Linear(100, 80)
        #在训练过程的前向传播中，让每个神经元以一定概率p处于不激活的状态。以达到减少过拟合的
效果。
        self.fc2_drop = nn.Dropout(0.2)
        #全连接层
        self.fc3 = nn.Linear(80, 10)
```

```
        #self.relu1 = nn.ReLU()

    def forward(self, x):
        #传入数字-1，自动对维度进行变换
        x = x.view(-1, 28*28)   # [32, 28*28]
        #relu激活函数对self.fc1(x)激活
        x = F.relu(self.fc1(x))
        #drop
        x = self.fc1_drop(x)
        #relu激活函数对self.fc2(x)激活
        x = F.relu(self.fc2(x))
        #drop
        x = self.fc2_drop(x)   # [32, 10]
        #self.fc3()全连接后，softmax激活
        return F.log_softmax(self.fc3(x), dim=1)
```

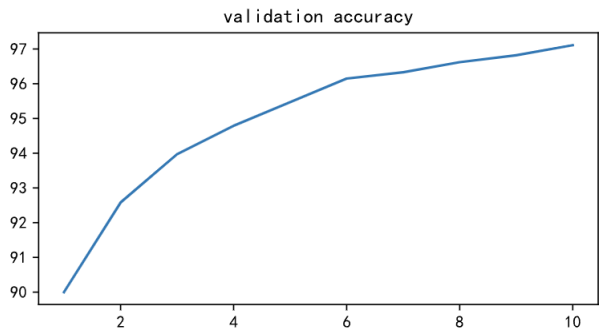
使用print()函数，打印网络结构：

```
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=10, bias=True)
)
```

loss曲线



准确度曲线



最终MLP的准确率稳定在97%左右，效果良好。

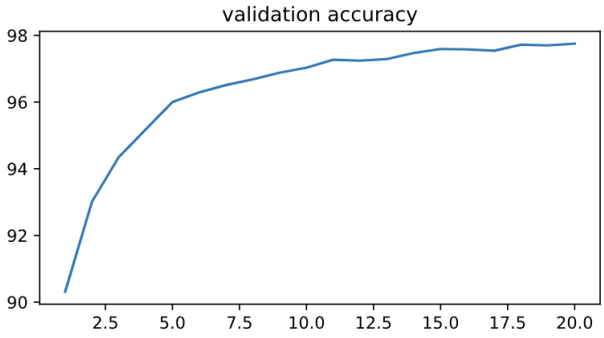
调节MLP参数，以提高准确度

采用控制变量法

调整epoch

将epoch改为20

准确度曲线如下：



最终MLP的准确率稳定在98%，超过了epoch=10的97%。

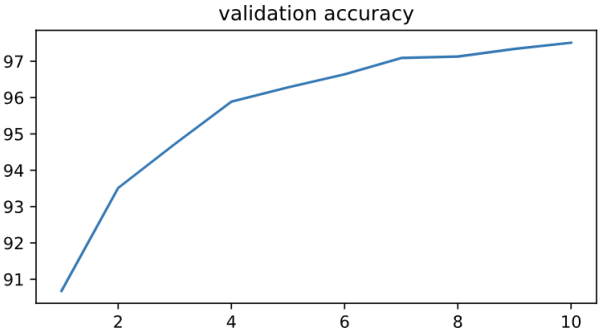
隐藏层参数（宽度）

输入层(28*28)->隐层1(512)->Dropout()->隐层2(128)->Droupout->输出层(10)

```
Net(
  (fc1): Linear(in_features=784, out_features=512, bias=True)
  (fc1_drop): Dropout(p=0.2, inplace=False)
  (fc2): Linear(in_features=512, out_features=128, bias=True)
  (fc2_drop): Dropout(p=0.2, inplace=False)
  (fc3): Linear(in_features=128, out_features=10, bias=True)
)
```

> Validation set: Average loss: 0.0793, Accuracy: 9751/10000 (98%)
> CPU times: total: 14min 37s
> Wall time: 1min 49s

准确度曲线如下：



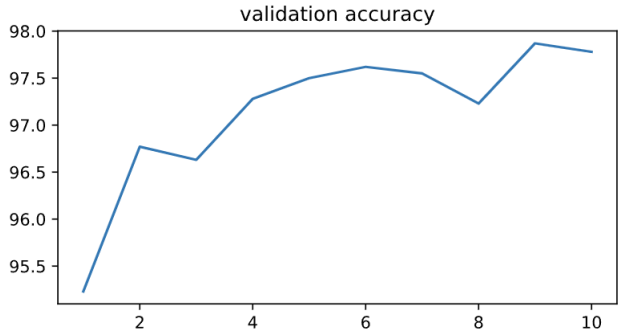
最终MLP的准确率稳定在98%，超过了原始项目的97%。

optimizer

加入动态学习率机制。epoch 每增加2，学习率就乘上0.98，由于学习率递减，所以一开始的学习率可以设置的大一些，这里设置为0.05

```
#随机梯度下降（优化器更新参数）
optimizer = torch.optim.SGD(model.parameters(), lr=0.05, momentum=0.5)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer,step_size=2, gamma=0.98)

Validation set: Average loss: 0.0741, Accuracy: 9778/10000 (98%)
CPU times: total: 12min 37s
Wall time: 1min 34s
```

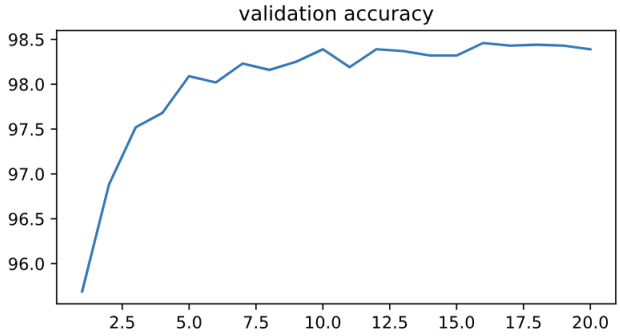
准确度曲线如下：



最终MLP的准确率稳定在98%，超过了原始项目的97%。

最好的MLP

将上面的三种方法结合到一起。



最终MLP的准确率稳定在98.5%，超过了原始项目的97%。

MLP_MIXER

网络结构

MLP

Fully-connected->GELU->Fully-connected。

```
# 因为有两个mixing,进出的维度都不变，只是中间全连接层的神经元数量不同
# 定义多层感知机
class FeedForward(nn.Module):
    def __init__(self,dim,hidden_dim,dropout=0.):
        super().__init__()
        self.net=nn.Sequential(
            #由此可以看出 FeedForward 的输入和输出维度是一致的
            nn.Linear(dim,hidden_dim),
            #激活函数
            nn.GELU(),
            #防止过拟合
```

```
            nn.Dropout(dropout),
            #重复上述过程
            nn.Linear(hidden_dim,dim),
            nn.Dropout(dropout)
        )
    def forward(self,x):
        x=self.net(x)
        return x
```

Mixer Block

从Layer Norm层出来的为Patches*Channel(即为table)，每个patch即上述对同一位置的所有通道进行展开，

通过T转为Channel*Patches，token-mixing MLPs（MLP1）对table的列进行映射；

channel-mixing MLPs（MLP2）对table的行进行映射，对同一空间位置在不同通道上的信息进行映射；

```
class MixerBlock(nn.Module):
    def __init__(self,dim,num_patch,token_dim,channel_dim,dropout=0.):
        super().__init__()
        #MLP1：token_mixer对列进行映射
        self.token_mixer=nn.Sequential(
            nn.LayerNorm(dim),
            #进行转置，第一维不变，二维<->三维
            Rearrange('b n d -> b d n'),
            #num_patch=channels,且token_dim为内部节点数，输出输入相同
            FeedForward(num_patch,token_dim,dropout),
            #转置回来
            Rearrange('b d n -> b n d')

         )
        #MLP2：channel_mixer对行进行映射
        self.channel_mixer=nn.Sequential(
            nn.LayerNorm(dim),
            FeedForward(dim,channel_dim,dropout)
        )
    def forward(self,x):
        #跳跃连接
        x = x+self.token_mixer(x)
        x = x+self.channel_mixer(x)
        return x
```

搭建MLP_MIXER

```
class MLPMixer(nn.Module):
    def
__init__(self,in_channels,dim,num_classes,patch_size,image_size,depth,token_dim,c
hannel_dim,dropout=0.):
        super().__init__()
        #不能划分为一个个patch报错
        assert image_size%patch_size==0
        self.num_patches=(image_size//patch_size)**2   # （224/16）**2=196
        # embedding 操作，看见没用卷积来分成一小块一小块的
        # 通过embedding可以将这张3*224*224的图片转换为Channel*Patches=512*196，再通过
Rearrange转为196*512
```

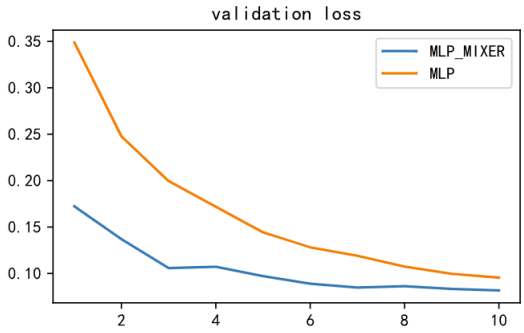
```
        self.to_embedding=nn.Sequential(
            #kernel_size=patch_size,stride=patch_size，切分

 Conv2d(in_channels=in_channels,out_channels=dim,kernel_size=patch_size,stride=pa
tch_size),
            Rearrange('b c h w -> b (h w) c')
        )

        # 以下为token-mixing MLPs（MLP1）和channel-mixing MLPs（MLP2）各一层
        #N*MIX_block中的N为depth
        self.mixer_blocks=nn.ModuleList([])
        for _ in range(depth):

 self.mixer_blocks.append(MixerBlock(dim,self.num_patches,token_dim,channel_dim,d
ropout))

        #normal
        self.layer_normal=nn.LayerNorm(dim)

        #全连接层，输出分类
        self.mlp_head=nn.Sequential(
            nn.Linear(dim,num_classes)
        )
    def forward(self,x):
        x = self.to_embedding(x)
        for mixer_block in self.mixer_blocks:
            x = mixer_block(x)
        x = self.layer_normal(x)
        #global average pooling
        x = x.mean(dim=1)
        x = self.mlp_head(x)
        return x
```

使用**torchsummary**打印网络结构，其功能特别强大，各层网络均给出shape，如下所示:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 256, 4, 4]          12,800
         Rearrange-2              [-1, 16, 256]               0
         LayerNorm-3              [-1, 16, 256]             512
         Rearrange-4              [-1, 256, 16]               0
            Linear-5             [-1, 256, 128]           2,176
              GELU-6             [-1, 256, 128]               0
           Dropout-7             [-1, 256, 128]               0
            Linear-8              [-1, 256, 16]           2,064
           Dropout-9              [-1, 256, 16]               0
      FeedForward-10              [-1, 256, 16]               0
        Rearrange-11              [-1, 16, 256]               0
        LayerNorm-12              [-1, 16, 256]             512
           Linear-13             [-1, 16, 1024]         263,168
             GELU-14             [-1, 16, 1024]               0
          Dropout-15             [-1, 16, 1024]               0
           Linear-16              [-1, 16, 256]         262,400
          Dropout-17              [-1, 16, 256]               0
      FeedForward-18              [-1, 16, 256]               0
       MixerBlock-19              [-1, 16, 256]               0
```

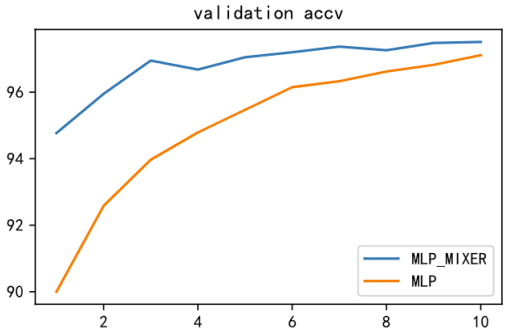
```
        LayerNorm-20              [-1, 16, 256]             512
           Linear-21                   [-1, 10]           2,570
================================================================
```

loss曲线



由此可见，MLP_MIXER比MLP收敛的速度快得多。

准确度曲线



由此可见，MLP_MIXER比MLP准确率更高，效果更好。

实验心得

- 隐藏层的数量较少,则模型的表示能力较弱；但是隐藏层的数量多，模型的表示能力也不一定强，作者尝试在原始MLP中加入隐藏层时，最后的准确率并没有增加。
- 如果学习率较小，则模型学习的能力较弱，收敛的速度更慢。如果学习率相对较大，则模型学习的能力会变强，收敛的速度更快，但容易梯度爆炸。可以采用动态学习率的方法，先设置一个较大的学习率，逐步减小。
- 隐藏层中神经元的数量较少,则模型的表示能力较弱,可能无法准确地拟合训练数据。此时可以增加隐藏层中神经元的数量,则模型的表示能力较强,收敛速度更快。

收获最大就是终于区分了Batch和Epoch

- Batch：训练数据集可以分为一个或多个Batch。当所有训练样本用于创建一个Batch时，学习算法称为批量梯度下降。当批量是一个样本的大小时，学习算法称为随机梯度下降。
- Epoch：在整个训练数据集中的循环次数。