深度学习-前馈神经网络

学号：2014074 姓名：费泽锟

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

实验内容

- 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
- 挑选MLP-Mixer，ResMLP，Vision Permutator中的一种进行实现
- 分析与总结格式不限

实验步骤与实验心得

实验环境配置

> 对于初次使用pytorch环境进行神经网络搭建，配置实验环境显然不是一件容易的事情，对于初次进行pytorch神经网络搭建，这里配置了基于CUDA和GPU的实验环境。

1. 我们首先查看pytorch的官网，发现pytorch的版本已经更新到了2.0.0的版本，对于该版本需要CUDA 11.8版本的驱动。
2. 下载CUDA最新版本的驱动，但是运行时的CUDA toolkit仍然下载使用11.8版本驱动，同时下载对应的cuDNN深度神经网络库。
3. 下载conda管理虚拟环境，使用conda create等命令创建一个pytorch环境，并且我们需要使用python -m ipykernel install --name等命令将pytorch虚拟环境配置到jupyter notebook中，并且使用pycharm连接conda的虚拟环境。
   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：

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

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

得到的结果为：

In [1]:
```
%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)
```

```
Using PyTorch version: 2.0.0  Device: cuda
```

可以看到PyTorch version的版本为2.0.0，Device为cuda。但是我们在实验初始的过程之中可能会遇到如下的warning情况，可能会遇到doesn't match a support version这个warning，可以参考如下解决方案进行解决。

```
[warning解决](https://blog.csdn.net/qq_45365214/article/details/122665571)
```

In [3]:
```
%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)
```

```
C:\Users\25747\AppData\Roaming\Python\Python39\site-packages\requests\__init__.py:102: RequestsDep
hardet (5.1.0)/charset_normalizer (2.0.4) doesn't match a supported version!
  warnings.warn("urllib3 ({}) or chardet ({})/charset_normalizer ({}) doesn't match a supported "

Using PyTorch version: 2.0.0  Device: cuda
```

初始MLP结构理解与运行结果

1. 我们首先对MLP的结构进行进一步的理解，Net部分的代码如下

```
super(Net, self).__init__()
self.fc1 = nn.Linear(28 * 28, 100)
self.fc1_drop = nn.Dropout(0.2)
self.fc3 = nn.Linear(100, 80)
self.fc3_drop = nn.Dropout(0.2)
self.fc4 = nn.Linear(80, 10)
```

其中，super部分为继承的父类的初始化函数，接着我们可以看到对应了两个线性的全连接层，并且为了防止过拟合，在每一层线性全连接层之后连接了一个Dropout层，用于防止模型训练时的过拟合现象。

Dropout层：Dropout可以作为训练深度神经网络的一种trick供选择。在每个训练批次中，通过忽略一半数量的特征检测器（让一半的隐层节点值为0），可以明显地减少过拟合现象。这种方式可以减少特征检测器（隐层节点）间的相互作用，检测器相互作用是指某些检测器依赖其他检测器才能发挥作用。

2. Net网络训练的pytorch框架流程
   1. 将数据和模型放入到GPU之中
   2. optimizer将参数的梯度部分置0
   3. 训练数据传递传过整体网络
   4. 计算损失
   5. 进行梯度下降从损失值开始
   6. optimizer.step进行参数和权重的更新

   具体的训练部分代码如下：

```
# 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
```

3. 初次运行结果
   我们初次使用CPU来对MLP网络进行训练得到的训练时间大致为5min，而如果我们采取GPU进行训练，最终在3min左右的时间就能够完成训练（1660 Ti好像没那么优秀），对于初始给定的Net结构训练得到的结果就就已经很好了，达到了97%的Accuracy。

```
Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.120152

Validation set: Average loss: 0.1007, Accuracy: 9680/10000 (97%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.299256
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.066091
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.170973
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.164638
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.074839
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.073196
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.080212
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.159855
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.209689
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.348697

Validation set: Average loss: 0.0932, Accuracy: 9713/10000 (97%)

CPU times: total: 54.2 s
Wall time: 3min 2s
```

参数以及MLP网络结构优化

1. 随机数种子以及backend算法固定
   因为我们在load训练数据集的时候，设定的shuffle参数为True，也就是每次load训练数据集时load进入的顺序是不相同的，为了使随机选取数据这一可能导致Accuracy结果不同的因素不再影响后续的优化，这里将随机数种子以及cuda的backend算法均进行固定。

```python
# 设置随机种子
# 固定shuffle随机数种子以及cuda等backend算法
# 争取保证每次的运算顺序是同样的
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.cuda.manual_seed(seed) # 为当前GPU设置随机种子；
torch.backends.cudnn.deterministic = True
```

我们将随机可能产生的影响进行了固定然后继续进行参数以及网络结构的优化，但是97%的准确率属实很难优化得到很大的提升了。

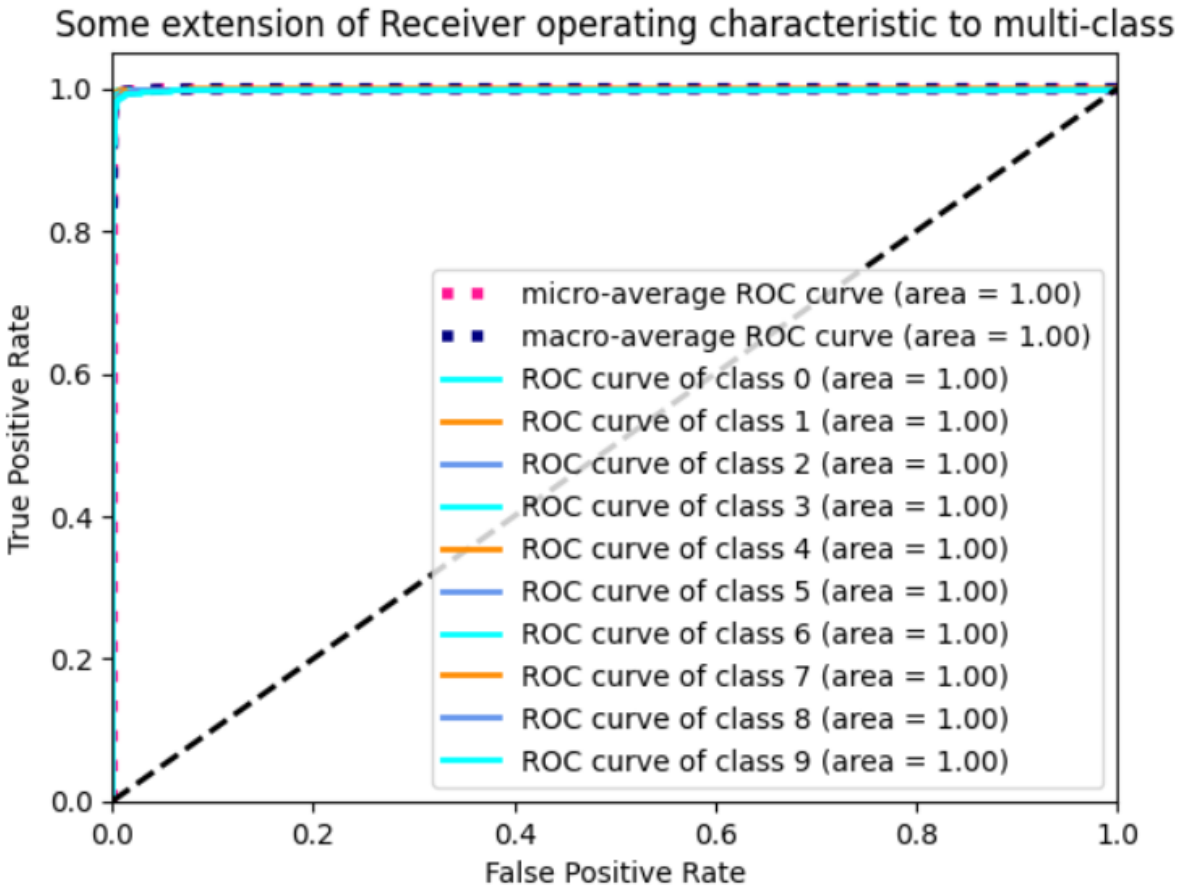
2. ROC-AUC曲线

在给出的初始代码之中，仅展示了每轮训练中的损失值以及准确率，并没有给出很常用的ROC曲线等内容，在给出的初始代码的基础之上，我们还对ROC-AUC曲线进行了补充，相应的实现代码如下：

```
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()
```

因为对于多分类问题，对应的ROC曲线可能会有两种情形，一种即为macro，也就是需要将多分类的每一类的结果进行加和，另一种为micro形式，也就是对每一类的ROC曲线进行统计，这里展示了两种形式的对应的ROC-AUC曲线。



实现结果如上图，可以发现MINST的不同类的分类效果基本相同，都很接近理想点（0，1），都表现出了很好的分类效果。

3. optimizer与loss策略优化

对于网络结构训练之中的 optimizer与loss策略我们可以对不同的策略进行选择测试，对于optimizer优化器的策略我们在优化过程之中，尝试了SGD、RMSprop、Adagrad和Adam的方

法，其中**AdaGrad**的效果是：在参数空间中更为平缓的倾斜方向会取得更大的进步（因为平缓，所以历史梯度平方和较小，对应学习下降的幅度较小），而**RMSProp**的主要思想：使用指数加权移动平均的方法计算累积梯度，以丢弃遥远的梯度历史信息（让距离当前越远的梯度的缩减学习率的权重越小）。**Adam**则是RMSProp和momentum的结合。
我们使用如下代码对不同的优化器进行实验：

```
# SGD随机梯度下降
# 修改尝试不同的优化器
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
# optimizer = torch.optim.RMSprop(model.parameters(), lr=0.01, alpha=0.9)
# optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01, lr_decay=0,
weight_decay=0, initial_accumulator_value=0)
# optimizer = torch.optim.Adam(model.parameters(), lr=0.01, betas=(0.9,0.99))
```

```
Train Epoch: 4 [0/60000 (0%)]   Loss: 0.458005
Train Epoch: 4 [6400/60000 (11%)]       Loss: 0.094558
Train Epoch: 4 [12800/60000 (21%)]      Loss: 0.230140
Train Epoch: 4 [19200/60000 (32%)]      Loss: 0.104135
Train Epoch: 4 [25600/60000 (43%)]      Loss: 0.132421
Train Epoch: 4 [32000/60000 (53%)]      Loss: 0.500127
Train Epoch: 4 [38400/60000 (64%)]      Loss: 0.077827
Train Epoch: 4 [44800/60000 (75%)]      Loss: 0.322782
Train Epoch: 4 [51200/60000 (85%)]      Loss: 0.274475
Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.337614

Validation set: Average loss: 0.2459, Accuracy: 9459/10000 (95%)

Train Epoch: 5 [0/60000 (0%)]   Loss: 0.457947
Train Epoch: 5 [6400/60000 (11%)]       Loss: 0.304821
Train Epoch: 5 [12800/60000 (21%)]      Loss: 0.059048
Train Epoch: 5 [19200/60000 (32%)]      Loss: 0.125563
Train Epoch: 5 [25600/60000 (43%)]      Loss: 0.332683
Train Epoch: 5 [32000/60000 (53%)]      Loss: 0.073692
Train Epoch: 5 [38400/60000 (64%)]      Loss: 0.145045
Train Epoch: 5 [44800/60000 (75%)]      Loss: 0.133096
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.713179
Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.174419

Validation set: Average loss: 0.2249, Accuracy: 9548/10000 (95%)

CPU times: total: 17.7 s
Wall time: 1min 10s
```

但是发现其实对于MINST这种不是很大也不是很复杂的数据集而言，使用更加复杂的优化策略，在分类精度上反而是反向优化了。

接着我们尝试了使用MSE损失函数进行测试，如果想要运用MSE损失函数需要在train和validate过程之中都修改部分代码，来使得tensor的维度可以匹配，具体的修改如下：

```
# 交叉熵损失函数
# 尝试MSE等损失函数
criterion = nn.CrossEntropyLoss()
# criterion = nn.MSELoss()
...
# modified
# loss = criterion(output, target)
temp = torch.max(output, dim=1)
temp = temp.indices.float()
print(temp)
print(target)
loss = criterion(temp, target)
...
# modified
temp = torch.max(output, dim=1)
temp = temp.indices.float()
val_loss += criterion(temp, target).data.item()
```

但是通过学习我们知道其实MSE损失函数并不适用于分类任务，更加适用于回归任务，我们得到的结果也展示了这一点。

```
Train Epoch: 5 [0/60000 (0%)]   Loss: 24.718750
Train Epoch: 5 [6400/60000 (11%)]       Loss: 28.031250
Train Epoch: 5 [12800/60000 (21%)]      Loss: 26.750000
Train Epoch: 5 [19200/60000 (32%)]      Loss: 22.031250
Train Epoch: 5 [25600/60000 (43%)]      Loss: 29.750000
Train Epoch: 5 [32000/60000 (53%)]      Loss: 27.875000
Train Epoch: 5 [38400/60000 (64%)]      Loss: 24.968750
Train Epoch: 5 [44800/60000 (75%)]      Loss: 31.562500
Train Epoch: 5 [51200/60000 (85%)]      Loss: 28.312500
Train Epoch: 5 [57600/60000 (96%)]      Loss: 27.625000

Validation set: Average loss: 28.1265, Accuracy: 979/10000 (10%)

CPU times: total: 17.8 s
Wall time: 1min
```

可以看到经过5轮训练的结果，基本和随机预测的准确率没有差别，所以在分类任务之中使用MSE损失函数的话需要进行一定的改进。

5. 网络参数优化

我们在对 optimizer与loss的策略进行优化过后，接着就是要对网络的参数进行优化，在这一部分的优化之中我们采用的是对于每个因素进行逐步测试的方法，如果有所改进那么就采取优化后的参数结果（如果是全排列测试，太多组合了，很难搞完），首先是对momentum进行参数优化，经过实验测试，当我们使用momentum=0.9时，得到的结果为：

```
Train Epoch: 5 [6400/60000 (11%)]       Loss: 0.118705
Train Epoch: 5 [12800/60000 (21%)]      Loss: 0.273150
Train Epoch: 5 [19200/60000 (32%)]      Loss: 0.071568
Train Epoch: 5 [25600/60000 (43%)]      Loss: 0.036146
Train Epoch: 5 [32000/60000 (53%)]      Loss: 0.190426
Train Epoch: 5 [38400/60000 (64%)]      Loss: 0.104582
Train Epoch: 5 [44800/60000 (75%)]      Loss: 0.124146
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.175408
Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.050754

Validation set: Average loss: 0.0854, Accuracy: 9750/10000 (98%)
```

可以看到同样经历了5轮训练之后得到的准确率为98%，取得了 **1%** 的进步，所以接下来的实验之中momentum参数就选定为0.9，接着我们对dropout层的丢掉参数的概率进行尝试，发现所有的其他的实验测试的效果均不如当dropout=0.2时的准确率，所以采用dropout=0.2。

```
Train Epoch: 5 [0/60000 (0%)]   Loss: 0.109181
Train Epoch: 5 [6400/60000 (11%)]       Loss: 0.036019
Train Epoch: 5 [12800/60000 (21%)]      Loss: 0.038810
Train Epoch: 5 [19200/60000 (32%)]      Loss: 0.035754
Train Epoch: 5 [25600/60000 (43%)]      Loss: 0.088719
Train Epoch: 5 [32000/60000 (53%)]      Loss: 0.132112
Train Epoch: 5 [38400/60000 (64%)]      Loss: 0.038988
Train Epoch: 5 [44800/60000 (75%)]      Loss: 0.057512
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.067557
Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.235873

Validation set: Average loss: 0.0832, Accuracy: 9743/10000 (97%)

CPU times: total: 20.7 s
Wall time: 1min 7s
```

如果采用dropout=0.25，那么测试结果会重新回到97%

6. 网络结构优化

接着我们进行网络结构的优化，在这里我仅仅尝试了加深了网络的深度，并且对于特征的抽取，我模仿了VGG的结构，从28*28的图片在经历了第一层linear层之后转化为14*14的特征，接着的两层linear层继承之前的Net结构。具体的Net结构优化后如下：

```
super(Net, self).__init__()
self.fc1 = nn.Linear(28 * 28, 14 * 14)
self.fc1_drop = nn.Dropout(0.2)
self.fc2 = nn.Linear(14 * 14, 100)
self.fc2_drop = nn.Dropout(0.2)
self.fc3 = nn.Linear(100, 80)
```

```
self.fc3_drop = nn.Dropout(0.2)
self.fc4 = nn.Linear(80, 10)
```

采取了三层线性层后的网络结构，训练的测试结果如下：

```
Train Epoch: 1 [0/60000 (0%)]   Loss: 0.096599
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.042883
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.133841
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.189715
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.073362

Validation set: Average loss: 0.0837, Accuracy: 9732/10000 (97%)

Train Epoch: 2 [0/60000 (0%)]   Loss: 0.140756
Train Epoch: 2 [12800/60000 (21%)]      Loss: 0.025354
Train Epoch: 2 [25600/60000 (43%)]      Loss: 0.085133
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.160063
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.044690

Validation set: Average loss: 0.0768, Accuracy: 9762/10000 (98%)

Train Epoch: 3 [0/60000 (0%)]   Loss: 0.018940
Train Epoch: 3 [12800/60000 (21%)]      Loss: 0.175405
Train Epoch: 3 [25600/60000 (43%)]      Loss: 0.087373
Train Epoch: 3 [38400/60000 (64%)]      Loss: 0.322034
Train Epoch: 3 [51200/60000 (85%)]      Loss: 0.063866

Validation set: Average loss: 0.0680, Accuracy: 9789/10000 (98%)

Train Epoch: 4 [0/60000 (0%)]   Loss: 0.020523
Train Epoch: 4 [12800/60000 (21%)]      Loss: 0.133478
Train Epoch: 4 [25600/60000 (43%)]      Loss: 0.046509
Train Epoch: 4 [38400/60000 (64%)]      Loss: 0.039229
Train Epoch: 4 [51200/60000 (85%)]      Loss: 0.054577

Validation set: Average loss: 0.0719, Accuracy: 9781/10000 (98%)
```

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模型没有区别，均为98%，但是对于更深的网络结构可以在2-3轮训练就可以基本完成模型的训练，收敛的速度更快。至此所有的MLP部分的实验结束，具体的代码可见同目录。

MLP-Mixer学习及实现

1. MLP-Mixer
   2021年谷歌提出的MLP-Mixer在各大数据集上都取得了很好的成绩，它的核心是将token-

mixing的信息和channel-mixing的信息分开提取。其中token是指一系列的线性映射后的patcheschannel-mixing是为了让mlp结构可以在通道之间学习特征。注意这里没有涉及到位置信息的embedding，文中给出MLP的位置信息十分敏感，token-mixing就已经可以满足MLP对不同空域位置的学习。

具体的结构如下（参考论文内容）：

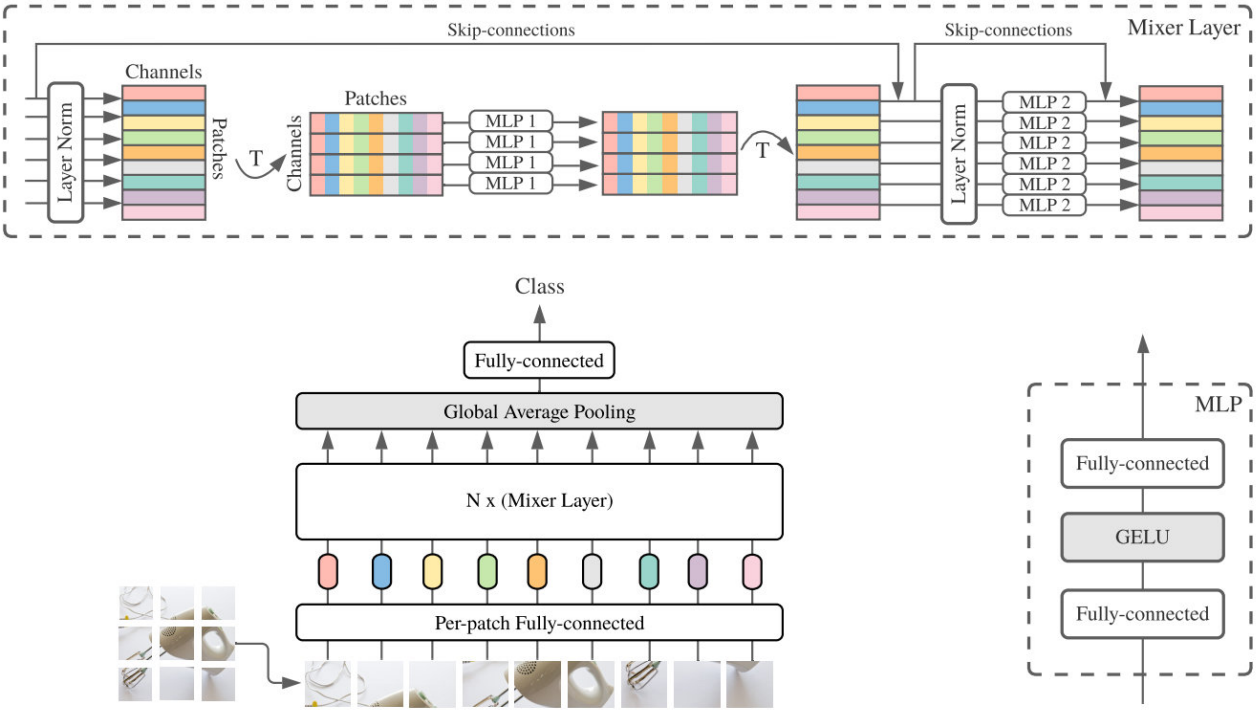


Figure 1: MLP-Mixer consists of per-patch linear embeddings, Mixer layers, and a classifier head. Mixer layers contain one token-mixing MLP and one channel-mixing MLP, each consisting of two fully-connected layers and a GELU nonlinearity. Other components include: skip-connections,

2. MLP-Mixer实现

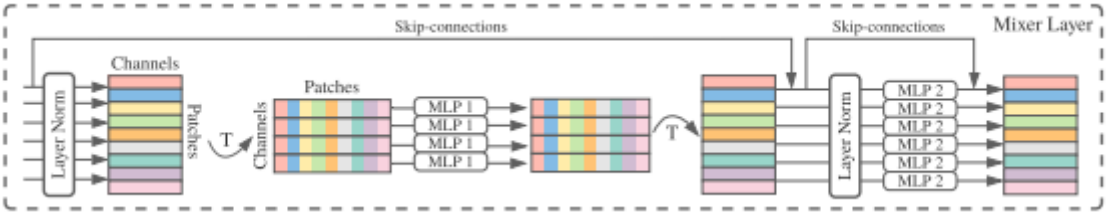
根据上图之中，最核心的网络结构即为MIxer Layer，在这一网络结构之中，将会在token-mixing和channel-mixing之上都进行结合，我们要实现这一结构就需要先实现MLP的结构，这里的MLP结构有点类似于ResNet的通道注意力机制，会先升维再进行降维，具体的MLP结构实现：

```
# 这里有点像残差网络那部分
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
```

可以看到skip connection的结构，接着就是对MLP网络结构进行实现，实现如下：

```
dense(dim, dim * expansion_factor),
nn.GELU(),
nn.Dropout(dropout),
dense(dim * expansion_factor, dim),
nn.Dropout(dropout)
```

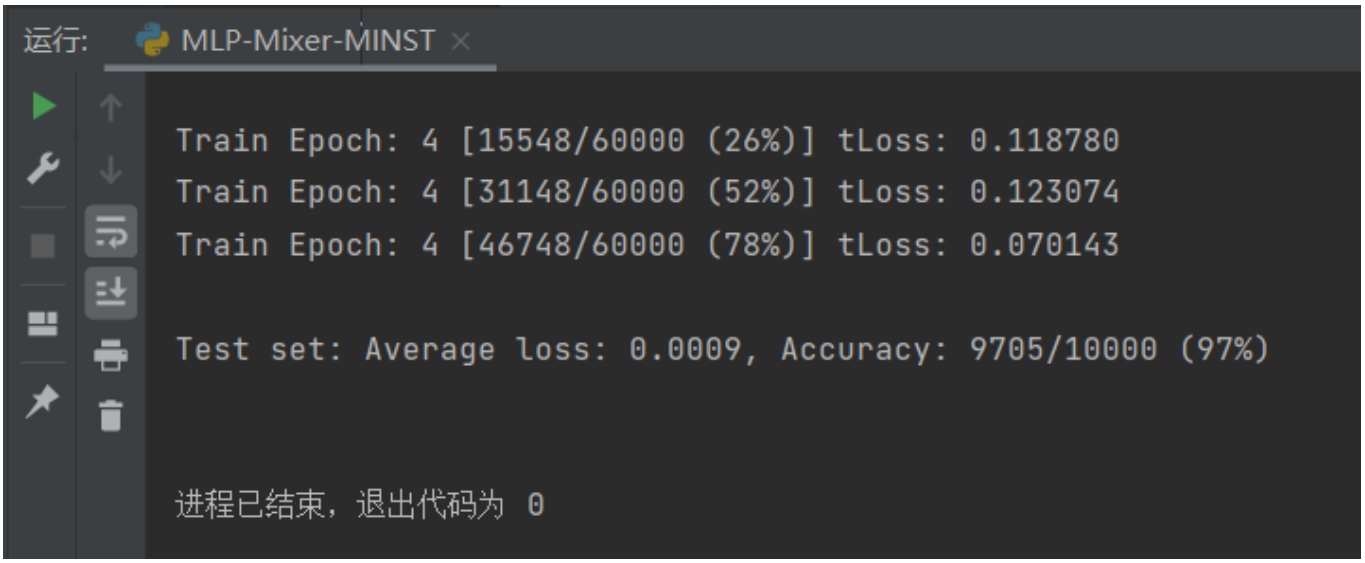
这里使用dense层，如果不进行堆叠的话，dense层和linear层是同样的，这里也可以使用linear层进行替代。
最后就是最重要的，将不同的channel的不同token进行特征结合的部分，流程如下：



在实现网络结构时使用nn.Sequential进行实现，实现代码如下：

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

使用Rearrange方法进行tensor各个维度的重新排列，接着对于PreNormResidual结构进行一定的堆叠，最后通过Global Average和Linear层进行最终的分类，我们在输入时的channel为1，具体的代码见同目录，实现的分类效果如下：



可见在在经历5轮训练之后（epcoh+1），分类的准确率也达到了97%，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（且MLP还经历很多优化）。

实验总结

本次实验分别采用MLP和MLP-Mixer两种深度学习模型，应用于图像分类任务。实验数据集采用MINST，其中包含10个不同的类别，共计70000张28*28的0-9数字图像。我们将数据集分为训练集、验证集，其中训练集用于训练模型，验证集用于调整超参数和选择最佳模型，并进行一部分的分类精度测试。

参考文献

【布客】PyTorch 中文翻译 (apachecn.org)
(117条消息) 机器学习分类性能指标ROC原理及（二分类与多分类）曲线绘制多分类roc潘多拉星系的博客-CSDN博客
机器学习2 -- 优化器（SGD、SGDM、Adagrad、RMSProp、Adam） - 知乎 (zhihu.com)
一文教你彻底理解Google MLP-Mixer（附代码） - 知乎 (zhihu.com)
google MLP-Mixer pytorch框架代码实现 - 知乎 (zhihu.com)