

# 人工智能Lab2 实验报告

雷雨轩 PB18111791 计算机学院

## 传统机器学习

### 数据分析

- train\_num: 3554  
test\_num: 983  
train\_feature's shape:(3554, 8)  
test\_feature's shape:(983, 8)

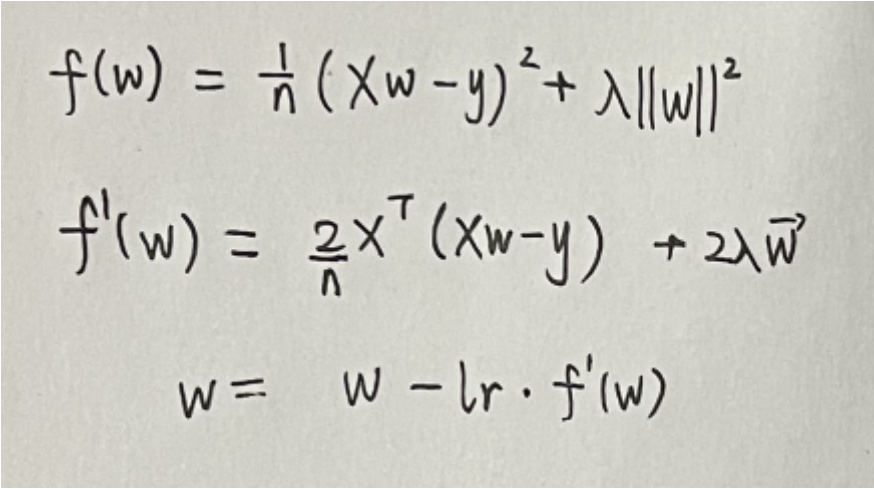
### 线性分类器

#### 实现思路

- 根据实验文档提供的公式，有两种实现方式
  - 岭回归直接求得闭式解

L2正则化约束的最小二乘拟合： $\min \|Y - XW\|_2^2 + \|W\|_2^2$ ，解法  $W = (X^T X + I)^{-1} X^T Y$

- 采用梯度下降法（因为闭式解可能不存在，即矩阵求逆失效时）
- 本次实验采用梯度下降法，对误差函数进行求导后思路如下：


$$\begin{aligned} f(w) &= \frac{1}{n} (Xw - y)^2 + \lambda \|w\|^2 \\ f'(w) &= \frac{2}{n} X^T (Xw - y) + 2\lambda \vec{w} \\ w &= w - lr \cdot f'(w) \end{aligned}$$

- 遇到的问题：实验一开始运行发现会报错，具体打印出gradient和self.W发现随着每个周期的迭代，值越来越大，直到inf。再参考课程群里的讨论后才知道，实验文档里给的误差函数有一定问题，即实际上应该求的是每个数据的平均误差，而不是误差和，因为用参数进行预测时也是对每个数据进行预测。

#### 结果展示与分析

- 在预测输出时，考虑到标签值为1,2,3，所以要对预测值四舍五入取整

- 采用代码默认参数，并在[0,1)上随机初始化W参数，得到如下结果

```
(ustc-ai) D:\科大\大三下\人工智能基础\Lab2\src1>python linearclassification.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6174974567650051
0.6378378378378378
0.5986842105263158
0.6325036603221084
macro-F1: 0.6230085695620873
micro-F1: 0.6178117048346056
```

由结果可以看到，线性分类器的效果比随机预测好一些，但仍存在很大局限性，仅0.6的准确率

## 贝叶斯

### 实现思路

- 即参考实验文档，在fit函数里依次计算每个label在每个属性上的条件概率，其中对连续属性采用方法二计算

在pred函数里，则是对测试集中每个数据，把每个属性的条件概率乘起来，这里注意为防止避免相乘为0，全部对概率取了自然对数，然后再相加。最后我们选择求和概率最大的那个label作为预测值

此外，对连续属性的取值，根据fit的算出的高斯分布的均值和标准差来计算即可，公式如下：

$$p(x_i|c) = \frac{1}{\sqrt{2\pi} \sigma_{c,i}} \exp\left(-\frac{(x_i - \mu_{c,i})^2}{2\sigma_{c,i}^2}\right)$$

- fit函数里的关键代码

```
for i, type_attr in enumerate(featuretype):
    if type_attr==0: #离散型
        self.Pxc[i]={} #第i个属性的条件概率
        for label in range(1,4):
            self.Pxc[i][label]={}
            D_c = len(trainlabel[trainlabel==label])
            data_index=np.where(trainlabel==label)
            temp=traindata[:,i].reshape(-1,1)[data_index] #选择标签值为label的数
            # 数据的对应属性那一列
            for value in range(1,4):
                self.Pxc[i][label][value]=
                (len(temp[temp==value])+1)/(D_c+3)

        elif type_attr==1: #连续型
            self.Pxc[i]={} #第i个属性的条件概率
            for label in range(1,4):
                self.Pxc[i][label]=[] #存放 标签为label的样本在 第i个属性上取值的均值和
                # 方差
                data_index=np.where(trainlabel==label)
                data=traindata[:,i].reshape(-1,1)[data_index] #选择标签值为label的数
                # 数据的对应属性那一列
                #print("data: ", data.shape)
                mean=np.sum(data,axis=0)/(data.shape[0])
```

```
std=np.std(data, axis=0)
self.Pxc[i][label].append(mean)
self.Pxc[i][label].append(std)
```

这里的难点在于对numpy数组按条件选取行、列的操作，并且涉及到的条件分别在traindata和trainlabel两个数据里，经过检索，用np.where等相关函数完成了实现

## 结果展示与分析

- ```
(ustc-ai) D:\科大\大三下\人工智能基础\Lab2\src1>python nBayesClassifier.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6134282807731435
0.7137404580152672
0.4725111441307578
0.6684005201560468
macro-F1: 0.6182173741006906
micro-F1: 0.6134282807731435
```

由结果可以看到，贝叶斯分类器的效果比随机预测好一些，跟线性分类器结果差不多。

## SVM

### 实现思路

- 实现思路参考西瓜书，即对支持软间隔和核函数的SVM分类器，西瓜书上给出了其对应的凸二次规划问题的形式（对偶形式），再参考CVXOPT模块的教程，即其提供的求解接口可以求解标准形式的二次规划问题。于是乎，通过对SVM对偶问题进行变式，可以分离得到标准形式的二次规划问题，数学推导如下：

标准形式:  $\min \frac{1}{2} x^T P x + q^T x$   
 s.t.  $Gx \leq h$   
 $Ax = b$

对偶问题:  $\min_{\alpha} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j K(x_i, x_j) - \sum_{i=1}^m \alpha_i$   
 or  $\min_{\alpha} \frac{1}{2} \alpha^T \begin{pmatrix} & \\ & \end{pmatrix} \alpha + \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}^T \alpha$   
 $\downarrow$   $\downarrow$   
 $1 \times m$   $m \times 1$   
 记为  $P (m \times m)$   
 $P_{ij} = y_i y_j K(x_i, x_j)$   
 $m \times m$

s.t.  $\sum_{i=1}^m \alpha_i y_i = 0$  or  $\begin{cases} y^T \alpha = 0 \\ \begin{pmatrix} -1 & -1 & -1 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} \leq \begin{pmatrix} C \\ C \\ C \end{pmatrix} \end{cases}$

1101—145 2014. 12. 2500

- 代入CVXOPT接口求解出 $\alpha$ 后, 需要找到对应的支持向量 ( $0 < \alpha < C$ 的数据项), 然后可求解出b的值

注意因为是用的核函数, 所以不能也不需要单独求解W的值

由左边计算出  $\alpha$  后

$$W = \sum_{n=1}^m \alpha_n y_n \underline{x_n} \rightarrow d_{n1}$$

$b$  则选一个  $0 < \alpha_i < C$  来算.

$$f(x) = \sum_{i=1}^M \alpha_i y_i \kappa(x, x_i) + b$$

#### 结果展示与分析

采用默认参数运行代码

- 线性核

```

Optimal solution found.
self.alpha: (3554, 1)
w shape: (3554, 1)
P: 3554 , 3554
      pcost      dcost      gap      pres      dres
0: -2.2283e+03 -1.0144e+04 5e+04 3e+00 4e-13
1: -1.5021e+03 -7.1327e+03 8e+03 2e-01 4e-13
2: -1.5747e+03 -2.6575e+03 1e+03 3e-02 3e-13
3: -1.7590e+03 -2.2104e+03 5e+02 1e-02 3e-13
4: -1.8490e+03 -2.0498e+03 2e+02 3e-03 4e-13
5: -1.8550e+03 -2.0397e+03 2e+02 2e-03 4e-13
6: -1.8649e+03 -2.0232e+03 2e+02 2e-03 3e-13
7: -1.9015e+03 -1.9629e+03 6e+01 4e-04 4e-13
8: -1.9107e+03 -1.9486e+03 4e+01 1e-04 4e-13
9: -1.9125e+03 -1.9453e+03 3e+01 8e-05 4e-13
10: -1.9211e+03 -1.9341e+03 1e+01 1e-05 4e-13
11: -1.9252e+03 -1.9293e+03 4e+00 3e-06 4e-13
12: -1.9267e+03 -1.9276e+03 9e-01 4e-07 4e-13
13: -1.9271e+03 -1.9272e+03 9e-02 4e-08 4e-13
14: -1.9271e+03 -1.9271e+03 4e-03 2e-09 4e-13
15: -1.9271e+03 -1.9271e+03 4e-05 2e-11 4e-13
Optimal solution found.
self.alpha: (3554, 1)
w shape: (3554, 1)
Acc: 0.602238046795524
0.6967509025270758
0.21973094170403587
0.7246376811594203
macro-F1: 0.547039841796844
micro-F1: 0.602238046795524

```

- 多项式核

```

Optimal solution found.
Acc: 0.6490335707019329
0.750551876379691
0.5822784810126582
0.6583679114799447
macro-F1: 0.6637327562907647
micro-F1: 0.6490335707019329

```

- 高斯核



```
Optimal solution found.
Acc: 0.6561546286876907
0.7539503386004515
0.5740498034076016
0.6815789473684212
macro-F1: 0.6698596964588247
micro-F1: 0.6561546286876907
```

- 由上述三种不同的核函数所得结果可以发现，高斯核效果最好，多项式核次之，线性核效果最差，线性核与前面的线性分类器、朴素贝叶斯结果差不多。这说明本次实验的数据，并不是线性可分的，而当利用核函数投影到高维空间后，可能使得划分平面更好找到，所以SVM预测结果准确度有所提升。

## 深度学习

### MLP\_manual

#### 实现思路

- 实现思路比较清晰，主要还是看懂实验文档里的公式，然后通过torch的矩阵运算进行实现即可。
- 首先是要确定数据的形式：我考虑epoch=20, 一个epoch里对100个(5,1)的数据进行训练，并且对每个数据都要进行一次梯度下降更新。

接着是实现sigmoid、softmax、CrossEntropy函数的实现，主要还是要尝试，验证自己所写、所定的维度是否一致，数据计算是否正确

```
def sigmoid(x):
    return 1.0/(1.0+torch.exp(-x))

def softmax(x):
    c=1.0/torch.sum(torch.exp(x),dim=0)
    return c.view(-1,1) * torch.exp(x)

def CrossEntropy(label,pred):#label: [1], pred:[3,1]
    #return torch.sum(-torch.log(torch.gather(pred,1,label).squeeze(1)))/100
    return -torch.log(pred[label[0]])
```

- 然后就是forward、BP、梯度下降的实现,这里也同时求了自动梯度以便于与手动求导结果比较，自动梯度求解使用autograd包

```
#forward
x=inputs[i].view(-1,1) #(5,1)
h1=sigmoid(torch.mm(w1,x))#(4,1)
h2=sigmoid(torch.mm(w2,h1))#(4,1)
y=softmax(torch.mm(w3,h2))#(3,1)
loss=CrossEntropy(labels[i],y)#[1]

#自动梯度求导
w3_grad = autograd.grad(loss, w3, retain_graph=True)
w2_grad = autograd.grad(loss, w2, retain_graph=True)
w1_grad = autograd.grad(loss, w1, retain_graph=True)

#手动BP
```

```

l_s3=y.clone() #(3,1)
for j in range(3):
    if j==labels[i][0]:
        l_s3[j][0]=l_s3[j][0]-1
    L_w3=torch.mm(l_s3,h2.T) #(3,4)
    w3_l_s3_s2 = torch.mm(w3.T,l_s3) * h2 * (1-h2) #(4,1)
    L_w2 = torch.mm(w3_l_s3_s2,h1.T) #(4,4)
    w2_w3_l_s3_s2_s1 = torch.mm(w2.T,w3_l_s3_s2) * h1 * (1-h1) #(4,1)
    L_w1 = torch.mm(w2_w3_l_s3_s2_s1, x.T) #(4,5)

#梯度下降
w1=w1-lr*L_w1
w2=w2-lr*L_w2
w3=w3-lr*L_w3

```

- loss的计算是计算一个epoch里所有数据loss的均值

## 结果展示与分析

- 梯度计算正确性验证：如下图，迭代完成后的梯度情况，Wi\_grad是自动梯度，L\_Wi是手动梯度，对比可知结果一致（其余轮数里经过检查也是一致的）

```

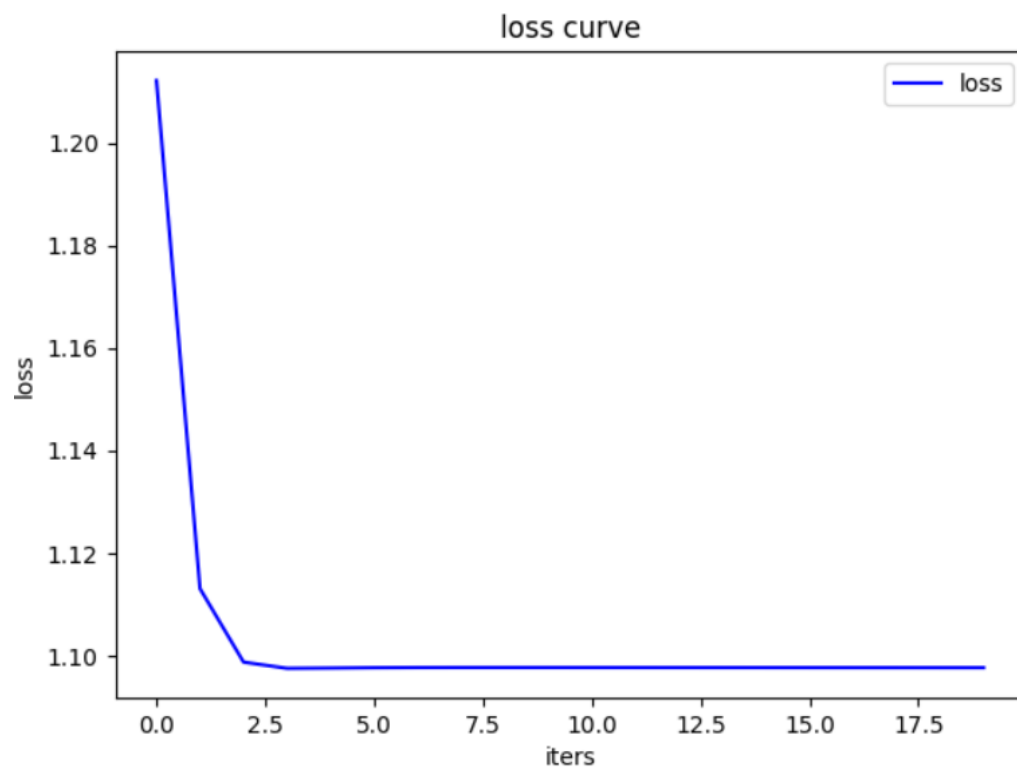
W1_grad: (tensor([[ -0.0009, -0.0002, -0.0002, -0.0002, -0.0008],
                  [ 0.0047, 0.0011, 0.0010, 0.0012, 0.0039],
                  [ 0.0018, 0.0004, 0.0004, 0.0005, 0.0015],
                  [-0.0007, -0.0002, -0.0002, -0.0002, -0.0006]]),)
W2_grad: (tensor([[ -0.0067, -0.0066, -0.0071, -0.0068],
                  [-0.0072, -0.0071, -0.0076, -0.0073],
                  [-0.0068, -0.0067, -0.0072, -0.0069],
                  [ 0.0271, 0.0268, 0.0288, 0.0275]]),)
W3_grad: (tensor([[ 0.3177, 0.3184, 0.3152, 0.3260],
                  [ 0.2442, 0.2448, 0.2423, 0.2506],
                  [-0.5619, -0.5632, -0.5576, -0.5765]]),)
L_W1: tensor([[ -0.0009, -0.0002, -0.0002, -0.0002, -0.0008],
              [ 0.0047, 0.0011, 0.0010, 0.0012, 0.0039],
              [ 0.0018, 0.0004, 0.0004, 0.0005, 0.0015],
              [-0.0007, -0.0002, -0.0002, -0.0002, -0.0006]], grad_fn=<MmBackward>)
L_W2: tensor([[ -0.0067, -0.0066, -0.0071, -0.0068],
              [-0.0072, -0.0071, -0.0076, -0.0073],
              [-0.0068, -0.0067, -0.0072, -0.0069],
              [ 0.0271, 0.0268, 0.0288, 0.0275]], grad_fn=<MmBackward>)
L_W3: tensor([[ 0.3177, 0.3184, 0.3152, 0.3260],
              [ 0.2442, 0.2448, 0.2423, 0.2506],
              [-0.5619, -0.5632, -0.5576, -0.5765]], grad_fn=<MmBackward>)

```

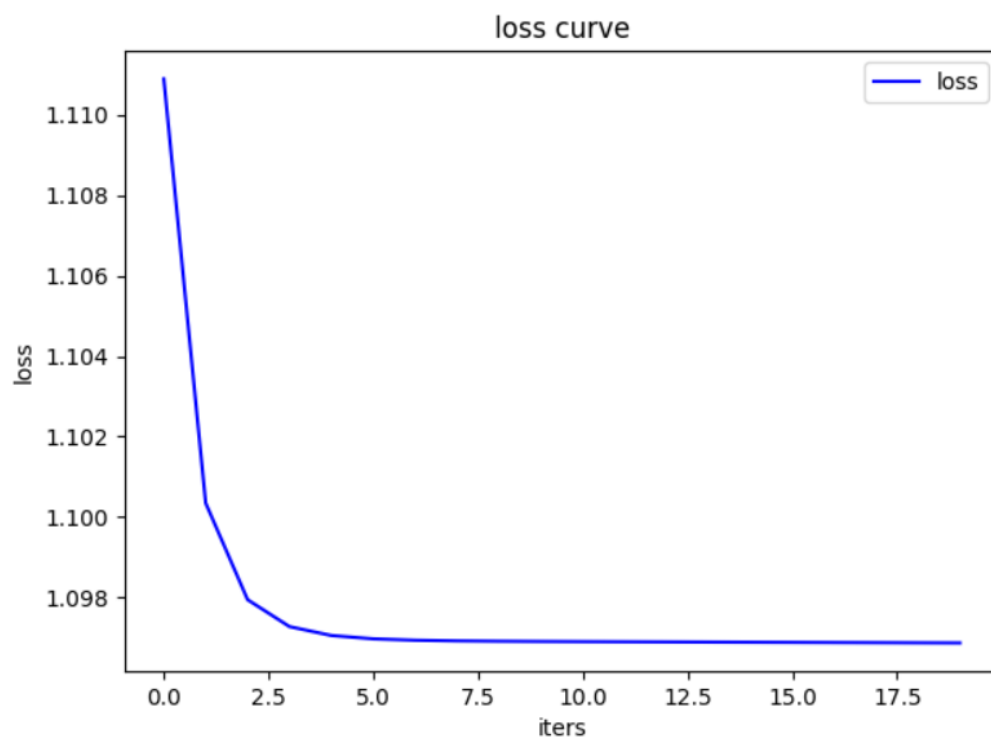
- loss曲线：对20个epoch的loss作图



- 使用手动梯度：

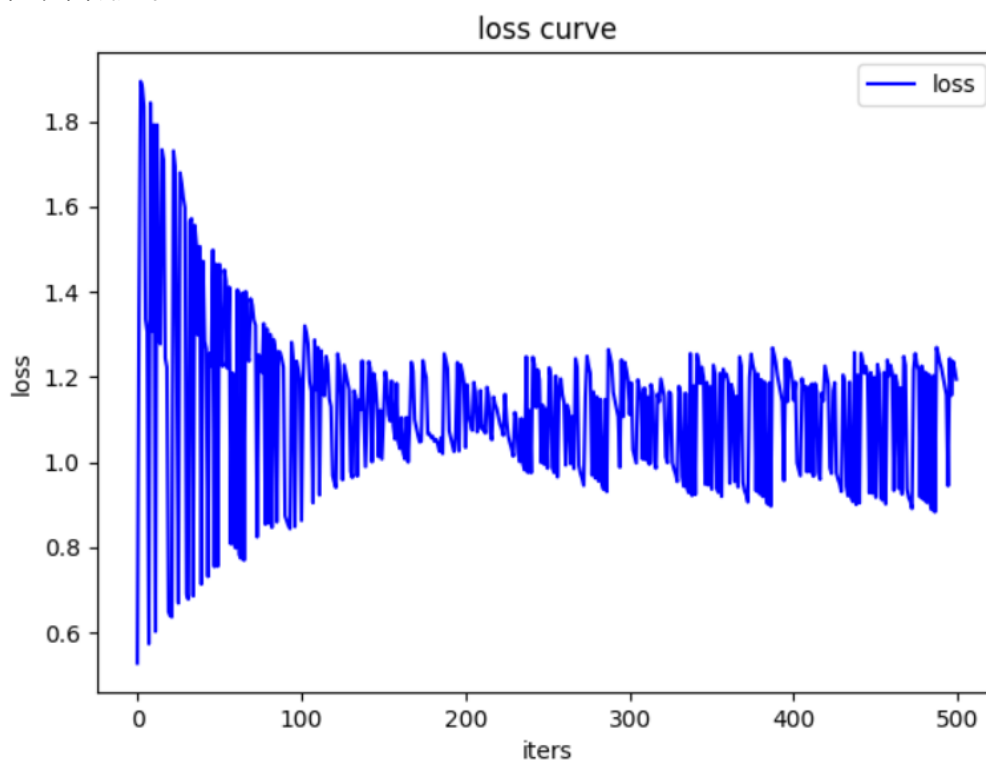


- 使用自动梯度：



对比可知结果一致（存在随机初始化数据的误差）

- 此外，还有需要注意的一点时，一开始我loss曲线画图的时候，是把每个数据作为数据点来作图的，曲线如下

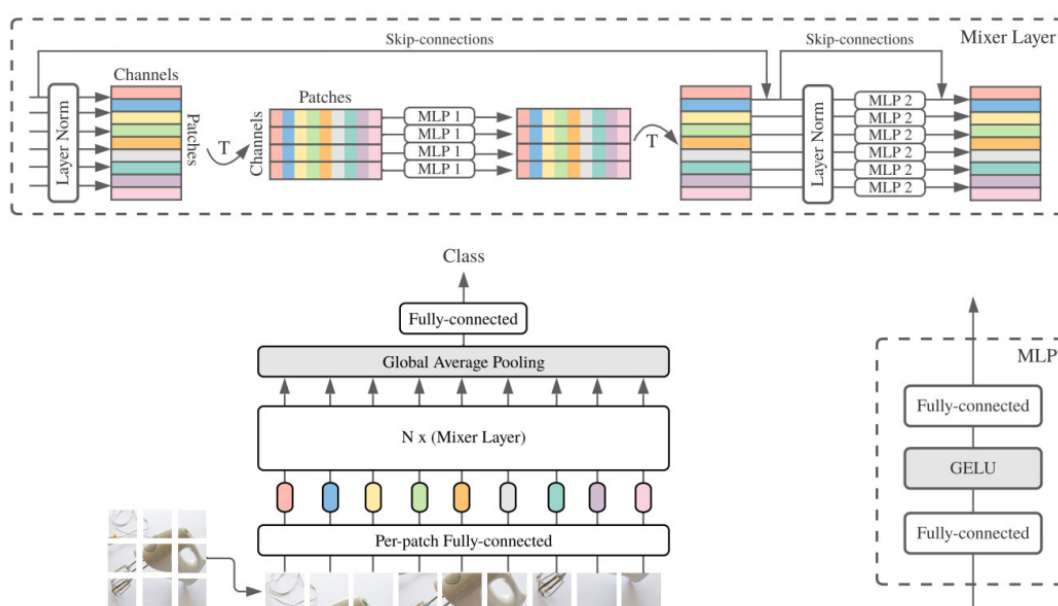


由图像可知，最后每个数据的loss也是会收敛到1.09左右，但与前述loss曲线差别很大：首先这与随机生成数据属性和标签有关系，其次，曲线震荡的原因是因为相邻的loss点是对不同数据来计算的，肯定会有偏差，因为我们梯度下降的目的是要最小化整个数据集的loss，而非单个数据的loss，所以可能模型参数的变化会使得某些数据的loss反而更大了。而若对每个epoch所有数据loss求均值，就可以发现loss是在随着迭代轮数增加而下降的。

## MLP\_Mixer

### 实现思路

- 首先阅读论文，理清楚模型结构



再对照代码已经给定的模块，即两部分，Mixer\_Layer和其余(Pre-patch Fully-connected, Global average pooling, fully connected)

自己写这类代码的经验就是，先弄清楚结构，然后从模型输入开始，一个module一个module的构建，并且给出每个module输出的维度，这样便于对照检查模型是否写对

- 需要特别注意的地方

- Per-patch Fully-connected：这里我们的输入是一张28\*28维的图片，patch的概念指的是将此图片分割为多个不重叠的部分，然后对每个部分进行线性映射为一个channels维的值，然后拼接起来，得到一个channels \* Patches的数据。具体实现上，其实可以利用卷积操作加矩阵转置、view等操作结合来实现

代码如下

```
#def __init__()
self.per_patch =
nn.Conv2d(1,hidden_dim,kernel_size=patch_size,stride=patch_size)#在
forward阶段还需要把patch铺平

#def forward()
y=self.per_patch(data) #
(batch_size,hidden_dim,28/patch_size,28/patch_size)
bs,c,h,w=y.shape
y=y.view(bs,c,-1).transpose(1,2) #(batch_size,patches,hidden_dim)
```

- 此外，对于Mixer\_Layer，需要注意矩阵运算的维度问题，存在两次转置操作

自己在具体实现时遇到的一个问题时：数据在cpu和gpu间的矛盾

一开始打算使用下面代码来实现多个Mixer Layer层，但是在具体运行时发现Cuda错误，即发现self.mlp\_blocks仍在cpu上，与数据、模型其他参数在gpu上产生矛盾。

```
self.mlp_blocks=[]
for i in range(depth):
    self.mlp_blocks.append(Mixer_Layer(self.patch_size,self.hidden_dim))
```

分析后发现是因为这里直接用了list，而list是没法直接放入gpu的，再尝试多种方法（诸如在cpu和gpu间切换数据，但这行不通）后，发现可以通过nn.Sequential以及add\_module操作来完成不确定数目的模块的添加，如下所示

```
self.mlp_blocks=nn.Sequential()
for i in range(depth):
    self.mlp_blocks.add_module('{0}th Mixer_Layer'.format(i),
                                Mixer_Layer(self.patch_size,self.hidden_dim))
```

- 此外，train和test函数则比较样板代码，不再赘述，而使用的损失函数和优化器如下

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=learning_rate)
```

## 结果展示与分析

- 多次调参后，取最好的结果如下，参数为patch\_size = 7, hidden\_dim = 30, depth = 10

```
Train Epoch: 2/5 [25600/60000] Loss: 0.196138
Train Epoch: 2/5 [38400/60000] Loss: 0.106556
Train Epoch: 2/5 [51200/60000] Loss: 0.098044
Train Epoch: 3/5 [0/60000] Loss: 0.101129
Train Epoch: 3/5 [12800/60000] Loss: 0.167981
Train Epoch: 3/5 [25600/60000] Loss: 0.099234
Train Epoch: 3/5 [38400/60000] Loss: 0.065266
Train Epoch: 3/5 [51200/60000] Loss: 0.061182
Train Epoch: 4/5 [0/60000] Loss: 0.093825
Train Epoch: 4/5 [12800/60000] Loss: 0.097859
Train Epoch: 4/5 [25600/60000] Loss: 0.077084
Train Epoch: 4/5 [38400/60000] Loss: 0.118652
Train Epoch: 4/5 [51200/60000] Loss: 0.047264
Test set: Average loss: 0.1033 Acc 0.96
```

- 调参过程中也发现，随着hidden\_dim 以及 depth的增加，Acc基本成上升趋势，loss成下降趋势，这说明了
  - hidden\_dim增加，数据能更充分的从各个方面学习，即数据的表示更加详细（具体则是有更多卷积核来抽取图像特征）
  - depth增加，即神经网络深度增加，能提升模型效果，这也是深度学习的本质所在

## 参考文档

- [https://blog.csdn.net/weixin\\_46649052/article/details/112215146](https://blog.csdn.net/weixin_46649052/article/details/112215146)
- <https://blog.csdn.net/u013164528/article/details/45042895>
- <https://blog.csdn.net/u014636245/article/details/102574938>
- [https://blog.csdn.net/Ocean\\_waver/article/details/104825064](https://blog.csdn.net/Ocean_waver/article/details/104825064)
- <https://www.cnblogs.com/redo19990701/p/11565361.html>
- <https://www.jianshu.com/p/df447c3e4efe>
- <https://blog.csdn.net/xholes/article/details/78461164>
- <https://blog.csdn.net/PanYHHH/article/details/113436204>