人工智能第二次实验实验报告

舒文炫

2021 年 7 月 15 日

目录

Chapter 1

实验介绍

1.1 实验内容

本次实验主要分为两个部分，第一部分为传统机器学习，第二部分为深度学习。

- 传统机器学习，包括实现线性分类器，多分类朴素贝叶斯以及支持向量机，使用的数据为鲍鱼数据集，助教已经做完了数据预处理，类别为三种，需要用这三种方法对数据分类
- 深度学习，分为实现多层感知机和复现 MLP_mixer，多层感知机数据为随机生成，主要考察实现前向传播，自动求导，反向传播，梯度下降。复现 MLP_mixer 主要考察 pytorch 各种包的使用以及阅读文献的能力。

1.2 实验环境

- 虚拟环境 Python3.6, 不过实测在我原环境 Python3.7 也能运行 (这主要是后面忘记在虚拟环境下运行，但是发现的时候并没有问题)
- Pytorch 版本:torch 1.9.0 torchvision 0.10.0

Chapter 2

实验设计

2.1 part1: 传统机器学习

线性分类器是考虑用线性模型

$$Y = Xw + b + e$$

拟合数据得出结果，这里 X 为数据，Y 为数据类别标签,w,b 为需要训练的参数,e 是误差项，一般我们认为其为正态分布

误差函数为

$$L = \frac{1}{2n}(Xw - Y)^2 + \frac{1}{2}\lambda w^2$$

n 为数据个数,λ 为惩罚项，这一项的添加是防止出现共线性的情况，导致模型预测不准，如果使用回归分析里面岭回归的知识，可以直接求得闭式解. 在这里，我们运用梯度下降法求解，默认参数学习率 lr=0.05，迭代 1000 次，$\lambda = 0.001$，在这个误差函数下，梯度为

$$dL = \frac{1}{n}(X^T Xw - X^T Y) + \lambda w$$

梯度下降法，每次向负梯度方向更新 w 为 $w - lr * dL$, 在学习率取的适当的情况下，可以保证损失函数递减，损失函数越小，这个模型在训练集上表现越好。预测则使用训练出的模型在测试集上判断类别，这里要分类，考虑分到每一个类别是的损失，我们取损失最小类别的为预测结果

朴素贝叶斯考虑用概率模型训练数据，主要用到贝叶斯公式，最简单的情况是

$$P(A|B) = \frac{P(B|A)P(A)}{P(B|A)P(A) + P(B|A^C)P(A^C)}$$

, 这个公式使得我们在得到每个类别的出现概率，已经在该类别下，每个特征出现的概率之后，可以预测出已知这些特征的情况后，该数据类别是哪一类的概率。故我们的训练，即通过训练数据，给出训练数据的每个类别的概率，由于我们假设各个特征是在给的类别条件独立的故我们只要得到每个特征在每一类下的概率即可。同时考虑到泛化性能，对离散数据使用了拉普拉斯平滑，这个保证了，即使在测试集该特征出现了训练集没有的情况，也能处理。对连续数据，我假设其为正态分布，使用极大似然估计，求出该分布期望和方差的估计，用这个作为连续数据的分布。期望的极大似然估计为样本均值，方差的极大似然估计为 $\frac{n-1}{n}Var(x)$, 不过这不是无偏估计，一般在大样本下我们直接用样本方差来估计分布方差。最后的概率即将这些乘起来，但是考虑到精度问题，概率一般很小，乘起来结果在 python 里面可能为 0. 我们对每一项取了对数，再相加，这是考虑到有 $\log(xy) = \log(x) + \log(y)$. 预测即用我们得出的条件概率计算，概率最高值对应类别为预测类别。

SVM 是考虑用一个分离超平面来分类数据，如果数据线性可分，SVM 的准确率可以很高，可以达到百分之百，这里数据不一定会线性可分，我们要实现的是有软间隔的，即允许一些数据跨过分隔超平面。原型的 SVM 往往在实现时比较困难，这里我们采用其对偶形式，并使用核方法，实现了线性核，高斯核以及多项式核。考虑到会有精度问题，对很小的参数，我们直接置为 0，即不将它对应的数据视为支持向量。对偶形式如下：

$$min\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(x_i,x_j)+\sum_{i=1}^{N}\alpha_i$$

$$s.t.\sum_{i=1}^{N}\alpha_i y_i=0$$

$$0\le\alpha_i\le C$$

$$i=1,2,3,...,N$$

其中 $K(x_i,x_j)$ 为第 i 个和第 j 个数据算出的核，我们会得到一个 N*N 的核矩阵，α_i 为我们需要求的参数，C 为惩罚参数，C 越大，对误分类的惩罚越大。由上形式可知，此为一个关于 α 的约束二次规划问题。

2.2 part2: 深度学习

深度学习这一块本身没有太多理论支撑，不过却有很好的结果，我觉得这很值得去探索。

MLP 即多层感知机，是最简单的神经网络形式，分为输入层，隐含层，输出层，这里我们要实现一个四层的感知机，每层节点为 5，4，4，3，其中 5 为输入层，两个隐含层节点个数为 4，一个输出层节点个数为 3. 层与层直接采用全连接的形式，即通过一个线性变换，然后隐含层之间使用 sigmoid 激活函数，这样才能拟合数据非线性的情形，如果没有这个激活函数，无论使用多少层，所能拟合的也只有线性。这里我还准备加入额外的偏倚项，实验文档里面没有提到这个，感觉加入偏倚项可以使模型更鲁棒。隐含层与输出层之间使用 softmax 函数连接。这个函数可以将输出的结果转为概率分布，即输出的三个数和为 1. 这样方便使用交叉熵损失函数。然后还要实现反向传播，需要自己手动实现自动求导与 torch 的自动求导进行比较，用矩阵运算实现求导的公式在实验文档里面有描述，我在此不多赘述，比较容易实现。最后输出 loss 曲线，看是否实现了梯度下降，这里需要调用 python 画图的包，将每次迭代的 loss 储存起来，最后绘制。

MLP_mixer 是谷歌提出的一套框架，只使用最简单的 MLP 就能是模型达到特别高的精度，阅读文献后总结出，要复现这个模型，最重要的构建 Mixer Layer. 整个的大体框架为，先将图片拆成多个 patch，然后用一个全连接网络对所有 patch 处理，提取出 tokens，之后就要用到 mixer layer 层，(这里深度可以自行设计, 要求是大于 1) 将特征信息不断提炼，最后通过一个全连接层输出结果进行预测。mixer 分为 token-mixer 以及 channel-mixer，简单来说就是分别对输入特征平面进行（1）沿列方向的特征提炼，（2）沿行方向的特征提炼。对 MLP 全连接层之间使用 GELU 激活函数。这一部分可以调用所有 torch 的包，实现起来代码比较精炼，主要是对框架的认识。

Chapter 3

实验实现

下面我将展示我的代码来解释具体实现：

3.1 part1: 传统机器学习

3.1.1 线性分类器

```
14      def __init__(self,lr=0.05,Lambda= 0.001,epochs = 1000):
15          self.lr=lr
16          self.Lambda=Lambda
17          self.epochs =epochs
18          self.w=np.array([0 for i in range(8)])
19
20      '''根据训练数据train_features,train_labels计算梯度更新参数w'''
21      def fit(self,train_features,train_labels):
22          self.w=self.w.reshape(-1,1)
23          qwq=np.dot(train_features.T,train_features)
24          qaq=np.dot(train_features.T,train_labels)
25          for i in range(0,self.epochs):
26              df=(np.dot(qwq,self.w)-qaq)/len(train_labels)+self.Lambda*self.w
27              self.w=self.w-self.lr*df
28              '''print(self.w.reshape(1,-1))
29              print(np.linalg.norm(np.dot(train_features,self.w)-train_labels)+self.Lambda*np.linalg.norm(self.w))'''
30
```

图 3.1: linearclassification fit

这里创建一个 LinearClassfication 类，init 方法里面助教并没有给出参数 w 的初始化，这在 fit 训练 w 时将结果传给 predict 有点麻烦，故我在此添加了 w 的初始化. 这对于这个类也是合理的，因为我们的模型就是要得到 w，需要将其保存起来。在 fit 方法里面，为防止重复计算，我将 X^TX, X^TY 的值分别用 qwq 与 qaq 保存了起来，后面可以直接使用，第 25 到 27 行用 for 循环实现了梯度下降，迭代 self.epochs 次。梯度如前面实验设计里面所示。

```
33
34      '''根据训练好的参数对测试数据test_features进行预测，返回预测结果
35      预测结果的数据类型应为np数组，shape=(test_num,1) test_num为测试数据的数目'''
36      def predict(self,test_features):
37          result=np.dot(test_features,self.w)
38          pred=np.array([0 for i in range(len(result[:,0]))])
39          for i in range(len(result[:,0])):
40              l1=(result[i,0]-1)*(result[i,0]-1)
41              l2=(result[i,0]-2)*(result[i,0]-2)
42              l3=(result[i,0]-3)*(result[i,0]-3)
43              if(l1<l2):
44                  if(l1<l3):
45                      pred[i]=1
46                  else:
47                      pred[i]=3
48              else:
49                  if(l2<l3):
50                      pred[i]=2
51                  else:
52                      pred[i]=3
53          return(pred.reshape(-1,1))
54
```

图 3.2: linearclassification predict

在这个 predict 方法里面, 第 37 行用训练到的 w 在测试机算出每个数据的结果，但这个结果显然不会是 1，2，3 这样的离散值，故我在后面使用 loss 函数判断，如果分到某一类的 loss 函数最小，我就将其分到这一类，但是考虑到 loss 函数中有很多相等的常量，比如惩罚项，在给定了 w 后就都一样了。最重要的项就是计算预测结果到类别的距离这一项，故我只保留了该项，简化计算。39 到 52 行用循环，一个一个分类，分类主要写了一个判断大小模块，l1,l2,l3 分别表示到 1，2，3 的距离，选择距离最小的分到该类。

3.1.2 多分类朴素贝叶斯

```
20      def fit(self,traindata,trainlabel,featuretype):
21          resultc=Counter(trainlabel.flatten())
22          self.Pc[1]=(resultc[1]+1)/(len(trainlabel)+3)
23          self.Pc[2]=(resultc[1]+1)/(len(trainlabel)+3)
24          self.Pc[3]=(resultc[1]+1)/(len(trainlabel)+3)
25          for i in range(8):
26              if(featuretype[i]==0):
27                  qwq=Counter(traindata[:,i].flatten())
28                  q1={}
29                  for key1 in self.Pc:
30                      p1={}
31                      for key2 in qwq:
32                          p1[key2]=(sum([1 for x in range(len(trainlabel)) if trainlabel[x]==key1 and traindata[x,i]==key2])+
33                      q1[key1]=p1
34                  self.Pxc[i]=q1
35              if(featuretype[i]==1):
36                  q1={}
37                  for key1 in self.Pc:
38                      p1={}
39                      p1["mean"]=np.sum([traindata[x,i] for x in range(len(traindata[:,i])) if trainlabel[x]==key1])/np.sum([
40                      p1["var"]=np.sum([pow(traindata[x,i]-p1["mean"],2) for x in range(len(traindata[:,i]))])/np.sum([1 for
41                      q1[key1]=p1
42                  self.Pxc[i]=q1
43
```

图 3.3: nbayes fit

这里某些行代码写的比较长，截不全，完整的可以直接看源代码。这个 fit 方法里面，我们先用 counter 方法，统计每个类别的个数，保存的数据类型类似字典，然后计算每个类别出现的概率，存到 Pc 字典里面。

后面对每个 feature，通过 featuretype 来区分时离散型还是连续型，离散型 featuretype=0，此时使用拉普拉斯光滑，将结果存入 Pxc 字典中，这里储存是 Pxc{q1{"1","2","3"}} 两个字典嵌套，q1 是该 feature 对应的字典，里面 1,2,3 对应储存了该 feature 在每个类别之中的概率，对连续型 featuretype=1, 此时用正态分布去估计，同理也是存在 Pxc 字典里面，这个结构与离散型稍有不同，Pxc{q1{"mean","var"}},q1 也是储存了对应 feature 的字典，里面由于是连续型，只需要保存均值和方差即可。

```
55  def predict(self,features,featuretype):
56      pred=np.array([0 for x in range(len(features[:,0]))])
57
58      for i in range(len(features[:,0])):
59          k=0
60          p1=float('-inf')
61          p2=0
62          for key1 in self.Pc:
63              p2=math.log(self.Pc[key1],2)
64              for j in range(len(featuretype)):
65                  if(featuretype[j]==1):
66                      p2=p2-1/2*math.log(2*math.pi*self.Pxc[j][key1]["var"],math.e)-math.pow(features[i,j]-self.Pxc[j][key1]
67                  if(featuretype[j]==0):
68                      p2=p2+math.log(self.Pxc[j][key1][features[i,j]])
69              if(p2>p1):
70                  k=key1
71                  p1=p2
72          pred[i]=k
73      return(pred.reshape(-1,1))
74
```

图 3.4: nbayes predict

这个 predict 方法，我先初始化了一个 pred 数组，用来储存预测出来的类别，后面一个 for 循环，对每组数据我们去计算这个数据在拥有对应特征下在每个类别里面的概率，去找最大值，这里我就直接用 k 暂时储存最大概率类别的类别，p1 为最大的概率，p2 为当前类别的概率。计算概率是取过了 log 的值，具

体原因在实验设计里面说过了。

3.1.3 SVM

```
50      def fit(self,train_data,train_label,test_data):
51          K=np.array([0.0 for x in range(len(train_data[:,0])*len(train_data[:,0]))])
52          K=K.reshape(-1,len(train_data[:,0]))
53          K1=np.array([0.0 for x in range(len(train_data[:,0])*len(train_data[:,0]))])
54          K1=K1.reshape(len(train_data[:,0]),len(train_data[:,0]))
55
56          for i in range(len(train_data[:,0])):
57              for j in range(len(train_data[:,0])):
58                  K1[i,j]=self.KERNEL(train_data[i,:],train_data[j,:],kernel=self.kernel)
59                  K[i,j]=train_label[i]*train_label[j]*K1[i,j]
60
61
62
63          p=np.array([-1.0 for x in range(len(train_data[:,0]))])
64          G=np.array([0.0 for x in range(2*len(train_data[:,0])*len(train_data[:,0]))])
65          G=G.reshape(2*len(train_data[:,0]),len(train_data[:,0]))
66          h=np.array([0.0 for x in range(2*len(train_data[:,0]))])
67          A=train_label.flatten()
68          b=0
69
70          for i in range(len(train_data[:,0])):
71              G[i,i]=-1.0
72              G[i+len(train_data[:,0]),i]=1.0
73              h[i]=0.0
74              h[i+len(train_data[:,0])]=float(self.C)
75
76          K=cvxopt.matrix(K,(len(train_data[:,0]),len(train_data[:,0])),'d')
77          b=cvxopt.matrix(b,(1,1),'d')
78          p=cvxopt.matrix(p,(len(train_data[:,0]),1),'d')
79          G=cvxopt.matrix(G,(2*len(train_data[:,0]),len(train_data[:,0])),'d')
80          h=cvxopt.matrix(h,(2*len(train_data[:,0]),1),'d')
81          A=cvxopt.matrix(A,(1,len(train_data[:,0])),'d')
82
```

图 3.5: SVM 1

SVM 只需要实现 fit 方法，不过里面的函数返回值就是预测类别。由于使用核方法，需要将这个核矩阵储存起来这里我使用 K1 来储存它，这里的 K 矩阵维数与 K1 一样，元素是 K1 对应元素乘上对应 label，是为了简化后面的代码创建的。这里由于是求解二次规划问题，调用了 cvxopt 包，使用了 cvxopt.solvers.qp(Q,p,G,h,A,b) 函数。对应二次规划问题

$$min\ \frac{1}{2}x^TQx + px \tag{3.1}$$

$$s.t.Gx \leq h \tag{3.2}$$

$$Ax = b \tag{3.3}$$

那么对应到这里，若记数据个数为 n，Q=K,p 为全为-1 的向量，维数为 n，G 的话由于约束问题对每个 α 两边都有约束，G 的行数为 2n，列数为 n。前 n 行与后 n 行都是一个 n 维单位阵，h 前 n 行为 0，后 n 行为 C。约束里面的等式约束为 $\sum_{i=1}^{N}\alpha_i y_i = 0$, 所以转化为 A 为数据的标签，b 为 0，然后比较搞的一点，直接用 numpy 还不行，所以后面我将 numpy 转为了 cvxopt 里面的矩阵。

```
        sol=cvxopt.solvers.qp(K,p,G,h,A,b)
        for i in range(len(train_data[:,0])):
            if(sol['x'][i]<self.Epsilon):
                sol['x'][i]=0
            if(sol['x'][i]>self.C-self.Epsilon):
                sol['x'][i]=1


        b=0.0
        qwq=np.array([0.0 for x in range(len(train_label))])
        for i in range(len(train_label)):
            qwq[i]=train_label[i]*sol['x'][i]
        count=0
        for i in range(len(train_data[:,0])):
            if(sol['x'][i]>0 and sol['x'][i]<self.C):
                count=count+1
                b=b+train_label[i]-np.dot(qwq,K1[:,i])
        b=b/count
        K2=np.array([0.0 for x in range(len(train_data[:,0])*len(test_data[:,0]))])
        K2=K2.reshape(len(train_data[:,0]),-1)
        for i in range(len(train_data[:,0])):
            for j in range(len(test_data[:,0])):
                K2[i,j]=self.KERNEL(train_data[i,:],test_data[j,:],kernel=self.kernel)
        result=np.dot(qwq,K2)+b

        result=result.reshape(-1,1)

        return result

```

图 3.6: SVM 1

解出来的结果，我们还要处理一下，将过于接近 0 的 α 去掉，和过于接近 C 的 α 都置为 C，这步是因为 python 自身的精度问题，过于小的数和接近 C 的数，最好是不视为支持向量，之前没有考虑这点，得出的准确度很低。后面的部分就是在测试集上预测了，我们使用公式

$$w^* = \sum_{i=1}^{N} \alpha_i^* y_i x_i \tag{3.4}$$

$$b^* = y_j - \sum_{i=1}^{N} y_i \alpha_i K(x_i, x_j) \tag{3.5}$$

其中 α_i^* 为二次规划的解，且第 j 个元素满足 $0 < \alpha_j < C$, 即表明第 j 个为支持向量，当然若考虑鲁棒性，我们可以使用所以支持向量算出来的平均值，不考虑的话，这里用一个支持向量也没有大问题。qwq 是我为了方便创建的一个数组，就是除掉核以外的那些系数，后面要预测，在测试集上我们需要算出新的核矩阵，记为 K2，用这个和 qwq 点积即可。最后调整一下结果的维数，变成列向量输出。

3.2 part2: 深度学习

3.2.1 MLP manual

```python
import torch
import numpy as np
import torch.autograd
import math
import matplotlib.pyplot as plt
device1 = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def sigmoid(x):
    return 1.0/(1+torch.exp(-x))
def softmax(x):

    return(torch.exp(x)/sum(torch.exp(x)))
def cross_entrophy(x,y):
    return(-torch.log(x[int(y)-1]))
def loss(x,y):
    sumloss=0.0
    for i in range(len(y)):
        sumloss=sumloss+cross_entrophy(x[:,i],y[i])

    sumloss=sumloss/len(y)

    return sumloss
```

图 3.7: mlp 1

这一模块是我调用的包,只用到了基本的 torch,以及需要比较自动求导与我手动实现,调用了 torch.autograd。然后 matplotlib.pyplot 用于 loss 曲线的绘制，我安装了 cuda，所以这个计算全都在 GPU 上运行，后面的 sigmoid,softmax 就是自己实现的对应的激活函数，其中交叉熵这里，我做了一点简化，考虑到如果用 one-hot 形式输入，在后面手动求梯度时还要去转为 1，2，3 这三类，比较麻烦。我就之间将 y 作为一个 100 维的列向量输入进去了 (实验文档里也只说了输入 lable 为 100 维列向量，真转化为 one-hot 也不复杂，前面机器学习部分的代码里面就有)，毕竟这种情况下的交叉熵其实也就是对对应的类别的值取负对数即可。然后 loss 函数就是对每组数据求交叉熵后求和的平均值。

```python
class mlpmodel():
    def __init__(self,hidden,outlayer,iterations=1000,lrate=0.001):
        self.hidden=hidden
        self.iterations=iterations
        self.lrate=lrate
        self.outlayer=outlayer


    def _initweight_(self,x,y):
        feature_nx,sample_n=x.shape
        feature_ny=self.outlayer
        self.W0=torch.randn(self.hidden,feature_nx,device=device1,requires_grad=True)
        self.w0=torch.randn(self.hidden,1,device=device1,requires_grad=True)
        self.W1=torch.randn(self.hidden,self.hidden,device=device1,requires_grad=True)
        self.w1=torch.randn(self.hidden,1,device=device1,requires_grad=True)
        self.W2=torch.randn(feature_ny,self.hidden,device=device1,requires_grad=True)
        self.w2=torch.randn(feature_ny,1,device=device1,requires_grad=True)


    def forward(self,x,y):
        input1=torch.matmul(self.W0,x)+self.w0
        output1=sigmoid(input1)
        input2=torch.matmul(self.W1,output1)+self.w1
        output2=sigmoid(input2)
        input3=torch.matmul(self.W2,output2)+self.w2
        output3=softmax(input3)
        cost=loss(output3,y)
        return output1,output2,output3,cost
```

图 3.8: mlp 2

这里 init 方法里面，我定义了 hidden 为隐含层的节点数，lrate 为每次梯度下降的学习率，outlayer 为输出层的节点数，iteration 为迭代次数，是这个网络最基本的构成，故我单独列出来。后面 initweight 方

法，是对权值矩阵的初始化，这一块比较多，为了方便阅读，没有直接放在 init 里面。这里权值矩阵的维数为 (4*5),(4*4),(4*3). 权值的初始化我用正态分布随机，w0,w1,w2 为我加的三个偏倚量。这是因为如果全随机成 0，经过测试比较奇怪，loss 曲线基本是不动的，虽然这个不会影响求导。后面 requires_grad=True 这个参数设置是为了后面与自动求导比较。

在 forward 方法，即正向传播里面，就依次对输入进行线性变换，激活这样操作，最后返回每一次激活后的输出以及 cost，为自动求导和画 loss 曲线做准备。

```
def backward(self,x,y,output1,output2,output3,cost):

    feature_ny=self.outlayer
    dfy=torch.zeros(feature_ny,len(y),device=device1)
    one1=torch.ones(1,len(y),device=device1)
    for i in range(len(y)):
        for j in range(feature_ny):
            if(j==y[i]-1):
                dfy[int(y[i])-1,i]=output3[int(y[i])-1,i]-1.0
            else:
                dfy[j,i]=output3[j,i]
    dw2=torch.matmul(dfy,
    dW2=torch.matmul(dfy,output2.t())/len(y)
    ds1=torch.zeros(self.hidden,len(y),device=device1)
    ds0=torch.zeros(self.hidden,len(y),device=device1)
    for i in range(len(y)):
        for j in range(self.hidden):
            ds1[j,i]=output2[j,i]*(1.0-output2[j,i])
            ds0[j,i]=output1[j,i]*(1.0-output1[j,i])
    dhi1=torch.mul(torch.matmul(self.W2.t(),dfy),ds1)
    dW1=torch.matmul(dhi1,output1.t())/len(y)
    dw1=torch.matmul(dhi1,one1.t())/len(y)
    dW0=torch.matmul(torch.mul(torch.matmul(self.W1.t(),dhi1),ds0),x.t())/len(y)
    dw0=torch.matmul(torch.mul(torch.matmul(self.W1.t(),dhi1),ds0),one1.t())/len(y)
```

(parameter) output2: Any

图 3.9: mlp back 1

这里开始反向传播，这部分分为两个小块，这里展示的是我手动用矩阵运算求导，公式都在助教给的实验文档里面写了，其中 dfy 矩阵是实验文档中对应的 $(l's_3')$. dhi1 矩阵是实验文档对应的 $(W_3^T(l's_3') \odot s_2')$,（注意：我的代码里面，权值矩阵的编号是从 0 开始的）, 保存这两个值，主要是在矩阵运算中会反复用到，可以简化一下代码。最后每一个梯度的输出，我都是在对应的矩阵或向量前面加一个 d 表示。

```
    cost.backward()
    print(self.W2.grad)
    print(dW2)
    print(self.W1.grad)
    print(dW1)
    print(self.W0.grad)
    print(dW0)
    print(self.w2.grad)
    print(dw2)
    print(self.w1.grad)
    print(dw1)
    print(self.w0.grad)
    print(dw0)
    self.W2.data=self.W2.data-self.lrate*dW2.data
    self.W1.data=self.W1.data-self.lrate*dW1.data
    self.W0.data=self.W0.data-self.lrate*dW0.data
    self.w2.data=self.w2.data-self.lrate*dw2.data
    self.w1.data=self.w1.data-self.lrate*dw1.data
    self.w0.data=self.w0.data-self.lrate*dw0.data
    self.W2.grad.zero_()
    self.W1.grad.zero_()
    self.W0.grad.zero_()
    self.w2.grad.zero_()
    self.w1.grad.zero_()
    self.w0.grad.zero_()
```

图 3.10: mlp back 2

这一块就是验证我的矩阵求导是否正确，输出是上面为 torch 自动求导的输出，下面为我手动求导的输出。每一步迭代这 12 个输出都会被打印出来，供比较。因为这里调用了 torch 的 backward，原来的张

量 W0,W1,W2,w0,w1,w2 已经被改动了，所以要在每个张量后面加上 data 表示用的是这个张量的值。梯度下降我用的是我自己求出来的梯度。后面是将梯度清零，这一步不做后面的梯度是错误的，会不断累积起来。

```
138     def fit(self,x,y):
139         
140             self._initweight_(x,y)
141             num=[]
142             costlist=[]
143             for i in range(self.iterations):
144                 output1,output2,output3,cost=self.forward(x,y)
145                 costlist.append(cost)
146                 num.append(i+1)
147                 self.backward(x,y,output1,output2,output3,cost)
148             plt.plot(num, np.squeeze(np.array(costlist)))
149             plt.xlim((1, 1500))
150             plt.ylim(0, 2)
151             plt.xlabel('iterate_nums')
152             plt.ylabel('costvalue')
153             plt.savefig('G:\绷带\LAB2_for_students\src2')
154             plt.show()
```

图 3.11: mlp fit

fit 这一模块就是进行梯度下降，初始了两个列表，num 和 costlist 用来保存迭代次数已经对应次数的 loss，这个后面绘图时会用到。143 行到 147 行就是不停的正向传播再反向传播。最后给出了一个结果 plt.show 方法将其展示出来。

```
158 if __name__=='__main__':
159         x=torch.randn(100,5,device=device1)
160         k=torch.rand(100,1,device=device1)
161         y=torch.zeros(100,1,device=device1)
162         for i in range(len(k)):
163             if(k[i]<=1/3):
164                 y[i]=1
165             if(k[i]>1/3 and k[i]<=2/3):
166                 y[i]=2
167             if(k[i]>2/3):
168                 y[i]=3
169         qwq=mlpmodel(hidden=4,outlayer=3)
170         qwq.fit(x.t(),y)
171         _,__,output,____=qwq.forward(x.t(),y)
172         pred=np.array([0 for x in range(len(y))])
173         pred=torch.argmax(output.t(),dim=1).cpu().numpy()+1
174         sum=0
175         for i in range(len(y)):
176             if(pred[i]==int(y[i])):
177                 sum=sum+1
178         print(sum/len(y))
179
```

图 3.12: mlp main

这里主模块，我用正态分布随机生成 100 个五维的数据，以及 (0,1) 均匀分布 100 个 1 维标签，不过这里的标签是连续值，所以我后面处理了一下将 (0,1) 区间平均分为三段，每一段对应一个类别。后面调用我的 mlp 类训练出模型，pred 是在训练集上预测结果, 简单求了一下准确率 (不过这里这步倒是没有太大必要)

3.2.2 MLPmixer

```python
class Mixer_Layer(nn.Module):
    def __init__(self, patch_size, hidden_dim):
        super(Mixer_Layer, self).__init__()
        ######################################################################
        # 这里需要写Mixer_Layer（layernorm, mlp1, mlp2, skip_connection）
        self.patch_size = patch_size
        self.hidden_dim = hidden_dim
        self.layer_norm = nn.LayerNorm(self.hidden_dim)
        self.mlp1 = nn.Sequential(
            nn.Linear(int(28/patch_size) ** 2, 98),
            nn.GELU(),
            nn.Linear(98, int(28/patch_size) ** 2)
        )
        self.mlp2 = nn.Sequential(
            nn.Linear(hidden_dim, patch_size**2),
            nn.GELU(),
            nn.Linear(patch_size**2,hidden_dim)
        )
        ######################################################################

    def forward(self, x):
        ######################################################################
        output = self.layer_norm(x).transpose(1, 2)
        output = self.mlp1(output).transpose(1, 2) + x
        output = self.mlp2(self.layer_norm(output)) + output
        return output
        ######################################################################

```

图 3.13: MLPmixer mixerlayer

这一块是 mixer 层，要求实现 layernorm,mlp1,mlp2,skip connection。layernorm 即做一个归一化，这在 torch 的包里面可以直接调用，这里 layernorm 的输入的最后一维是 hidden_dim 维的所以参数用 hidden_dim, 做归一化后需要进行转置，这里需要对特定的后面两维转置，这里用了 transpose 方法，可以指定将某两维转置。mlp1,mlp2 对应的是文献中提到的那两个 mlp 层分别称为，token-mixing 和 channal-mixing，其中我直接用 nn.linear 做连接，mlp1 层每一个数据输入为 patchnum 维即 $(28/patchsize)^2$, 中间的输出维数理论上没有要求，我直接就用 patchnum 维输出，只要最后面的 linear 保证输出维数和输入维数一样就行，mlp2 层也是同理，保证输入和输出的维数相同，这里都是 hidden_dim 维。两个 linear 中间是 GELU 激活函数，，这三个我用 nn.sequential, 按顺序包装在一起，显得紧凑。在 forward 里面，mlp1 层的输出也是需要做了一个转置，然后做 layernorm 之后传给 mlp2。skip connection 的实现是在 forward 里面后面加上了 x 和 output。skip connection 就是直接拿上层的输入加入到输出里面, 作用就是防止若网络比较深导致梯度爆炸或者梯度消失。

```python
class MLPMixer(nn.Module):
    def __init__(self, patch_size, hidden_dim, depth):
        super(MLPMixer, self).__init__()
        assert 28 % patch_size == 0, 'image_size must be divisible by patch_size'
        assert depth > 1, 'depth must be larger than 1'
        ######################################################################
        # 这里写Pre-patch Fully-connected, Global average pooling, fully connected
        self.patch_size = patch_size
        self.hidden_dim = hidden_dim
        self.patchnum=int(28/self.patch_size)**2
        self.pre_batch_full_connect = nn.Linear(28 * 28, 28 * 28)
        self.conv = nn.Linear(self.patch_size ** 2, hidden_dim)
        self.mix_layers = nn.Sequential(*[Mixer_Layer(patch_size, hidden_dim) for i in range(depth)])
        self.full_connect = nn.Linear(self.patchnum* self.hidden_dim, 10)
        ######################################################################

    def forward(self, x):
        ######################################################################
        # 注意维度的变化
        x = x.view(-1, 28 * 28)
        output = self.pre_batch_full_connect(x)
        output = self.conv(output.view(-1, self.patchnum, self.patch_size ** 2))
        output = self.mix_layers(output)
        output = output.view(-1, self.patchnum * self.hidden_dim)
        output = self.full_connect(output)
        return output
        ######################################################################

```

图 3.14: MLPmixer mlpmix

这一块就是 mipmixer 的框架，需要实现的 Pre-patch Fully-connected, Global average pooling, fully

connected 如代码所示。Pre-patch Fully-connected 嵌入层, 即将图片由 28*28 展成 1*(28*28) 做一个全连接, 这里用的 linear。然后进行卷积，我们将一张图片拆成 patchnum*$patchsize^2$, 卷积之后得到每个 patch 整体的信息，得到三维的张量 (datanum,patchnum,hidden_dim)，datanum 是数据个数，patchnum 是 patch 个数。将这个送入上面描述的 mixer 层，需要做 depth 个这样的 mixer 层，这里就直接用 sequential 简化了代码，sequential 里面按顺序包装了 depth 个同样的 mixer 层。最后将结果通过一个全连接输出，输出维数 (datanum,10)，这里 10 是有 10 个类别。

```
70  def train(model, train_loader, optimizer, n_epochs, criterion):
71      model.train()
72      for epoch in range(n_epochs):
73          for batch_idx, (data, target) in enumerate(train_loader):
74              data, target = data.to(device), target.to(device)
75              ########################################################################
76              # 计算loss并进行优化
77              optimizer.zero_grad()
78              output = model(data)
79              loss = criterion(output, target)
80              loss.backward()
81              optimizer.step()
82              ########################################################################
83              if batch_idx % 100 == 0:
84                  print('Train Epoch: {}/{} [{}/{}]\tLoss: {:.6f}'.format(
85                      epoch, n_epochs, batch_idx * len(data), len(train_loader.dataset), loss.item()))
86
```

图 3.15: MLPmixer train

这里是 train 模块，每次 train 会传入一个 batch 数目的数据，每 100 次迭代就会输出一下。这里就调用 optimizer 对 loss 进行优化，loss 函数用 criterion 函数进行计算，这里损失函数是交叉熵。

```
88  def test(model, test_loader, criterion):
89      model.eval()
90      test_loss = 0.
91      num_correct = 0  # correct的个数
92      total = 0
93      with torch.no_grad():
94          for data, target in test_loader:
95              data, target = data.to(device), target.to(device)
96              ########################################################################
97              # 需要计算测试集的loss和accuracy
98              output = model(data)
99              loss = criterion(output, target)
100             test_loss =test_loss+ loss
101             num_correct =num_correct+ (torch.argmax(output, 1) == target).sum().item()
102             total =total+ len(target)
103         accuracy = num_correct / total
104         test_loss=test_loss*len(target)/total
105         ########################################################################
106         print("Test set: Average loss: {:.4f}\t Acc {:.2f}".format(test_loss, accuracy))
```

图 3.16: MLPmixer test

这里是 test 模块，将所有的测试集分批次导入到模型中，每次得到的 loss 相加, 注意这里的 loss 是每个批次的平均 loss，所有要求总的平均 loss 我们要将这个所有 loss 相加的结果乘以 batchsize 除以 total，这里的 batchsize 可以由 len(target) 得到。这里的 accuracy 也是每次导入一批，我们将每个数的 10 个输出结果中最大的那个对应的类别视为预测类别。去和 target 比较，求和再将这个数提取出来，即为这一批的 num_correct, 然后所有批次的相加即可。

Chapter 4

实验结果与分析

4.1 part1: 传统机器学习

这一部分我会贴上我的输出

4.1.1 线性分类器

```
(base) G:\绸带\LAB2_for_students\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
0.6214099216710182
0.6032786885245902
0.6347305389221557
macro-F1: 0.6198063830392546
micro-F1: 0.6174974567650051
```

图 4.1: linearclassification output

这是线性分类器的输出结果，可以看到总的预测准确度达到了 0.61，对每个类别的准确度也有 0.6 朝上，micro-F1 和 macro-F1 也有 0.6 以上，说明模型实现没有问题，结果比较理想

4.1.2 多分类朴素贝叶斯

```
(base) G:\绸带\LAB2_for_students\src1>python nBayesClassifier.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.5198372329603256
0.33834586466165417
0.5924006908462867
0.45387453874538747
macro-F1: 0.46154036475110943
micro-F1: 0.5198372329603256
```

图 4.2: nbayes output

这是朴素贝叶斯的输出结果，可以看到总的预测准确度达到 0.5，比线性分类器稍差，也有可能还是精度有一定影响，或者假设为正态分布没有特别合理，但是结果也还可以。总体实现没有太大问题。

4.1.3 SVM

```
12: -1.8757e+03 -1.8783e+03  3e+00  6e-07  5e-14
13: -1.8760e+03 -1.8779e+03  2e+00  3e-07  4e-14
14: -1.8763e+03 -1.8776e+03  1e+00  2e-07  4e-14
15: -1.8765e+03 -1.8774e+03  9e-01  1e-07  4e-14
16: -1.8767e+03 -1.8772e+03  5e-01  3e-08  5e-14
17: -1.8768e+03 -1.8771e+03  3e-01  1e-08  5e-14
18: -1.8769e+03 -1.8769e+03  1e-01  1e-09  5e-14
19: -1.8769e+03 -1.8769e+03  5e-02  2e-10  5e-14
20: -1.8769e+03 -1.8769e+03  7e-03  2e-11  5e-14
21: -1.8769e+03 -1.8769e+03  2e-04  1e-13  5e-14
Optimal solution found.
Acc: 0.6561546286876907
0.7539503386004515
0.5740498034076016
0.6815789473684212
macro-F1: 0.6698596964588247
micro-F1: 0.6561546286876907
```

图 4.3: SVM gauss output

```
14: -1.9271e+03 -1.9271e+03  4e-03  2e-09  5e-13
15: -1.9271e+03 -1.9271e+03  4e-05  2e-11  5e-13
Optimal solution found.
Acc: 0.602238046795524
0.6967509025270758
0.21973094170403587
0.7246376811594203
macro-F1: 0.547039841796844
micro-F1: 0.602238046795524
```

图 4.4: SVM linear output

```
17: -1.8679e+03 -1.8689e+03  1e+00  1e-07  3e-12
18: -1.8683e+03 -1.8684e+03  8e-02  7e-09  3e-12
19: -1.8683e+03 -1.8683e+03  1e-02  9e-10  3e-12
20: -1.8683e+03 -1.8683e+03  4e-04  2e-11  3e-12
Optimal solution found.
Acc: 0.6490335707019329
0.750551876379691
0.5822784810126582
0.6583679114799447
macro-F1: 0.6637327562907647
micro-F1: 0.6490335707019329
```

图 4.5: SVM poly output

这三张图分别是选择高斯核，线性核，多项式核的结果，前面是二次规划迭代过程，很长就没有一一截下来，截下来也没有太大意义，可以看到高斯核准确率 0.65，各个类别的 f1-score 也很不错，线性核低一点只有 0.60，多项式核准确率 0.64，比线性核好，比高斯核差一点。总体而言这三种核的准确率都很好，所以 SVM 实现也没有问题。

4.2 part2: 深度学习

4.2.1 MLP manual

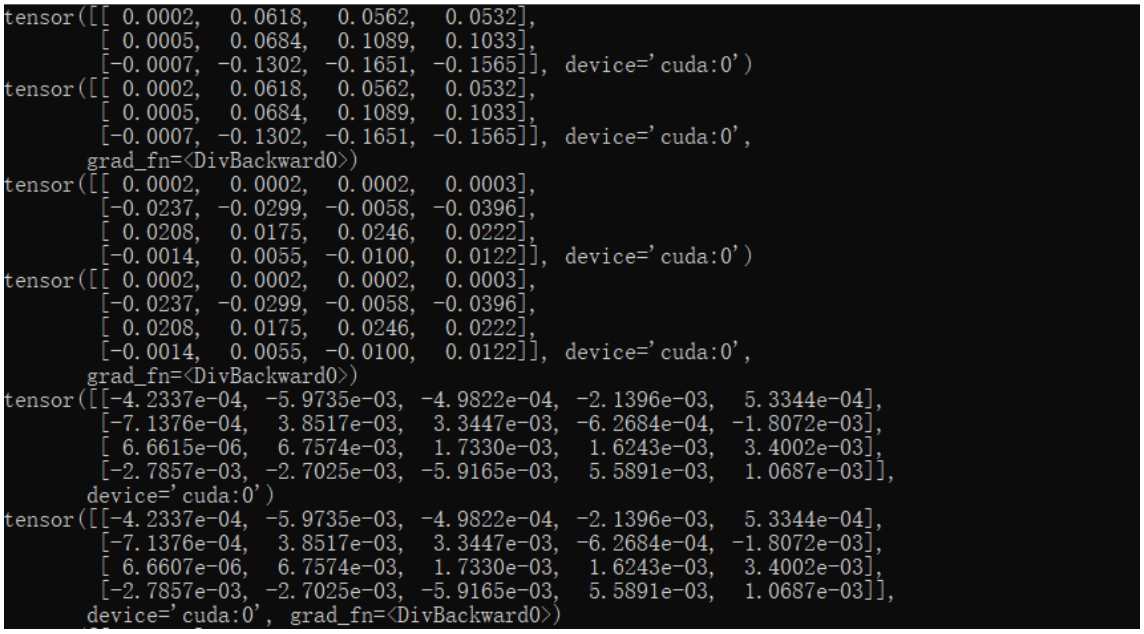

图 4.6: mlp output1

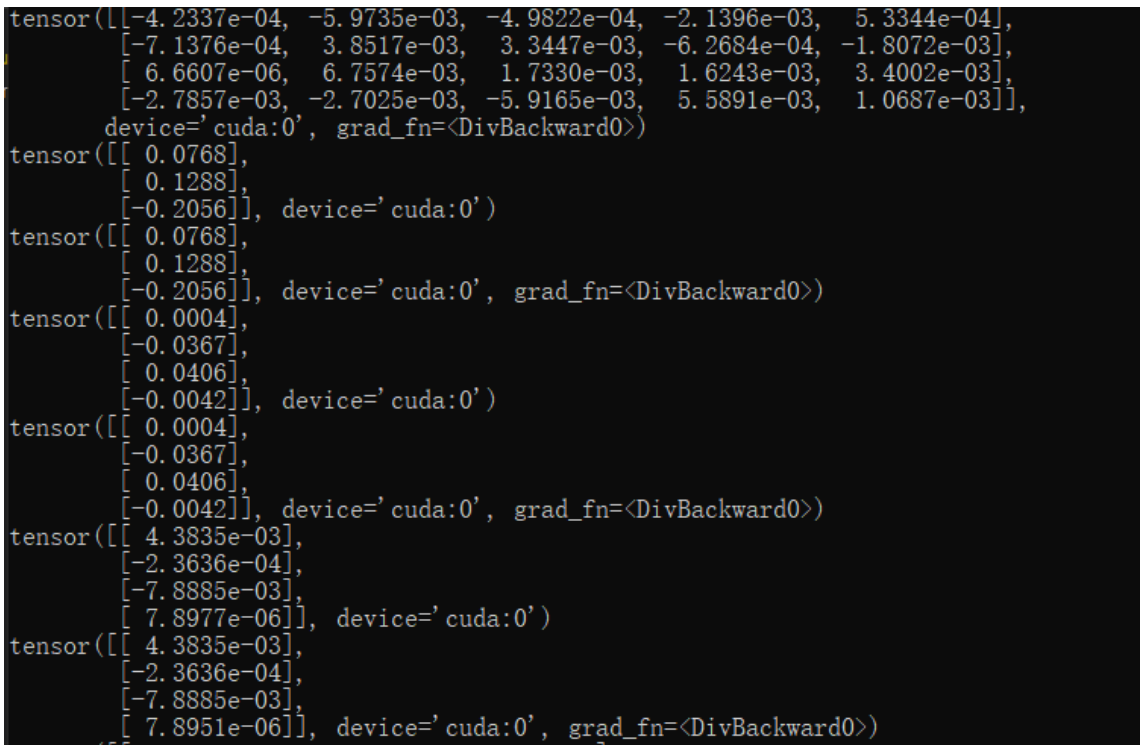

图 4.7: mlp output2

这一个输出是检验手动求导和自动求导是否相同。我迭代了 1000 次，这里输出有很多，我只随机截取其中一组输出。每一组有 12 个输出，其中连续两个输出进行比较，对应 W2,W1,W0,w2,w1,w0 上面的为自动求导的参数，下面为手动求导参数，可以看到，这个是相同的，助教也可以自行查看其它的，都是相同的。手动求导的实现是没有问题的。

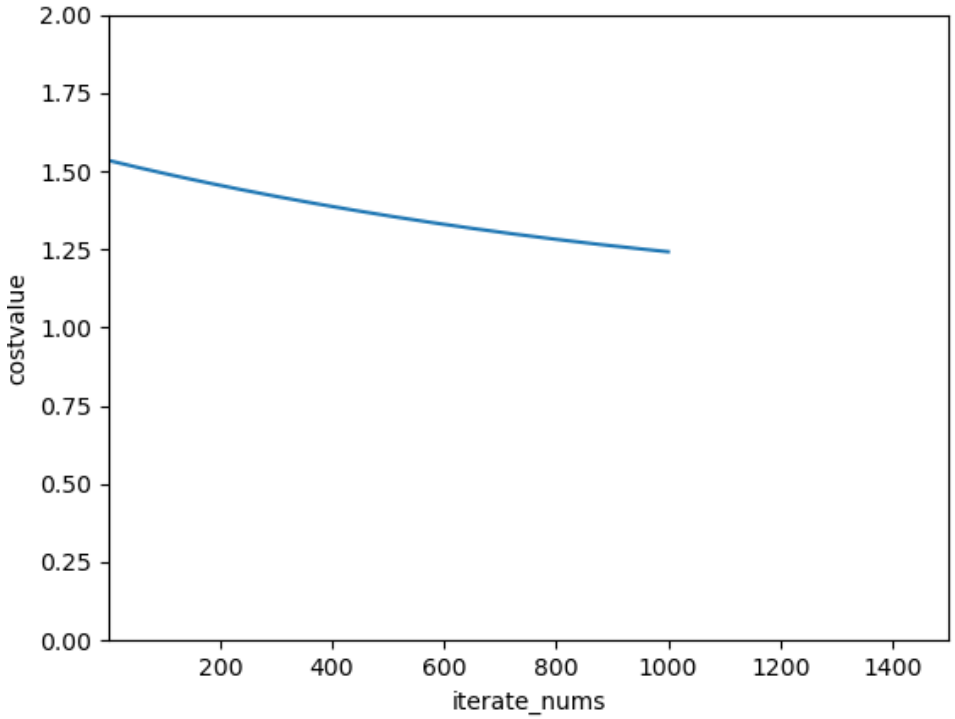

图 4.8: mlp output3

这个图就是 loss 曲线，可以看到这个曲线是下降趋势的，不过由于数据随机，这个 loss 比较高。这个梯度下降的实现是没有问题的。

4.2.2 MLPmixer

```
Train Epoch: 0/5 [0/60000]       Loss: 2.318613
Train Epoch: 0/5 [12800/60000]   Loss: 0.472387
Train Epoch: 0/5 [25600/60000]   Loss: 0.229506
Train Epoch: 0/5 [38400/60000]   Loss: 0.242937
Train Epoch: 0/5 [51200/60000]   Loss: 0.082396
Train Epoch: 1/5 [0/60000]       Loss: 0.181275
Train Epoch: 1/5 [12800/60000]   Loss: 0.241996
Train Epoch: 1/5 [25600/60000]   Loss: 0.100883
Train Epoch: 1/5 [38400/60000]   Loss: 0.173019
Train Epoch: 1/5 [51200/60000]   Loss: 0.071197
Train Epoch: 2/5 [0/60000]       Loss: 0.095003
Train Epoch: 2/5 [12800/60000]   Loss: 0.163765
Train Epoch: 2/5 [25600/60000]   Loss: 0.081080
Train Epoch: 2/5 [38400/60000]   Loss: 0.059081
Train Epoch: 2/5 [51200/60000]   Loss: 0.044107
Train Epoch: 3/5 [0/60000]       Loss: 0.026677
Train Epoch: 3/5 [12800/60000]   Loss: 0.174109
Train Epoch: 3/5 [25600/60000]   Loss: 0.041058
Train Epoch: 3/5 [38400/60000]   Loss: 0.046677
Train Epoch: 3/5 [51200/60000]   Loss: 0.131479
Train Epoch: 4/5 [0/60000]       Loss: 0.038090
Train Epoch: 4/5 [12800/60000]   Loss: 0.029652
Train Epoch: 4/5 [25600/60000]   Loss: 0.084057
Train Epoch: 4/5 [38400/60000]   Loss: 0.013280
Train Epoch: 4/5 [51200/60000]   Loss: 0.018735
Test set: Average loss: 0.0126   Acc 0.97
```

图 4.9: mlpmixer

这个图是我实现的 MLPmixer 的输出，可以看到在测试集上平均损失只有 0.0126，准确率也达到了 0.97，这说明我的实现也没有问题。

Chapter 5

实验总结

本次实验，我接触到了传统机器学习的各种经典方法，并尝试实现了他们，这让我对机器学习理解更为深刻，也初步接触了深度学习相关知识，体会到了深度学习比传统的机器学习的优越性，也锻炼了我查找资料的能力.