AI Lab2 实验报告

PB18000268 曾勇程

1. 实验内容与提示

本次实验包含**传统机器学习**与**深度学习**两部分。
实验部分需要使用python=3.6，建议使用anaconda管理python环境，深度学习部分要求使用pytorch=1.8.1， torchvision=0.9.1完成（安装说明见 https://pytorch.org ，学习教程可以参考 https://pytorch123.com ），实验部分使用CPU足够训练，如果想体验GPU的速度可以使用colab。

2. 传统机器学习

数据集：鲍鱼数据集

任务：根据鲍鱼的物理测量属性预测鲍鱼的年龄

2.1 实现一个线性分类算法

对引入了 L2 规范化项之后的最小二乘分类问题进行推导。即求解以下优化问题：

$$min_w(Xw-y)^2+\lambda||w||^2$$

L2规范化即是在最小二乘法的基础上，加1个对系数的**惩罚项**，为了方便计算所以加上的是**L2-norm**的平方，这时候损失函数就为

$$L(w)=\frac{1}{2M}\sum_{j=1}^{M}(y^{(j)}-h_w(x^{(j)}))^2+\lambda\sum_{i=1}^{N}w_i^2$$

$$h_w(x)=w_1x_1+w_2x_2+\ldots+w_nx_n$$

关于w_i 对 $L(w)$ 求导，可得：

$$\frac{\partial L(w)}{w_i}=-\frac{1}{M}\sum_{j=1}^{M}(y^{(j)}-h_w(x^{(j)}))*x_i^{(j)}+2\lambda w_i$$

由梯度下降算法，可得迭代公式：

$$w_i = w_i + \frac{\alpha}{M}\sum_j (y^{(j)} - h(x^{(j)}))x_i^{(j)} - \alpha\lambda' w_i$$

$$\lambda' = 2\lambda$$

其中，α 为学习速率。

线性回归代码如下：

```
    def fit(self,train_features,train_labels):
        ''''
        需要你实现的部分
        '''
        # print(train_features)
        train_features = np.c_[np.ones(len(train_features)), train_features]  # 增加常数偏移值
        self.W = np.ones(9)   # 权值

        for k in range(self.epochs):     # 迭代 epochs 次，训练权值
            for i in range(9):           # 梯度下降
                Grad = 0
                # 计算最小二乘法部分的导数
                for j in range(len(train_features)):
                    Grad = Grad + (train_labels[j][0] - np.dot(self.W, train_features[j])) * train_features[j][i]
                self.W[i] = self.W[i] + self.lr * (Grad / len(train_features) - self.Lambda * self.W[i])

        print("weights: ", self.W)  # 输出最终训练权值
        return self.W
```

预测函数：

```
    def predict(self,test_features):
        ''''
        需要你实现的部分
        '''
        pred = []
        test_features = np.c_[np.ones(len(test_features)), test_features]  # 加上常数列
        for i in range(len(test_features)):
            cla = np.dot(self.W, test_features[i])       # 预测类别
            pred.append(int(round(cla)))
        pred = np.array(pred).astype(int).reshape(-1, 1) # 转换矩阵（数组）形式
        return pred
```

最终结果如下：

```
D:\Anaconda\envs\pytorch\python.exe D:/Pycharm/python_project/AI_Lab/Lab2/src1/linearclassification.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
weights:  [ 0.9835947  -0.11432215  0.92699779  0.95903375  0.95393291 -0.03141004
  0.24126694  0.75614358  0.89689363]
Acc: 0.612410986775178
score:  0.630057803468208
score:  0.5991471215351813
score:  0.6226138032305433
macro-F1: 0.6172729094113109
micro-F1: 0.6127226463104325

Process finished with exit code 0
```

正确率在百分之六十几，效果较好。但是线性分类器较为单调、简单，因此性能上也不太可能有很大的提升了，对于线性回归，百分之六十几应该算是比较好的预测结果。

2.2 实现一个朴素贝叶斯分类器

完善nBayesClassifier.py的代码,以实现朴素贝贝叶斯分类器，使用拉普拉斯平滑计算条件概率和先验概率。

$$\hat{P}(c) = \frac{|D_c + 1|}{|D| + N}$$
$$\hat{P}(x_i|c) = \frac{|D_{c,x_i}| + 1}{|D_c + N_i|}$$

其中D表示训练集，D_c表示其中类别为c的数据，。D_{c,x_i}表示类别为c，第i个属性值为x的数据。N_i表示第i个属性可能的取值数。

判定准则为

$$h_{nb}(x) = argmax_{c \in Y} P(c) \prod_{i=1}^{d} P(x_i|c)$$

即，预测时，只需判断他属于哪一类的概率最高，预测它为概率最高的那一类。

对于连续性数据，采取的处理方法是：

假设连续变量服从某种概率分布，然后使用训练数据估计分布的参数，通常使用高斯分布用来表示连续属性的类条件概率分布，即用训练数据估计对应于每个类的均值 μ 和方差 σ^2。即用概率密度替换概率。

代码如下：

首先定义 高斯概率密度函数：

```python
'''
    采用第二种方法，计算高斯概率密度函数
    mean: 平均值
    std: 方差
'''
def GaussProb(self, x, mean, std):
    exponent = np.exp(-(np.power(x - mean, 2)) / (2 * np.power(std, 2)))
    GaussProb = (1 / (np.sqrt(2 * np.pi) * std)) * exponent
    return GaussProb
```

学习适应训练集函数:

```python
    def fit(self,traindata,trainlabel,featuretype):
        '''
        需要你实现的部分
        '''
        # 先计算先验概率和离散的 Sex 的后验概率，因为 Sex是离散型变量
        Label = {}  # 每个标签的数量
        Sex = np.zeros((3, 3))  # 3 * 3 矩阵，表示3个类别3种性别的数量
        totalnum = len(traindata)
        for l in range(len(trainlabel)):
            if trainlabel[l][0] in Label:
                Label[trainlabel[l][0]] += 1
            else:
                Label[trainlabel[l][0]] = 1
            Sex[trainlabel[l][0]-1][int(traindata[l][0])-1] += 1     # 每个类别中每种性别的数量
        for key, value in Label.items():
            self.Pc[key] = (value + 1) / (totalnum + 3)     # N = 3

        SexSum = list(map(sum, Sex))    # 求出每一个类别的总数
        self.Pxc["class1"] = {}
        for i in range(3):
            for j in range(3):
                self.Pxc["class1"]["%d-%d" % (i+1, j+1)] = (Sex[i][j] + 1) / (SexSum[i] + 3)
        # print(self.Pxc)

        # 开始处理连续变量
        for i in range(1, 8):
            par = {}
            Classes = [[], [], []]

            for j in range(len(traindata)):     # 分类不同类别的属性
                Classes[trainlabel[j][0] - 1].append(traindata[j][i])

            par["1"] = [np.mean(np.array(Classes[0])), np.std(np.array(Classes[0]))]    # 计算参数
            par["2"] = [np.mean(np.array(Classes[1])), np.std(np.array(Classes[1]))]
            par["3"] = [np.mean(np.array(Classes[2])), np.std(np.array(Classes[2]))]
            self.Pxc["class%d" % (i+1)] = par

        # print(self.Pxc)
```

首先计算先验概率 $\hat{P}(c) = \frac{|D_c+1|}{|D|+N}$，用 Label 字典记录出现的每个标签的数量，按照公式算出先验概率；

然后先算出离散属性的后验概率，这里只有 Sex 属性是离散属性，只有 3 个可取的值， 计算出每个类别每种 Sex 的频率近似他的条件概率；

对于连续型变量，只需记录高斯分布的参数，后面用到时将参数代入即可。

预测函数：

```
'''
    根据先验概率分布p(c)和条件概率分布p(x|c)对新样本进行预测
    返回预测结果,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
    feature_type为0-1数组，表示特征的数据类型，0表示离散型，1表示连续型
    '''
    def predict(self,features,featuretype):
        '''
        需要你实现的部分
        '''
        pred = []
        S = [1, 1, 1]
        for i in range(len(features)):
            S[0] = self.Pc[1] * self.Pxc["class1"]["%d-%d" % (1, features[i][0])]  # 判断这3类哪一类概率最高
            S[1] = self.Pc[2] * self.Pxc["class1"]["%d-%d" % (2, features[i][0])]
            S[2] = self.Pc[3] * self.Pxc["class1"]["%d-%d" % (3, features[i][0])]
            for j in range(1, 8):
                S[0] = S[0] * self.GaussProb(features[i][j], self.Pxc["class%d" % (j+1)]["1"][0], self.Pxc["class%d" % (j+1)]["1"][1])
                S[1] = S[1] * self.GaussProb(features[i][j], self.Pxc["class%d" % (j+1)]["2"][0], self.Pxc["class%d" % (j+1)]["2"][1])
                S[2] = S[2] * self.GaussProb(features[i][j], self.Pxc["class%d" % (j+1)]["3"][0], self.Pxc["class%d" % (j+1)]["3"][1])
            pred.append(S.index(max(S))+1)  # 预测类别
        pred = np.array(pred).astype(int).reshape(-1, 1)
        return pred
```

取出概率最高（按照公式）的类的下标 （+1）作为预测类别。

输出如下：

```
D:\Anaconda\envs\pytorch\python.exe D:/Pycharm/python_project/AI_Lab/Lab2/src1/nBayesClassifier.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6134282807731435
score:  0.7137404580152672
score:  0.4725111441307578
score:  0.6684005201560468
macro-F1: 0.6182173741006906
micro-F1: 0.6134282807731435

Process finished with exit code 0
```

朴素贝叶斯分类器依靠的主要是频率，准确率也在百分之六十几，和线性回归差不多。同线性回归，朴素贝叶斯分类器也只是简单地运用了数据集，因此准确率相差不多。

2.3 实现SVM分类器

完善 `SVM.py` 中的代码，以实现支持软间隔与核函数的 SVM。
对于 K 分类(K>2)，我们使用 `one-vs-all` 策略训练，具体为：对于任一类别，我们将其看作正类 “1”，其余类别看作负类 “-1”, 分别训练得到K个二分类器；测试时，对于一给定样本，分别计算该样本在K个二分类器上的输出/分数，取最大输出/分数所对应的分类器的正类作为最终的预测类别。（这一部分已在代码中给出）
在给出的代码中已经给出了线性核、高斯核、多项式核的实现，大家可以比较不同实现方式的结果。

增加软间隔后我们的优化目标变成了：

$$\min_{w} \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i$$
$$s.t. \quad g_i(w,b) = 1 - y_i(w^T x_i + b) - \xi_i \le 0, \quad \xi_i \ge 0, \quad i = 1,2,\ldots,n$$

其中 C 是一个大于 0 的常数，可以理解为错误样本的惩罚程度，若 C 为无穷大，ξ_i 必然无穷小，如此一来线性 SVM 就又变成了线性可分 SVM；当 C 为有限值的时候，才会允许部分样本不遵循约束条件。

接下来我们将针对新的优化目标求解最优化问题：

步骤 1：

构造拉格朗日函数：

$$\min_{w,b,\xi}\max_{\lambda,\mu} L(w,b,\xi,\lambda,\mu) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i + \sum_{i=1}^{n}\lambda_i[1 - \xi_i - y_i(w^T x_i + b)] - \sum_{i=1}^{n}\mu_i\xi_i$$
$$s.t. \quad \lambda_i \ge 0 \quad \mu_i \ge 0$$

其中 λ_i 和 μ_i 是拉格朗日乘子，w、b 和 ξ_i 是主问题参数。

根据强对偶性，将对偶问题转换为：

$$\max_{\lambda,\mu}\min_{w,b,\xi} L(w,b,\xi,\lambda,\mu)$$

步骤 2：

分别对主问题参数w、b 和 ξ_i 求偏导数，并令偏导数为 0，得出如下关系：

$$w = \sum_{i=1}^{m}\lambda_i y_i x_i$$
$$0 = \sum_{i=1}^{m}\lambda_i y_i$$
$$C = \lambda_i + \mu_i$$

将这些关系带入拉格朗日函数中，得到：

$$\min_{w,b,\xi} L(w,b,\xi,\lambda,\mu) = \sum_{j=1}^{n}\lambda_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j (x_i \cdot x_j)$$

最小化结果只有 $\boldsymbol{\lambda}$ 而没有 $\boldsymbol{\mu}$，所以现在只需要最大化 $\boldsymbol{\lambda}$ 就好：

$$\max_{\lambda}[\sum_{j=1}^{n}\lambda_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\lambda_i\lambda_j y_i y_j (x_i \cdot x_j)]$$

$$s.t. \quad \sum_{i=1}^{n}\lambda_i y_i = 0, \quad \lambda_i \geq 0, \quad C - \lambda_i - \mu_i = 0$$

我们可以看到这个和硬间隔的一样，只是多了个约束条件。

转化一下表达形式，最终形式如下：

$$\min_{\boldsymbol{\alpha}} \ \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \left(\boldsymbol{x_i} \cdot \boldsymbol{x_j}\right) - \sum_{i=1}^{N}\alpha_i$$

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

$$0 \leq \alpha_i \leq C, \ i = 1, 2, \ldots, N$$

对于硬间隔，仅要求 $\alpha \geq 0$, 这是硬间隔和软间隔的区别。

考虑到要运用不同的核函数，$x_i \cdot x_j$ 应该改为 $\phi(x_i, x_j)$.

最终运用 `cvxopt.solvers.qp` 求解上面约束问题，`cvxopt.solvers.qp` 的格式如下：

标准形式：

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

用上面这种格式求解约束问题， 其中， $x = \alpha$.

步骤 3：

$$w = \sum_{i=1}^{m}\lambda_i y_i x_i$$

$$b = \frac{1}{|S|}\sum_{s \in S}(y_s - w x_s)$$

然后我们通过上面两个式子求出 w 和 b，最终求得超平面 $\boldsymbol{w^T x + b = 0}$，

代码如下：

```python
    '''
        根据训练数据train_data,train_label（均为np数组）求解svm,并对test_data进行预测,返回预测分数，即svm使用符号函数sign之前的值
        train_data的shape=(train_num,train_dim),train_label的shape=(train_num,) train_num为训练数据的数目，train_dim为样本维度
        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
    '''
    def fit(self,train_data,train_label,test_data):
        '''
        需要你实现的部分
        '''
        train_num = len(train_label)    # 标签数目
        # 先求 X 的核矩阵
        K = np.zeros((train_num, train_num))    # 核函数大小
        for i in range(train_num):
            for j in range(train_num):
                K[i][j] = self.KERNEL(train_data[i], train_data[j], kernel=self.kernel) # 计算核函数
        P = cvxopt.matrix(np.outer(train_label, train_label) * K)   # 标签已经 (-1,1)化， 计算P矩阵
        q = cvxopt.matrix(np.ones(train_num) * -1)   #为列向量，计算 q 矩阵
        A = cvxopt.matrix(train_label, (1, train_num), 'd')    #y的转置为行向量（1, train_num）表示排列为1* train_num 的矩阵
        b = cvxopt.matrix(0.0)
        #对于线性可分数据集
        if self.C is None:  # 对于硬间隔， 仅要求 向量 x >= 0，不过这里仅支持 <= 因此 G 改为 -1 的对角矩阵
            G = cvxopt.matrix(np.diag(np.ones(train_num) * -1))
            h = cvxopt.matrix(np.zeros(train_num))
        #对于软间隔
        else:
            arg1 = np.diag(np.ones(train_num) * -1)     # 对于软间隔，既要求 x >= 0, 也要求 x <= C
            arg2 = np.diag(np.ones(train_num))
            G = cvxopt.matrix(np.vstack((arg1,arg2)))   #加括号  因为vstack只需要一个参数，纵向堆叠
            arg1 = np.zeros(train_num)
            arg2 = np.ones(train_num) * self.C
            h = cvxopt.matrix(np.hstack((arg1,arg2)))   # 横向堆叠

        solution = cvxopt.solvers.qp(P, q, G, h, A, b)

        sol = np.ravel(solution['x'])   # 变成一个一维数组, alpha 的解

        if self.C is None:      # Epsilon为拉格朗日乘子阈值，低于此阈值时将该乘子设置为0
            sv = sol > self.Epsilon
        else:
            sv = (sol > self.Epsilon) * (sol < self.C)

        index = np.arange(len(sol))[sv]   #sol > 1e-15  返回true or false的数组   取出对应为 true 的下标

        self.alpha = sol[sv]  # 大于0对应的sol的值 index
        self.sv_X = train_data[sv]  # 支持向量的X index
        self.sv_y = train_label[sv]    # 支持向量的y  index

```

```python
        # 算 w 以及 b
        # 求 w
        self.w = np.zeros(len(train_data[0]))    # 总共 8 个特征
        for i in range(len(self.alpha)):
            self.w += self.alpha[i] * self.sv_y[i] * self.sv_x[i]
        print("W: ", self.w)
        # 求b
        self.b = 0
        for i in range(len(self.alpha)):
            self.b += self.sv_y[i]
            self.b -= np.sum(self.alpha * self.sv_y * K[index[i]][index])
        self.b /= len(self.alpha)   # 取平均
        print("b:", self.b)

        pred = np.dot(test_data, self.w) + self.b
        return pred
```

按照`cvxopt.solvers.qp`的格式，得到 P, q, G, h, A, b 这些矩阵，代入求得参数 α 的解，用该参数得到权值 W 和 b，从而得出分隔界面。

主代码如下：

```python
def main():
    # 加载训练集和测试集
    Train_data,Train_label,Test_data,Test_label=load_and_process_data()
    Train_label=[label[0] for label in Train_label]
    Test_label=[label[0] for label in Test_label]
    train_data=np.array(Train_data)
    test_data=np.array(Test_data)
    test_label=np.array(Test_label).reshape(-1,1)
    #类别个数
    num_class=len(set(Train_label))


    #kernel为核函数类型，可能的类型有'Linear'/'Poly'/'Gauss'
    #C为软间隔参数；
    #Epsilon为拉格朗日乘子阈值，低于此阈值时将该乘子设置为0
    kernel='Linear'
    # kernel = 'Gauss'
    # kernel = 'Poly'
    C = 1
    Epsilon=10e-5
    #生成SVM分类器
    SVM=SupportVectorMachine(kernel,C,Epsilon)

    predictions = []
    #one-vs-all方法训练num_class个二分类器
    for k in range(1,num_class+1):
        #将第k类样本label置为1，其余类别置为-1
        train_label=svm_label(Train_label,k)
        # 训练模型，并得到测试集上的预测结果
        prediction=SVM.fit(train_data,train_label,test_data)
        predictions.append(prediction)
    predictions=np.array(predictions)
    print(predictions)
    #one-vs-all，最终分类结果选择最大score对应的类别
    pred = np.argmax(predictions,axis=0)+1
```

```python
    pred = np.array(pred).astype(int).reshape(-1, 1)

    # 计算准确率Acc及多分类的F1-score
    print("Acc: "+str(get_acc(test_label,pred)))
    print("macro-F1: "+str(get_macro_F1(test_label,pred)))
    print("micro-F1: "+str(get_micro_F1(test_label,pred)))
```

对于 `Linear` 核函数，预测效果最好，结果如下：

```
D:\Anaconda\envs\pytorch\python.exe D:/Pycharm/python_project/AI_Lab/Lab2/src1/SVM.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
     pcost       dcost       gap    pres   dres
 0: -1.4159e+03 -9.7614e+03  6e+04  3e+00  3e-13
 1: -9.4986e+02 -6.5633e+03  1e+04  4e-01  2e-13
 2: -9.0554e+02 -3.5160e+03  4e+03  1e-01  2e-13
 3: -9.5053e+02 -1.6024e+03  8e+02  3e-02  2e-13
 4: -1.0444e+03 -1.2923e+03  3e+02  8e-03  2e-13
 5: -1.0729e+03 -1.2298e+03  2e+02  4e-03  2e-13
 6: -1.0917e+03 -1.1902e+03  1e+02  2e-03  2e-13
 7: -1.1024e+03 -1.1692e+03  7e+01  1e-03  2e-13
 8: -1.1119e+03 -1.1517e+03  4e+01  7e-04  2e-13
 9: -1.1162e+03 -1.1438e+03  3e+01  4e-04  2e-13
10: -1.1203e+03 -1.1364e+03  2e+01  2e-04  2e-13
11: -1.1227e+03 -1.1328e+03  1e+01  1e-04  2e-13
12: -1.1246e+03 -1.1300e+03  6e+00  4e-05  2e-13
13: -1.1261e+03 -1.1280e+03  2e+00  6e-06  2e-13
14: -1.1266e+03 -1.1275e+03  9e-01  2e-06  2e-13
15: -1.1270e+03 -1.1270e+03  5e-02  6e-09  2e-13
16: -1.1270e+03 -1.1270e+03  2e-03  2e-10  2e-13
17: -1.1270e+03 -1.1270e+03  4e-05  3e-12  2e-13
16: -1.1270e+03 -1.1270e+03  2e-03  2e-10  2e-13
17: -1.1270e+03 -1.1270e+03  4e-05  3e-12  2e-13
17: -1.1270e+03 -1.1270e+03  4e-05  3e-12  2e-13
Optimal solution found.
W:  [ 0.40478879 -4.35037711 -4.39807734 -2.30781914 -1.84422463  3.84468052
 -0.35786308 -4.22365232]
b: 3.112680870918066
     pcost       dcost       gap    pres   dres
 0: -3.0380e+03 -1.0857e+04  5e+04  3e+00  6e-13
 1: -2.0875e+03 -7.9495e+03  7e+03  1e-01  4e-13
 2: -2.3734e+03 -3.2502e+03  9e+02  2e-02  3e-13
 3: -2.5886e+03 -3.0175e+03  4e+02  7e-03  3e-13
 4: -2.6536e+03 -2.9271e+03  3e+02  4e-03  3e-13
 5: -2.6544e+03 -2.9264e+03  3e+02  4e-03  3e-13
 6: -2.6618e+03 -2.9250e+03  3e+02  3e-03  3e-13
 7: -2.6805e+03 -2.8981e+03  2e+02  2e-03  3e-13
 8: -2.6816e+03 -2.8990e+03  2e+02  2e-03  3e-13
 9: -2.7432e+03 -2.7953e+03  5e+01  4e-04  3e-13
10: -2.7541e+03 -2.7802e+03  3e+01  1e-04  3e-13
10: -2.7541e+03 -2.7802e+03  3e+01  1e-04  3e-13
11: -2.7602e+03 -2.7715e+03  1e+01  4e-05  4e-13
12: -2.7628e+03 -2.7681e+03  5e+00  2e-05  3e-13
13: -2.7642e+03 -2.7662e+03  2e+00  5e-06  4e-13
14: -2.7648e+03 -2.7655e+03  7e-01  1e-06  4e-13
15: -2.7651e+03 -2.7652e+03  7e-02  6e-08  4e-13
16: -2.7651e+03 -2.7651e+03  7e-03  6e-09  4e-13
17: -2.7651e+03 -2.7651e+03  6e-04  4e-10  4e-13
Optimal solution found.
W:  [-0.01916461  1.58418105  0.74690411  0.52981861 -0.77286988  1.0062085
 -0.19785329 -0.85507293]
b: -0.7204577986564743
     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  3e-13
 5: -1.8550e+03 -2.0397e+03  2e+02  2e-03  3e-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  3e-13
 9: -1.9125e+03 -1.9453e+03  3e+01  8e-05  3e-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.
W:  [-0.13479064 -1.65700058 -0.02007422  0.69086905  3.68062818 -6.31640081
  0.84284961  5.50435175]
b: -1.282768341685419
[[-3.76570337 -2.3211648  -1.76523693 ... -2.56669168 -3.20714542
  -2.59654939]
 [-0.13954666 -0.01952679  0.0147784  ...  0.02515487 -0.02609918
   0.05669682]
 [ 1.40898004  0.22075202 -0.26966504 ...  0.0865072   0.66714322
   0.02962966]]
Acc: 0.6581892166836215
score:  0.7678571428571428
score:  0.568733153638814
score:  0.6804123711340206
macro-F1: 0.6723342225433259
micro-F1: 0.6581892166836215
```

因为总共有 3 个不同的类别，因此要训练 3 个二分类器，由上图可以看出，总共有3组不同的 W 和 b 的值，对应 3 个不同的二分类器。

由上图可以看出 `Linear` 核函数的正确率在 65% 左右，正确率较高。

而对于 `Gauss` 核函数，准确率如下：

```
Acc: 0.3947100712105799
```

仅为 39.5% 左右，准确率较低。

对于 `Poly` 核函数(d = 2)，准确率如下：

```
Acc: 0.23499491353001017
```

仅为 23.5% 左右，准确率较低。

可见，对于不同的训练数据，适用的核函数也不一样，选择合适的核函数，对提高预测的正确率有很大帮助，例如本实验的数据集，比较适用的是 `Linear` 核函数。

SVM相较前面两个模型而言要相对复杂一些，对数据的处理和分析也要深入一些，因此准确率上有了一定的提升（65%左右）。

3. 深度学习

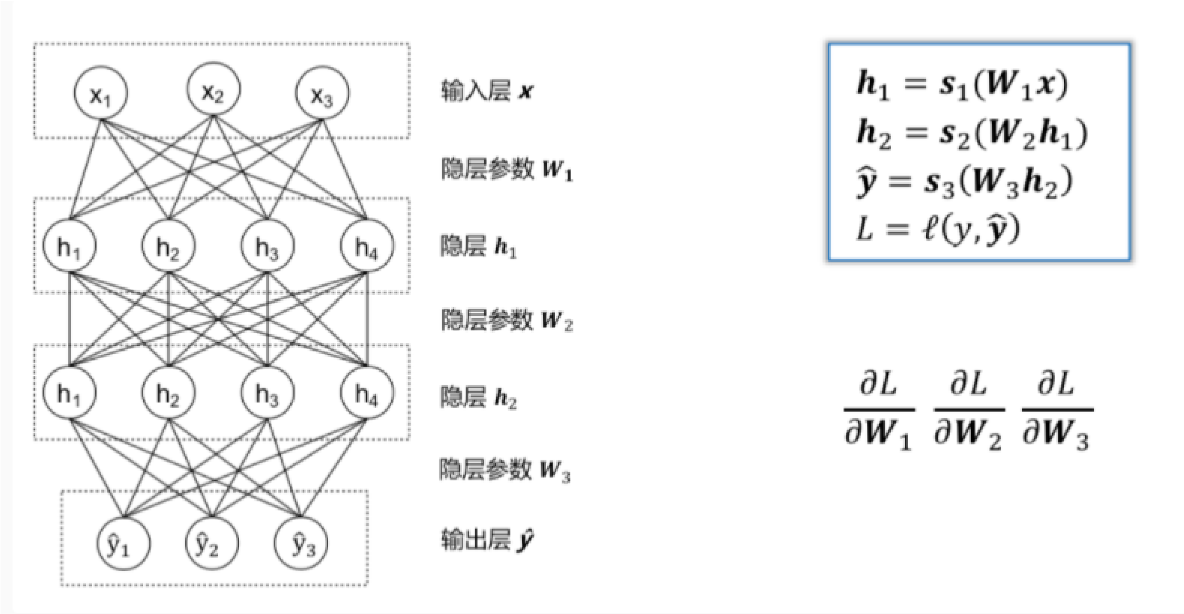
3.1 手写感知机模型并进行反向传播

实验目的：考察同学们对矩阵链式求导的掌握

实验内容：实现一个4层的感知机模型（隐层神经元设置为5， 4， 4， 3， 即输入的特征尾为5，输出的类别个数的3，激活函数设置为sigmoid）（1分）；实现BP算法（1分）；实现梯度下降算法（1分）。

实验要求：通过矩阵运算实现模型；实现各参数的梯度计算，给出各参数矩阵的梯度，并与pytorch自动计算的梯度进行对比；实现梯度下降算法优化参数矩阵，给出loss的训练曲线。

原理图：



$$
\begin{aligned}
\boldsymbol{h}_1 &= \boldsymbol{s}_1(\boldsymbol{W}_1\boldsymbol{x}) \\
\boldsymbol{h}_2 &= \boldsymbol{s}_2(\boldsymbol{W}_2\boldsymbol{h}_1) \\
\hat{\boldsymbol{y}} &= \boldsymbol{s}_3(\boldsymbol{W}_3\boldsymbol{h}_2) \\
L &= \ell(y, \hat{\boldsymbol{y}})
\end{aligned}
$$

$$\frac{\partial L}{\partial \boldsymbol{W}_1} = \left(\boldsymbol{W}_2^{\mathrm{T}}\left(\boldsymbol{W}_3^{\mathrm{T}}(\ell' \boldsymbol{s}_3')\odot \boldsymbol{s}_2'\right)\odot \boldsymbol{s}_1'\right)\boldsymbol{x}^{\mathrm{T}}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_2} = \left(\boldsymbol{W}_3^{\mathrm{T}}(\ell' \boldsymbol{s}_3')\odot \boldsymbol{s}_2'\right)\boldsymbol{h}_1^{\mathrm{T}}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_3} = (\ell' \boldsymbol{s}_3')\boldsymbol{h}_2^{\mathrm{T}}$$

梯度下降算法

$$\boldsymbol{W}_i = \boldsymbol{W}_i - \eta \frac{\partial L}{\partial \boldsymbol{W}_i}$$

$$\boldsymbol{s}_1 = \boldsymbol{s}_2 = \boldsymbol{\sigma}$$
$$\boldsymbol{\sigma}' = \boldsymbol{\sigma}(1 - \boldsymbol{\sigma})$$

$$\boldsymbol{s}_3(x_1, x_2, x_3) = \boldsymbol{Softmax}(x_1, x_2, x_3) = \frac{1}{e^{x_1} + e^{x_2} + e^{x_3}}(e^{x_1}, e^{x_2}, e^{x_3})$$

$$\ell(y, \hat{\boldsymbol{y}}) = CrossEntropy(y, \hat{\boldsymbol{y}}) = -\log \hat{y}_i \,, i = y$$

$$(\ell' \boldsymbol{s}_3')_i = \begin{cases} \hat{y}_i - 1, i = y \\ \hat{y}_i, i \neq y \end{cases}$$

代码如下：

初始化：

```python
class MLP_manual:

    def __init__(self, sizes, weights, biases, epochs=200):
        """
            初始化神经网络，给每层的权重和偏置赋初值
            权重为一个列表，列表中每个值是一个二维n×m的numpy数组
            偏置为一个列表，列表中每个值是一个二维n×1的numpy数组
        """
        self.num_layers = len(sizes)  # 神经网络层数
        # 构造神经网络权值矩阵
        self.weights = weights
        # 从第一层隐含层开始添加 偏置项
        self.biases = biases
        self.epochs = epochs # 迭代次数
```

按照 sizes 的格式搭建网络（按照题目要求，这里sizes = [5, 4, 4, 3]）。

激活函数相关：

```python
    """
        定义激活函数 sigmoid
    """

    def sigmoid(self, x):
        y = 1.0 / (1.0 + np.exp(-x))
        return y

    """
        激活函数 sigmoid 的导数
    """

    def sigmoid_back(self, x):
        # y = np.exp(-x) / (1.0 + np.exp(-x)) ** 2
        y = self.sigmoid(x) * (1 - self.sigmoid(x))
        return y
```

`softmax`:

```python
    """
        第3层的激活函数softmax
    """

    def softmax(self, X):       # 横向的， [3, 100] 这种格式
        # 输入X向量，输出Y向量
        # print(X.shape)
        X_T = X.transpose()
        Y = np.zeros((len(X_T),len(X_T[0])))
        for k in range(len(X_T)):
            c = 0.0
            for i in range(len(X_T[0])):
                c += np.exp(X_T[k][i])
                Y[k][i] = np.exp(X_T[k][i])
            Y[k] = 1 / c * Y[k]
        return Y.transpose()
```

按照图片中的定义实现:

$$s_3(x_1,x_2,x_3) = Softmax(x_1,x_2,x_3) = \frac{1}{e^{x_1}+e^{x_2}+e^{x_3}}(e^{x_1},e^{x_2},e^{x_3})$$

交叉熵:

```python
"""
    交叉熵CrossEntropy
"""

def CrossEntropy(self, y, y_p):
    y_pred = y_p.transpose()
    loss = np.zeros(len(y))
    sum = 0.0
    for i in range(len(y)):
        loss[i] = -math.log(y_pred[i][y[i] - 1])  # 按照交叉熵定义
        sum += abs(loss[i])
    sum = sum / len(y)
    return sum, loss
```

按照定义：

$$\ell(y,\hat{\mathbf{y}}) = CrossEntropy(y,\hat{\mathbf{y}}) = -\log \hat{y}_i\,, i = y$$

下面的代码求得上述原理图中的最内层一项：

$$(\ell' \boldsymbol{s}_3')_i = \begin{cases} \hat{y}_i - 1, i = y \\ \hat{y}_i, i \neq y \end{cases}$$

```python
"""
    交叉熵导数l' * s3'
"""

def nabla_ls(self, y, y_p):
    """
    :param y: 输入结果，监督学习，一维,类别从 1 开始
    :param y_pred: 预测结果，2维（第2维3个元素），注意，y_pred的下标从 0 开始
    :return: 导数乘积
    """
    y_pred = y_p.transpose()
    ans = np.zeros((len(y_pred), len(y_pred[0])))
    for k in range(len(y)):
        for i in range(len(y_pred[0])):
            if i + 1 == y[k]:
                ans[k][i] = y_pred[k][i] - 1
            else:
                ans[k][i] = y_pred[k][i]
    return ans.transpose()
```

前向传播代码：

```python
    """
        前向传播 feed_forward
    """

    def feed_forward(self, X):
        # 前向传播
        vec = X.transpose()sigmoid
        for i in range(len(self.weights) - 1):  # 前两层采用 sigmoid 激活
            vec = self.sigmoid(np.dot(self.weights[i], vec) + self.biases[i])
        # 最后一层采用 softmax 激活
        vec = self.softmax(np.dot(self.weights[len(self.weights) - 1], vec) + self.biases[len(self.weights) - 1])
        Y = vec  # 最终结果
        return Y
```

反向传播代码(BP算法):

```python
    """
        反向传播函数 feed_back
    """

    def feed_back(self, x, y):
        # y : 输入的结果标签，监督学习
        nabla_w = [np.zeros(w.shape) for w in self.weights]  # 用于计算导数，反向传播
        nabla_b = [np.zeros(b.shape) for b in self.biases]

        # 前向传播，计算各层的激活前的输出值以及激活之后的输出值，为下一步反向传播计算作准备
        activations = [x.transpose()]
        zs = []
        for i in range(len(self.weights) - 1):  # 前两层采用 sigmoid 激活
            z = np.dot(self.weights[i], activations[-1]) + self.biases[i]
            zs.append(z)
            activation = self.sigmoid(z)  # 激活函数
            activations.append(activation)
        z = np.dot(self.weights[len(self.weights) - 1], activations[-1]) + self.biases[len(self.weights) - 1]       # 最后一层采用 softmax 激活
        zs.append(z)
        activation = self.softmax(z)  # softmax
        activations.append(activation)

        # 先求最后一层的delta误差以及b和W的导数
        delta = self.nabla_ls(y, activations[-1])
        lossavg, loss = self.CrossEntropy(y, activations[-1])  # 损失函数值
        nabla_b[-1] = np.sum(delta, axis=1)/len(delta[0])    # 按行求平均
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # 将delta误差反向传播以及各层b和W的导数，一直计算到第二层
        for l in range(2, self.num_layers):
            delta = np.dot(self.weights[-l + 1].transpose(), delta) * self.sigmoid_back(zs[-l])
            nabla_b[-l] = np.sum(delta,axis=1)/len(delta[0])   # 按行求平均
            nabla_w[-l] = np.dot(delta, activations[-l - 1].transpose())
        return nabla_b, nabla_w, lossavg
```

先前向激活，后反向求梯度，求梯度的主要按照上面原理图中的：

$$\frac{\partial L}{\partial \boldsymbol{W}_1} = (\boldsymbol{W}_2^{\mathrm{T}}(\boldsymbol{W}_3^{\mathrm{T}}(\ell' \boldsymbol{s}_3')\odot \boldsymbol{s}_2')\odot \boldsymbol{s}_1')\boldsymbol{x}^{\mathrm{T}}$$
$$\frac{\partial L}{\partial \boldsymbol{W}_2} = (\boldsymbol{W}_3^{\mathrm{T}}(\ell' \boldsymbol{s}_3')\odot \boldsymbol{s}_2')\boldsymbol{h}_1^{\mathrm{T}}$$
$$\frac{\partial L}{\partial \boldsymbol{W}_3} = (\ell' \boldsymbol{s}_3')\boldsymbol{h}_2^{\mathrm{T}}$$

实现。

从最后一层开始求梯度(W_3开始)，每往内一层的梯度，都包括后面一层的梯度，比如

$$\frac{\partial L}{\partial W_2} = (W_3^T \frac{\partial L}{\partial W_3})h_1^T$$
$$\frac{\partial L}{\partial W_{j-1}} = (W_j^T \frac{\partial L}{\partial W_j})h_{j-2}^T$$

因此可以迭代实现求每个权值矩阵的导数。

梯度下降算法：

```
    """
        梯度下降
    """

    def gradient_descent(self, x, y, lr=0.001):
        """
        :param x:   输入
        :param y:   输出
        :param lr:  学习速率
        :return:    权值矩阵，训练好的
        """
        X = np.zeros(self.epochs)   # 用于作图
        loss = np.zeros(self.epochs)    # 用于作图
        for i in range(self.epochs):
            DNB, DNW, loss[i] = self.feed_back(x, y)    # 梯度
            print("权值w的梯度：  ", DNW)
            print("偏置项b的梯度：  ", DNB)
            X[i] = i
            self.weights = [w - lr / len(x) * nw for w, nw in
zip(self.weights, DNW)] # 梯度下降，这里梯度下降时要除以数据集大小
            self.biases = [b - lr * nb.reshape(b.shape) for b, nb in
zip(self.biases, DNB)] # 在反向传播中梯度已经求过平均，不需要除以数据集大小
        print("最终权值W:  ", self.weights)
        print("偏置项b: ", self.biases)
        return X, loss  # 用于作图
```

原理图：

梯度下降算法

$$W_i = W_i - \eta \frac{\partial L}{\partial W_i}$$

下面是用 torch 实现的用于对比的 MLP：

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        # 使用父类的初始化参数
        self.mlp = nn.Sequential(    # 搭建网络
            nn.Linear(5, 4),
            nn.Sigmoid(),
            nn.Linear(4, 4),
            nn.Sigmoid(),
            nn.Linear(4, 3),
            # nn.Softmax(1)  # torch.nn.CrossEntropyLoss() 已有 Softmax层，不需要重复定义
        )
        # 定义神经网络里的输入、隐藏和输出层

    def initial(self, weights, biases):      # 初始化权值和偏置项，使得torch搭建的网络的初始值和手写MLP的初始值一致
        i = 0
        for layer in self.mlp:
            if isinstance(layer, nn.Linear):  # 判断是否是线性层
                layer.weight.data = torch.from_numpy(weights[i])  # double类型
                layer.bias.data = torch.from_numpy(biases[i])
                i = i + 1

    def forward(self, x):
        # 前向传播
        y_pred = self.mlp(x)
        return y_pred
```

讲解见注释。

画图比较程序：

```
import matplotlib.pyplot as plt

```

```python
plt.rcParams['font.sans-serif'] = ['SimHei']  # 显示中文标签
plt.rcParams['axes.unicode_minus'] = False    # 这两行需要手动设置


def DrawDiagram(x, y, x2, y2, path):
    # plt.scatter(x, y, s=10, c='blue')  # 将每个规模和对应的运行时间的对数的散点在图中描出来
    # for a, b in zip(x, y):
    #     plt.text(a, b, (a, b), ha='right', va='bottom', fontsize=10, color='r', alpha=0.5)  # 给这些散点打上标记
    plt.plot(x, y, c='blue', label='manual')  # 描绘出光滑曲线
    plt.plot(x2, y2, c='red', label='torch')  # 描绘出光滑曲线
    plt.legend(loc=1)  # 指定legend图例的位置为右下角
    plt.title("loss损失函数", fontsize=18)  # 标题及字号
    plt.xlabel("迭代次数 n", fontsize=15)  # X轴标题及字号
    plt.ylabel("loss", fontsize=15)  # Y轴标题及字号
    plt.tick_params(axis='both', labelsize=14)  # 刻度大小
    plt.xticks(np.arange(0, 201, 20))
    plt.yticks(np.arange(1.0, 2.1, 0.1))
    plt.savefig(path)
    plt.show()
```

主代码:

```python
def main():
    sizes = [5, 4, 4, 3]    # 网络规格
    X = np.random.randn(100, 5) # 随机生成训练数据
    Y = np.random.randint(1, 4, 100)    # 打上标签
    print("X: ", X)
    print("Y: ", Y)
    W = [np.random.randn(n, m) for m, n in zip(sizes[:-1], sizes[1:])]
    B = [np.random.randn(n, 1) for n in sizes[1:]]
    learning_rate = 0.01    # 学习速率
    EPOCH = 200     # 迭代次数
    path = '../photos/loss.png'
    NET = MLP_manual(sizes=sizes, weights=W, biases=B)  # 手写网络
    xc, yc = NET.gradient_descent(X, Y, learning_rate)  # BP,梯度下降

    # 生成随机数当作样本，同时用Variable 来包装这些数据，设置 requires_grad=False 表示在方向传播的时候，
    # 我们不需要求这几个 Variable 的导数
    X_torch = Variable(torch.from_numpy(X))     # 转化格式
    Y_torch = Variable(torch.from_numpy(Y) - 1).long()   # 转化格式

    net_torch = MLP()   # torch网络
    B_torch = [np.random.randn(n) for n in sizes[1:]]   # 匹配格式
    for i in range(len(B)):
        B_torch[i] = B[i].reshape(B_torch[i].shape)
    net_torch.initial(W, B_torch)

    # 定义损失函数
    loss_fn = torch.nn.CrossEntropyLoss()

    # 使用optim包来定义优化算法，可以自动的帮我们对模型的参数进行梯度更新。这里我们使用的是随机梯度下降法。
```

```
    # 第一个传入的参数是告诉优化器，我们需要进行梯度更新的Variable 是哪些，
    # 第二个参数就是学习速率了。
    optimizer = torch.optim.SGD(net_torch.parameters(), lr=learning_rate)

    # 开始训练
    xc_t = xc
    yc_t = np.zeros(EPOCH)
    for t in range(EPOCH):
        # 向前传播
        y_pred = net_torch.forward(X_torch)
        # 计算损失
        loss = loss_fn(y_pred, Y_torch)
        # 显示损失
        yc_t[t] = loss
        # 在我们进行梯度更新之前，先使用optimier对象提供的清除已经积累的梯度。
        optimizer.zero_grad()
        # 计算梯度
        loss.backward()
        # 更新梯度
        optimizer.step()
    DrawDiagram(xc, yc, xc_t, yc_t, path)   # 画图
```

得到的最终结果如下：

由于迭代次数较多（200次），只截取最后一次梯度和最终的权值矩阵W以及偏置项b，**助教们可以自行运行代码观察每次迭代时各参数矩阵的梯度**：

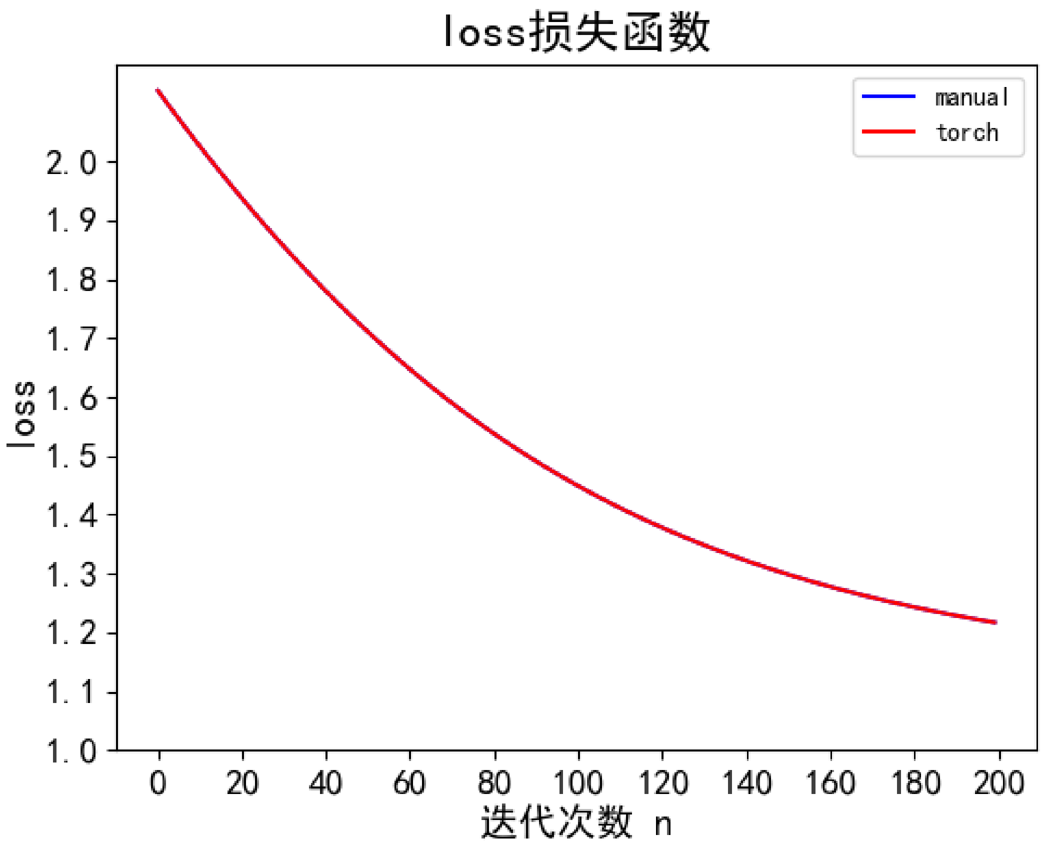
```
x:  [[ 0.83488902  0.23153961  0.47533387 -1.21848032 -0.12007636]
 [ 1.46939798  0.0572719  -0.11851576  1.46267249 -0.88854435]
 [-1.33892934  2.09351508 -0.04852904 -1.11790677 -0.91252277]
 [-0.83430126 -1.01072233  0.67728023  0.38879291  0.00997398]
 [-0.44426901  2.78759152  0.47503129  0.68527825  0.45899218]
 [ 0.5925791   0.48603452  1.72485288  0.03516986  0.56736816]
 [-0.98652551  0.91365496 -1.17384462 -0.48263474  0.05925806]
 [ 1.05871913  1.1584931  -1.38860985  1.99327577 -0.10163657]
 [-0.92296307  0.09511347 -0.11437427 -0.01399916 -0.86625038]
 [-0.73594553 -0.07961658  0.15887615  1.17251506 -0.59960658]
 [ 1.36096773 -0.68740978  1.9075558   0.32084597  0.18737522]
 [ 1.4600593   1.20094358 -1.16325907 -1.83679572  0.21242848]
 [ 1.24087443 -0.51662119 -0.15292404  0.51309608 -0.4838295 ]
 [-0.91712725  0.45778383 -1.56769885 -1.25781159  0.85980353]
 [-0.72741193  0.11332112  0.73887247 -0.1818219   0.49531001]
 [-0.67100722 -0.41890987 -0.71829914 -1.17451857 -0.96541388]
 [-1.05151839 -0.38455553 -0.68007927  1.53509849 -0.6125221 ]
 [-0.92577306 -1.18321932  0.95840937  1.02806786 -0.4828016 ]
 [-0.00405364  0.64654168  0.09395961  1.56248195 -0.29150326]
 [-1.45469118  0.05549694 -0.50610932  1.3091876  -1.804321  ]
 [ 1.26335834 -0.60921331 -0.02984466 -0.35131276  1.38513603]
 [ 0.07194634  0.08571002  0.89928942 -0.02861828  0.1482392 ]
 [-0.49554878 -0.02741953 -0.79775581 -1.19562597  0.72389209]
 [-0.8068407   1.34443341 -0.93972469 -0.49152879  1.11094768]
 [ 1.00481427  0.39454128 -0.26097344  0.0064474  -0.21818126]
 [ 0.51591462 -0.55020014 -0.58719097 -0.18258468 -0.31805762]
 [ 0.15453501  0.9308792  -0.50568949  0.39281427  0.33670162]
 [ 0.24737016 -1.74926382 -1.41077325 -0.61583215  1.08322424]
 [ 0.05985057 -0.61235343 -1.36876396  1.0432108   0.43755955]
 [-0.34717722 -0.06736756 -0.71021974 -0.16022278  1.73251906]
```

```
 [-0.30545703 -0.79291187 -0.05042232  1.46302357  1.31634629]
 [ 0.80906558 -0.44545464  0.30197889  0.85266227  1.4135824 ]
 [ 0.39910983  0.2930576  -1.57178504 -0.58202383 -0.90844584]
 [-0.65474553  0.48472197  0.38399148 -0.71781543  1.1359159 ]
 [-0.25933595  0.70738538  0.63409583  0.33394601  0.29658603]
 [-0.67487638 -1.24072435  0.36099152 -0.33308361  2.04059057]
 [ 0.03077101  0.01992702 -0.12177077  0.16637707 -0.43614148]
 [-0.1389848  -0.4380541  -0.24714162  0.17304385 -1.29752049]
 [-0.2894404   0.51771372 -0.64274378  1.58943955  2.24999956]
 [-0.84283755 -1.30959921 -2.20747659  1.88037881 -0.74525005]
 [ 0.91980744 -0.88277919  0.1296434   0.6513306   0.90430835]
 [-1.05195288  0.3884025   0.16345633 -0.47773476 -0.50655469]
 [-1.64492953  0.38402737 -0.56450409  0.25923263 -0.66352488]
 [-0.54720276 -0.79375868 -1.08199372 -0.49031401  0.65993611]
 [-1.77944923  0.67767416 -0.75329668 -1.08209539  0.35314374]
 [ 0.01255226  0.08611852  1.85928414 -0.22906634 -0.79700699]
 [ 1.05998687  0.12599458 -1.27207005  0.16512147 -2.77614823]
 [-1.35911223  0.14094921 -0.04387857 -0.44095405 -0.22483604]
 [ 1.36502208  0.67842152  0.96127413 -0.59425554 -0.66592134]
 [-1.94347734 -1.24152054  0.23080841  1.10825441  1.19377116]
 [-0.19568659 -1.25432844 -1.47525288 -0.24181263  1.55157866]
 [-1.07648927 -1.7915057   0.69953044 -0.42698437  1.24217932]
 [ 0.12461681 -0.5337466   0.77454648  0.25960149 -0.21117116]
 [-0.16006732  1.63375352  0.36330031  0.31874503 -0.75470043]
 [ 0.40505691  1.62287383 -0.01843553  0.86973426 -0.73617225]
 [-0.60725595  0.82773987 -2.44693556  0.27921946 -0.73187145]
 [-0.77863375  1.1332869  -0.90196955  0.7376919   1.59447287]
 [ 0.53561842 -0.43851007  0.29036594 -0.29608347  1.35394095]
 [-1.21755867 -0.10473705 -0.1065995  -0.4331435   0.32436681]
 [-1.14196726 -0.61148465  0.16062321 -1.06002281 -0.7875055 ]
 [ 2.27506059 -0.699645   -1.27464986 -0.20864239 -0.05991806]
 [ 0.59778166  0.80064544 -0.23148685  0.7459596   0.01526342]
 [-1.88345263  1.01019894  0.17020846 -1.04226723  1.67630496]
 [ 0.68994458  1.36354683 -0.09806423  0.67724152  1.56380222]
 [ 0.57309639 -2.57008741 -0.71982032  0.27305772  0.57856456]
 [-0.75093116  0.46262167 -0.18470568 -0.34679018 -0.7249639 ]
 [ 2.55531317 -0.85922591  2.09573011  0.90325367 -0.5335606 ]
 [-0.6647763   0.13846687 -0.9688558  -0.85244734 -0.33094539]
 [-0.08363142 -1.44322213  0.46293876 -0.02901496  1.57121849]
 [ 1.1638642  -1.67135351  0.30268492 -0.23243489 -0.57318673]
 [ 1.34767626 -0.27695519 -0.44517173  1.33464624 -0.26347442]
 [ 0.65206771  0.64022523  0.05438969 -0.94011253  0.23306771]
 [ 0.95372861  0.96658815  1.835293   -0.5537257  -1.20392265]
 [ 1.47330793  0.1839157   0.67840075 -0.95921284 -0.66205136]
 [ 0.68597773  0.05463347  1.31434039 -1.01076823 -1.95869872]
 [-0.39554742  1.48644499 -0.38480679 -1.03568207 -1.59286306]
 [ 0.41868069 -0.0570767  -0.37318524 -0.46710579  1.62038416]
 [ 1.02077776  0.36728826  0.63079586  1.05112724  0.09742864]
 [ 1.26875192  0.20011602 -0.98482161 -0.12422853 -1.35121384]
 [ 0.93417313  1.54022861  1.70707617  1.56840633  0.75448529]
 [-0.38310039 -0.69744404  0.0868327  -0.12513525 -1.61589297]
 [-0.30237824  0.54262569  0.84495467  0.39760994  0.76918821]
 [-0.48807899  0.91490061 -0.63953081 -1.37548514 -0.39800088]
 [-0.81124514  0.70753692  0.71943975 -0.67038897  0.60230753]
 [ 1.4186475  -0.75374606 -0.39168169 -1.64850674  0.09138435]
 [ 0.05816796  0.76210707 -0.40751329  0.395935    0.48017597]
 [-1.18190429  0.52440846 -3.38546547 -0.00822893  0.46150844]
 [-0.03635531  0.2754694   0.79363126  0.92771058 -0.1228405 ]
```

```
 [-0.74848184  1.28590379 -0.18823094  0.48195931  3.18212108]
 [ 0.23794026  1.34745379 -0.45349018  0.02875414 -1.89538894]
 [ 1.11318316 -1.09723136  0.22057002 -0.78707496  0.9473564 ]
 [-1.13945942  0.33214101 -0.43392607  0.99396153 -0.71415538]
 [ 1.01636593 -0.95312132 -0.53154838  0.16627817 -0.50240594]
 [-0.69315551  0.90419522 -0.6373801  -0.42690054 -0.65709746]
 [-0.11676204 -0.41941862  0.39561197 -1.77547944 -0.23830797]
 [-0.40782769  1.66468182 -0.14801752 -0.03445689  0.56181128]
 [ 1.86664401  0.76248674 -0.71699224 -0.33820584 -0.31409013]
 [-0.35621352  1.15052135 -1.44676619  0.51090262  0.25032605]
 [-0.06722164  0.69194487  0.21703902  1.47006379  0.40017461]
 [-0.13778534 -0.15082999 -0.31770692  1.78973517  0.21038692]]
Y:  [2 2 2 1 3 1 1 2 1 1 1 2 3 3 2 2 1 2 3 2 2 1 2 3 2 2 2 2 3 3 3 3 3 2 2 1 3
 3 1 1 2 1 3 3 3 1 3 1 1 2 1 1 3 1 1 3 2 1 3 1 3 1 1 1 2 3 1 3 1 3 1 2 1 2
 2 2 1 2 3 2 1 1 3 1 2 1 2 1 3 3 3 3 3 2 3 2 2 2 1 3]
```

```
权值W的梯度：  [array([[-0.15546151, -0.0580196 , -0.51497458,  0.12058515, -0.81256573],
       [-0.02333019, -0.4593307 ,  0.60582536,  0.19337429,  0.41685792],
       [ 0.21162927,  0.41658104,  0.14120357, -0.14032193,  0.03486692],
       [-1.0842209 , -1.15885873,  0.63715942,  0.54083536,  0.36304542]]), array([[ 2.7051578 ,  5.19139402,  1.80477424,  3.37608263],
       [-0.63712428,  0.0953337 ,  0.74677931, -0.75000792],
       [-0.05424791,  0.26632508,  0.26158023, -0.03319106],
       [ 1.87312406,  1.3810832 , -0.32665619,  2.84440685]]), array([[-9.23548807,  0.10860248, -0.77053945, -5.00866463],
       [12.42307669,  2.56853286,  2.12997982,  8.01176338],
       [-3.18758862, -2.67713534, -1.35944037, -3.00309876]])]
偏置项b的梯度：  [array([ 0.00846461, -0.00156768, -0.0042308 ,  0.01772123]), array([0.0728672 , 0.00822937, 0.00558353, 0.01259409]), array([-0.12105603, 0.21805069, -0.09699466])]
最终权值W：  [array([[-1.43018682, -1.02963234,  2.36857808,  0.1187577 , -2.00941532],
       [ 0.98681826, -1.23083911, -0.02927504,  1.62847881, -0.94336185],
       [ 0.57425908, -0.46033831, -0.22610129, -1.57089737,  0.18417166],
       [ 0.95775996,  1.56097356,  2.93832804, -0.12252879,  0.03707383]]), array([[ 0.72175533,  0.16840664, -0.68115176,  1.73409849],
       [-0.04429821, -2.01708744,  1.46791251, -1.67160732],
       [ 0.87243134, -0.53495334,  0.1763126 , -1.52595714],
       [-0.19657111, -0.6942152 ,  0.74182962,  1.07316614]]), array([[ 0.0291497 ,  1.77864602, -0.02791594, -0.95259736],
       [ 0.68534689,  0.30219509, -0.30215346,  0.14663755],
       [-1.91920954, -0.62491014, -0.95370465,  1.10429903]])]
偏置项b：  [array([[-1.02805667],
       [ 1.49060832],
       [-1.04725462],
       [ 1.4157048 ]]), array([[-1.2622664 ],
       [ 0.11161132],
       [-1.32070885],
       [-0.82040383]]), array([[-0.54969045],
       [-0.14876096],
       [ 0.09316995]])]
```

以及loss的训练曲线：



可见我们实现的MLP模型和 torch 内部本身的MLP模型是一样的，在loss损失函数的下降图像上完全一致，得到的梯度和最终得到的权值矩阵W和偏置项b也完全一致（**用图像来表达要直观一些，因此直接用图像来对比两者的梯度**）。

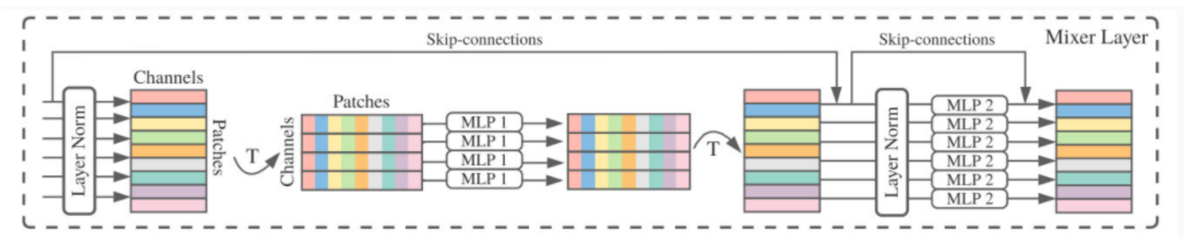
3.2 复现MLP-Mixer

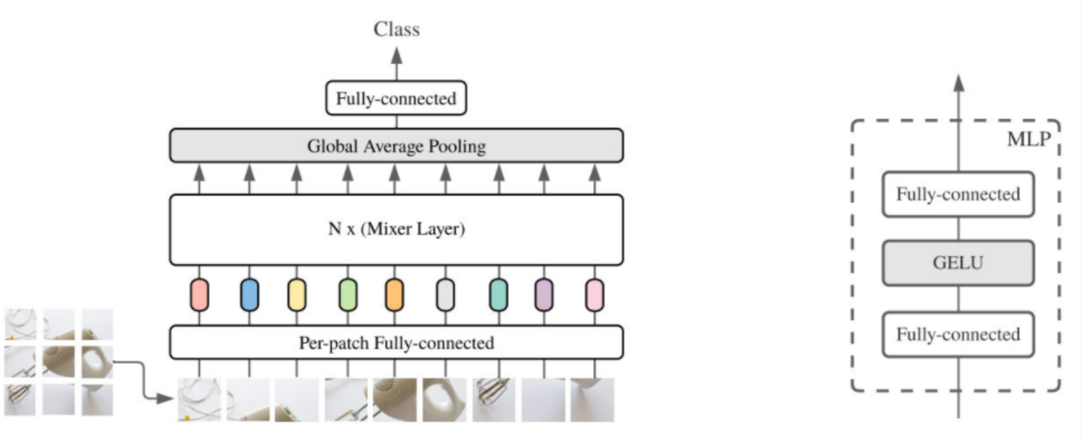
实验目的：对深度学习的初步掌握，仅使用最基础的多层感知机。考察自行搜索相关资料学习的能力。

实验内容：复现MLP-Mixer模型，并在MNIST数据集上进行测试（模型可以自行搜索各种博客，论文）。

数据集介绍：数据集由60000行的训练数据集（trainset）和10000行的测试数据集（testset）组成，包含从0到9的手写数字图片，如下图所示，分辨率为28*28。每一个MNIST数据单元有两部分组成：一张包含手写数字的图片和一个对应的标签（对应代码文件中的data和target）。

实验要求：可以使用torch的所有功能。模型的参数自定。仅可以在注释的方框中书写你的代码， 不能修改其他代码，不能超出方框外书写。报告中需要贴上终端输出的截图。





对于 MLP-Mixer 的理解：

Mixer Layer层利用了两种MLP层：

- channel-mixing MLPs：允许不同channels特征之间的交流；
- token-mixing MLPs：允许不同空间位置之间的交流。
- 这两个MLP层是交错的。

对于整个 MLP-Mixer：

- Per-patch Fully-connected 相当于是embedding层。
- Mixer Layer 是文章提出的主要创新结构。其中，**每一个Mixer Layer包含一个token-mixing MLP 和一个channel-mixing MLP**，这两个结构都是由**两个全连接层和GELU激活函数组成。**
- 在 Mixer Layer 层中：
    1. 矩阵先经过Layer Norm，相当于是先进行了归一化；
    2. 然后矩阵经过转置；
    3. 经过第一个全联接层，这个MLP应该就是channel-mixing了；
    4. 然后再次转置，再进行Layer Norm；
    5. 然后是token-mixing channels层；
    6. 中间加了两个skip connection，连接输入和输出。

代码如下：

```
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.layer_norm1 = nn.LayerNorm(hidden_dim)
        self.mlp1 = nn.Sequential(  # 第一个全连接层channel-mixing
            nn.Linear((28 // patch_size) ** 2, 256),
            nn.GELU(),
            nn.Linear(256, (28 // patch_size) ** 2)
        )
        self.layer_norm2 = nn.LayerNorm(hidden_dim)
        self.mlp2 = nn.Sequential(  # 第2个全连接层token-mixing
            nn.Linear(hidden_dim, 2048),
            nn.GELU(),
            nn.Linear(2048, hidden_dim)
        )
        ########################################################################

```

```
    def forward(self, x):
        ########################################################################
        y = self.layer_norm1(x) # layernorm
        y = y.transpose(1, 2)   # 转置
        y = self.mlp1(y)     # 第一个全连接层
        y = y.transpose(1, 2)       # 转置回来
        x = x + y      # skip_connection

        y = self.layer_norm2(x)
        y = self.mlp2(y)
        return x + y   # skip_connection
        ########################################################################

```

这一段代码主要介绍的是 MLP-Mixer 主体结构 Mixer Layer层，和上面叙述的一样，一个 Mixer Layer层，主要的神经网络流图是：

Layer Norm -> channel-mixing MLP -> Layer Norm -> token-mixing MLP

因此，在 `__init__` 中，定义了 4 个主要的网络，对应上面的 4 层网络。

`self.mlp1` 中的`nn.Linear`全连接层中的某个参数为 `(28 // patch_size) ** 2`，这是因为总共有这么多的图片分片(一张图片分片的大小为 $(patch\_size)^2$ 个像素，而原图片大小为 28^2 个像素)，`Linear` 层将所有的图片分片作为输入，因此参数为 `(28 // patch_size) ** 2`。

其他数值参数(256，2048)的设置是在参考了网上的代码，以及自己的调试尝试后发现的能使准确率维持比较高的数值。

整体 MLP-Mixer 如下：

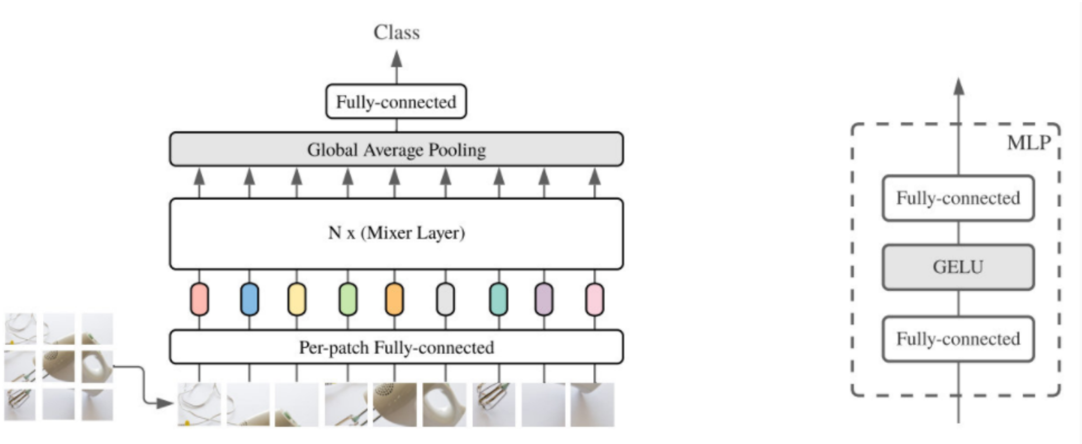
```
class MLPMixer(nn.Module):
    def __init__(self, patch_size, hidden_dim, depth):
        super(MLPMixer, self).__init__()
        assert 28 % patch_size == 0, 'image_size must be divisible by patch_size'
        assert depth > 1, 'depth must be larger than 1'
        ########################################################################
        #这里写Pre-patch Fully-connected, Global average pooling, fully connected

        #  Per-patch Fully-connected 相当于 embedding ( 嵌入 ) 层
        self.embedding = nn.Conv2d(1, hidden_dim, kernel_size=patch_size, stride=patch_size)
        mix_layer = Mixer_Layer(patch_size, hidden_dim) # Mixer Layer
        self.mixer_layers = nn.Sequential(*[mix_layer for _ in range(depth)]) # n * Mixer Layer
        self.norm = nn.LayerNorm(hidden_dim)    # 归一化
        self.cls = nn.Linear(hidden_dim, 10)    # 最后一个全连接层预测类别， 共10类
        ########################################################################


```

```
    def forward(self, data):
        ########################################################################
        #从第2个维度开始展开，将后面的维度转化为一维，只保留2之前的维度，其他维度的数据全都挤在2这一维
        emb_out = self.embedding(data).flatten(2) # 将下标为 2(包含) 以后的矩阵维度合并
        emb_out = emb_out.transpose(1, 2)
        mix_out = self.mixer_layers(emb_out)    # 经过 n 个 Mixer Layer层
        norm_out = self.norm(mix_out)   # 归一化
        data_avg = torch.mean(norm_out, dim=1)  # 逐通道求均值 Global Average Pooling
        c = self.cls(data_avg)  # 全连接层预测
        return c
        ########################################################################
```

整体架构的实现主要是参考下面这张图片：



按照这张图片，搭建整体架构，每一层对应的代码由注释标注。

训练函数：

```
def train(model, train_loader, optimizer, n_epochs, criterion):
    model.train()
    for epoch in range(n_epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to(device), target.to(device)
            ########################################################################
            #计算loss并进行优化
            output = model(data)
            loss = criterion(output, target)
            # 在我们进行梯度更新之前，先使用optimier对象提供的清除已经积累的梯度。
            optimizer.zero_grad()
            # 计算梯度
            loss.backward()
            # 更新梯度
            optimizer.step()
            ########################################################################
            if batch_idx % 100 == 0:
                print('Train Epoch: {}/{} [{}/{}]\tLoss: {:.6f}'.format(
```

```
                        epoch, n_epochs, batch_idx * len(data), len(train_loader.dataset), loss.item()))
```

调用 `criterion` 计算损失，调用 `backward` 更新梯度， 调用 `optimizer.step()` 更新参数，通过反向传播更新 权值W 和 偏置项b。

测试函数：

```
def test(model, test_loader, criterion):
    # print("test: ", test_loader)
    model.eval()
    test_loss = 0.
    num_correct = 0 #correct的个数
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
 ########################################################################
        #需要计算测试集的loss和accuracy
            out = model(data)
            loss = criterion(out, target)
            value_max, pred = torch.max(out, 1)     # 按行索引，得到每行最大值及其索引
            test_loss += loss.data * len(target)    # len(target) : 预测图片数量
            for i in range(len(target)):    # 统计预测正确的元素个数
                if pred[i] == target[i]:
                    num_correct += 1

        totalnum = 0      # 统计预测的总数量
        for _, target in test_loader:
            target = target.to(device)
            totalnum += len(target)
        # totalnum = 10000 按照题目所给条件，最终结果应该为 10000
        test_loss = test_loss / totalnum
        accuracy = num_correct / totalnum

 ########################################################################
        print("Test set: Average loss: {:.4f}\t Acc {:.2f}".format(test_loss.item(), accuracy))
```

前面的每次迭代循环中， 每次 target 中的测试集标签数量均为 `batch_size = 128`，仅循环最后一次的标签数量可能不足 `batch_size` 的大小。

从 out 中，得到每一个元素(包含10个概率，分别表示 0 - 9 的概率大小)中，最大概率的下标，作为该元素的预测结果。

用 `loss.data * len(target)` 得到这一轮迭代的总损失loss，加到 test_loss 中，用于求得最终的测试集平均损失。

另外，要计算下测试数据集的大小totalnum，用于求得平均测试集损失和准确率。**或者也可直接用题目所给条件：测试集的大小为10000行。**

主代码如下：

```python
if __name__ == '__main__':
    n_epochs = 5
    batch_size = 128
    learning_rate = 1e-3

    transform = transforms.Compose(
        [transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))])

    trainset = MNIST(root = './data', train=True, download=True, transform=transform)
    train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2, pin_memory=True)

    testset = MNIST(root = './data', train=False, download=True, transform=transform)
    test_loader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=2, pin_memory=True)

    ########################################################################
    model = MLPMixer(patch_size=4, hidden_dim=256, depth=8).to(device) # 参数自己设定，其中depth必须大于1
    # 这里需要调用optimizer，criterion(交叉熵)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
    ########################################################################

    train(model, train_loader, optimizer, n_epochs, criterion)
    test(model, test_loader, criterion)
```

选定交叉熵作为损失函数。

最终结果如下：

```
D:\Anaconda\envs\pytorch\python.exe D:/Pycharm/python_project/AI_Lab/Lab2/src2/MLP_Mixer.py
Train Epoch: 0/5 [0/60000]  Loss: 2.358246
Train Epoch: 0/5 [12800/60000]  Loss: 2.245153
Train Epoch: 0/5 [25600/60000]  Loss: 2.129940
Train Epoch: 0/5 [38400/60000]  Loss: 1.793247
Train Epoch: 0/5 [51200/60000]  Loss: 1.453742
Train Epoch: 1/5 [0/60000]  Loss: 1.345111
Train Epoch: 1/5 [12800/60000]  Loss: 1.063033
Train Epoch: 1/5 [25600/60000]  Loss: 0.873148
Train Epoch: 1/5 [38400/60000]  Loss: 0.756925
Train Epoch: 1/5 [51200/60000]  Loss: 0.732408
Train Epoch: 2/5 [0/60000]  Loss: 0.607281
Train Epoch: 2/5 [12800/60000]  Loss: 0.552570
Train Epoch: 2/5 [25600/60000]  Loss: 0.554707
Train Epoch: 2/5 [38400/60000]  Loss: 0.382511
Train Epoch: 2/5 [51200/60000]  Loss: 0.461305
Train Epoch: 3/5 [0/60000]  Loss: 0.423186
Train Epoch: 3/5 [12800/60000]  Loss: 0.395927
Train Epoch: 3/5 [25600/60000]  Loss: 0.368498
Train Epoch: 3/5 [38400/60000]  Loss: 0.431208
Train Epoch: 3/5 [51200/60000]  Loss: 0.233815
Train Epoch: 4/5 [0/60000]  Loss: 0.344410
Train Epoch: 4/5 [12800/60000]  Loss: 0.261858
Train Epoch: 4/5 [25600/60000]  Loss: 0.237520
Train Epoch: 4/5 [38400/60000]  Loss: 0.308428
Train Epoch: 4/5 [51200/60000]  Loss: 0.247133
Test set: Average loss: 0.2439   Acc 0.93
```

准确率达到了惊人的 93%！ 可见该模型效果很好，而且 93% 并不是该模型的最好测试结果，通过修改参数，还能更好地提高模型的性能与准确度，不过相应的，会对硬件资源有更高的要求。

该模型综合运用了CNN 和 self-addition 模型，并经过了多层 `Mixer Layer`的训练，因此效果很好。