人工智能 LAB2 实验报告

和泳毅 PB19010450

1.传统机器学习

实验要求：利用三种不同的分类算法，根据鲍鱼的物理测量属性预测鲍鱼的年龄。
数据集说明：

Data Set Characteristics:	Multivariate	**Number of Instances:**	4177
Attribute Characteristics:	Categorical, Integer, Real	**Number of Attributes:**	8
Associated Tasks:	Classification	**Missing Values?**	No

训练集 3554 个，测试集 983 个。

特征说明：

Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
M	0.35	0.265	0.09	0.2255	0.0995	0.0485	0.07	1
F	0.53	0.42	0.135	0.677	0.2565	0.1415	0.21	2

标签说明：
由于该数据的标签数较多(1-27,29),大部分类别的数目极少,在针对每个类别计算 F1-score 时 TP,FP,FN 常为 0,影响计算,故仅保留了该数据集中 label⩽11 的数据.在此基础上,为了使各类数据均衡,将原有的 11 个类按照[1,7],[8,9],[10,11]划分得到了新的三个类。

1.1 线性分类算法

实现思路：
记N为样本数，n为特征数，α为学习率，λ_{L2}为L_2规范系数。(不考虑偏置b)
损失函数：

$$loss_i = (\mathbf{x_i} \cdot \omega - y_i)^2 + \lambda_{L2}\|\omega\|_2^2 \quad \mathbf{x_i} = (x_{i1}, x_{i2}, \dots, x_{in})$$

$$\text{Loss} = \frac{1}{N}\sum_{i}^{N} loss_i$$

梯度：

$$\frac{\partial Loss}{\partial \omega_j} = \frac{2}{N}(XW - Y)\mathbf{x_j} + 2\lambda_{L2}\omega_j \quad \mathbf{x_j} = \left(x_{1j}, x_{2j}, \dots, x_{Nj}\right)^T$$

$$J = \left(\frac{\partial Loss}{\partial \omega_1}, \dots, \frac{\partial Loss}{\partial \omega_n}\right)^T \quad X = \left(x_{ij}\right)_{N\times n}$$

梯度下降：

$$\omega^{k+1} = \omega^k - \alpha J(\omega^k)$$

结果：

$$\hat{Y} = XW^{epochs}$$

核心代码：

```
def fit(self, train_features, train_labels):

    def dj(w_now, x, y):  # 梯度
        Y = np.dot(x, w_now)
        N, num = np.shape(x)
        dJ = np.zeros([num, 1])
        for i in range(num):
            dJ[i, 0] = 2 * np.dot((Y - y).T, x[:, i]) / N + 2 * self.Lambda * w_now[i]
        return dJ

    num = train_features.shape[1]
    w = np.random.random((num, 1))
    # 梯度下降训练
    for i in range(self.epochs):
        gradient = dj(w, train_features, train_labels)
        w = w - self.lr * gradient
    self.w = w
```

测试结果：

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6286876907426246
0.6432432432432432
0.6130434782608696
0.6420118343195267
macro-F1: 0.6327661852745464
micro-F1: 0.6286876907426246
```

准确率约为 62.87%，macro-F1 指标为 0.6328，micro-F1 指标为 0.6287。

注：

对该线性分类器也尝试加入偏置 b 并梯度下降训练，发现对准确率的提升程度非常的小，于是去除了偏置 b。

1.2 朴素贝叶斯分类器

实现思路：

首先由贝叶斯定理：

$$\hat{P}(y_k \mid \mathbf{x}) = \frac{\hat{P}(\mathbf{x} \mid y_k)\hat{P}(y_k)}{\hat{P}(\mathbf{x})}$$

假设数据各特征之间条件独立：

$$\hat{P}(\mathbf{x} \mid y_k) = \prod_{i=1}^{D} \hat{P}(x_i \mid y_k) = \hat{P}(x_1 \mid y_k)\hat{P}(x_2 \mid y_k) \ldots \hat{P}(x_D \mid y_k)$$

若特征取值离散，使用拉普拉斯平滑计算条件概率和先验概率，记D为训练集，N为标签数，D_k为类别为k的数据，D_{k,x_i}为类别为k，第i个特征为 x 的数据，N_i为第i个特征的取值数：

$$\hat{P}(y_k) = \frac{|D_k| + 1}{|D| + N}$$

$$\hat{P}(x_i \mid y_k) = \frac{|D_{k,x_i}| + 1}{|D_k| + N_i}$$

若特征取值连续，可以将连续特征离散化，用区间代替取值，或者使用训练数据估计分布的参数，通常使用高斯分布来表示连续特征的条件概率分布：

$$\hat{P}(x_i \mid y_k) = \frac{1}{\sqrt{2\pi\hat{\sigma}_{k,x_i}^2}} e^{\frac{-\left(x_i - \hat{\mu}_{k,x_i}\right)^2}{2\hat{\sigma}_{k,x_i}^2}}$$

核心代码：

```
def fit(self, traindata, trainlabel, featuretype):
    num = traindata.shape[0]
    N = traindata.shape[1]
    self.Gau = {}
    # 先验概率计算
    label_result = Counter(trainlabel[:, 0])  # 统计
    label_num = len(label_result)
    for i in range(label_num):
        k = list(label_result.keys())[i]
        self.Pc[k] = np.log((label_result[k] + 1) / (num + label_num))
    # 条件概率计算
    for i in range(N):
        # 离散属性
        if featuretype[i] == 0:
            data1 = {}
            for j in range(label_num):
                label = list(label_result.keys())[j]
                data2 = []
                for k in range(num):
                    if trainlabel[:, 0][k] == label:
                        data2.append(traindata[:, i][k])
                data_result = Counter(data2)
                data_num = len(data_result)
```

```
                for k in range(data_num):
                    t = list(data_result.keys())[k]
                    data_result[t] = np.log((data_result[t] + 1) / (label_result[label] + data_num))
                data1[label] = data_result
            self.Pxc[i] = data1
        # 连续属性
        else:
            data1 = {}
            for j in range(label_num):
                label = list(label_result.keys())[j]
                data2 = []
                for k in range(num):
                    if trainlabel[:, 0][k] == label:
                        data2.append(traindata[:, i][k])
                u = np.mean(data2)  # 均值
                v = np.var(data2)  # 方差
                data1[label] = [u, v]
            self.Gau[i] = data1
```

测试结果：

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6185147507629705
0.7104247104247104
0.49484536082474223
0.6657997399219766
macro-F1: 0.623689937057143
micro-F1: 0.6185147507629705
```

准确率约为 61.85%，macro-F1 指标为 0.6237，micro-F1 指标为 0.6185。

注：

处理连续特征时，先将其离散化，将每一个连续值乘以 10 并四舍五入作为一个类，这样做完的预测准确率约为 40%，不太理想。于是重新处理连续特征，使用高斯分布来表示连续特征的条件概率分布，准确率提高至 60%。若要进一步优化，可以在处理连续数据前先做数据分析，查看各特征具体的分布情况，用符合特征分布的分布函数表示条件概率分布。

1.3 SVM 分类器

实现思路：

对于 K 分类(K>2)，我们使用 *one-vs-all* 策略训练，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。

对每个样本点引进一个松弛变量$\xi_i \geq 0$，使得函数间隔加上松弛变量大于等于 1，这样 SVM 的约束条件变为：

$$y_i(\omega \cdot x_i + b) \geq 1 - \xi_i$$

目标函数由原来的$\frac{1}{2}\|\omega\|^2$变为：

$$\frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{N}\xi_i$$

$C > 0$为惩罚参数。该 SVM 求解问题称为软间隔最大化：

$$\min_{\omega,b,\xi_i} \frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{N}\xi_i$$

求解该凸二次规划问题得到分离超平面：

$$\omega^* \cdot x + b^* = 0$$

以及相应的分类决策函数：

$$f(x) = \text{sign}(\omega^* \cdot x + b^*)$$

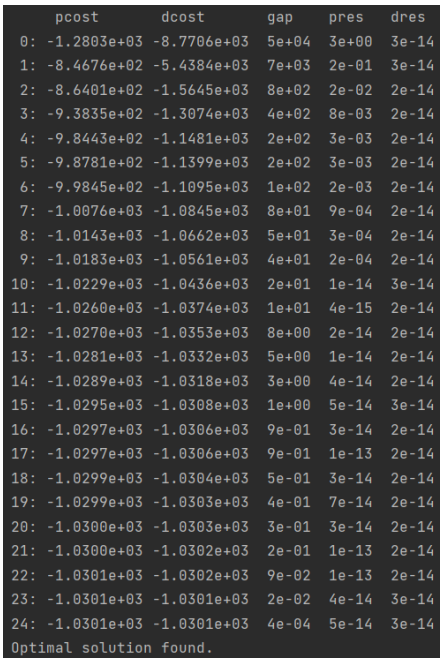
核心代码：

```
def fit(self, train_data, train_label, test_data):
    def KernelMatrix(data, kernel, fun_KERNEL):  # 创建核矩阵
        '''...'''
        return K_Matrix
    def fit_bias(a, vlabel, v, K_Matrix, ind):  # 计算偏置 b
        '''...'''
        return b
    def fit_weight(kernel, data, a, v, vlabel):  # 计算权重 w
        '''...'''
        return w
    def fit_Lagrange(data, vlabel, C, K_Matrix):  # 求 Lagrange 乘子
        '''...'''
        return a
    self.a = None  # 乘子
    self.b = 0  # 偏置
    self.w = []  # 权值
    self.v = []  # 向量
    self.vlabel = []  # 向量标签
    # 训练参数
    _, N = train_data.shape
    K_Matrix = KernelMatrix(train_data, self.kernel, self.KERNEL)
    a = fit_Lagrange(train_data, train_label, self.C, K_Matrix)
    v = a > self.Epsilon
    ind = np.arange(len(a))[v]  # 向量下标
    self.a = a[v]
    self.v = train_data[v]  # 向量
```

```
27.     self.vlabel = train_label[v]
28.     self.b = fit_bias(self.a, self.vlabel, v, K_Matrix, ind)
29.     self.w = fit_weight(self.kernel, N, self.a, self.v, self.vlabel)
30.     # 预测
31.     '''...'''
32.     return y
```

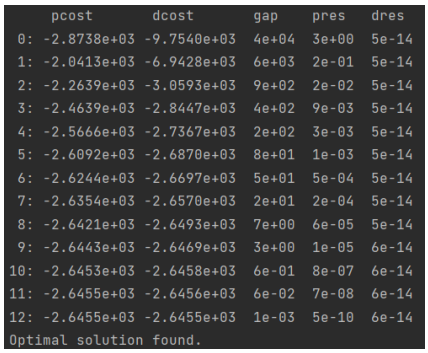
测试结果：

Gauss 核：



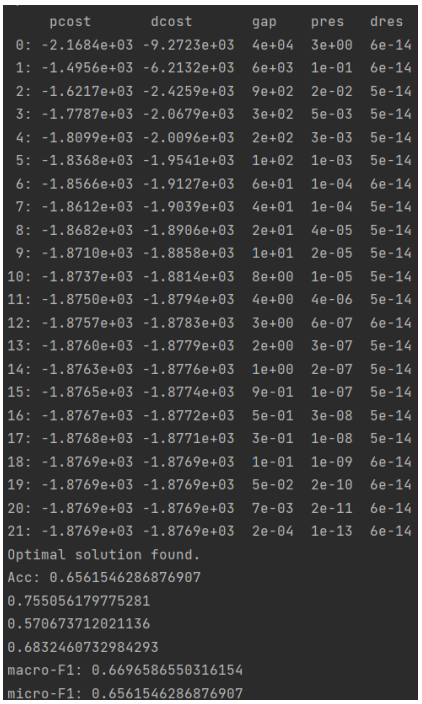
	pcost	dcost	gap	pres	dres
0:	-1.2803e+03	-8.7706e+03	5e+04	3e+00	3e-14
1:	-8.4676e+02	-5.4384e+03	7e+03	2e-01	3e-14
2:	-8.6401e+02	-1.5645e+03	8e+02	2e-02	2e-14
3:	-9.3835e+02	-1.3074e+03	4e+02	8e-03	2e-14
4:	-9.8443e+02	-1.1481e+03	2e+02	3e-03	2e-14
5:	-9.8781e+02	-1.1399e+03	2e+02	3e-03	2e-14
6:	-9.9845e+02	-1.1095e+03	1e+02	2e-03	2e-14
7:	-1.0076e+03	-1.0845e+03	8e+01	9e-04	2e-14
8:	-1.0143e+03	-1.0662e+03	5e+01	3e-04	2e-14
9:	-1.0183e+03	-1.0561e+03	4e+01	2e-04	2e-14
10:	-1.0229e+03	-1.0436e+03	2e+01	1e-14	3e-14
11:	-1.0260e+03	-1.0374e+03	1e+01	4e-15	2e-14
12:	-1.0270e+03	-1.0353e+03	8e+00	2e-14	2e-14
13:	-1.0281e+03	-1.0332e+03	5e+00	1e-14	2e-14
14:	-1.0289e+03	-1.0318e+03	3e+00	4e-14	2e-14
15:	-1.0295e+03	-1.0308e+03	1e+00	5e-14	3e-14
16:	-1.0297e+03	-1.0306e+03	9e-01	3e-14	2e-14
17:	-1.0297e+03	-1.0306e+03	9e-01	1e-13	2e-14
18:	-1.0299e+03	-1.0304e+03	5e-01	3e-14	2e-14
19:	-1.0299e+03	-1.0303e+03	4e-01	7e-14	2e-14
20:	-1.0300e+03	-1.0303e+03	3e-01	3e-14	2e-14
21:	-1.0300e+03	-1.0302e+03	2e-01	1e-13	2e-14
22:	-1.0301e+03	-1.0302e+03	9e-02	1e-13	2e-14
23:	-1.0301e+03	-1.0301e+03	2e-02	4e-14	3e-14
24:	-1.0301e+03	-1.0301e+03	4e-04	5e-14	3e-14

Optimal solution found.



	pcost	dcost	gap	pres	dres
0:	-2.8738e+03	-9.7540e+03	4e+04	3e+00	5e-14
1:	-2.0413e+03	-6.9428e+03	6e+03	2e-01	5e-14
2:	-2.2639e+03	-3.0593e+03	9e+02	2e-02	5e-14
3:	-2.4639e+03	-2.8447e+03	4e+02	9e-03	5e-14
4:	-2.5666e+03	-2.7367e+03	2e+02	3e-03	5e-14
5:	-2.6092e+03	-2.6870e+03	8e+01	1e-03	5e-14
6:	-2.6244e+03	-2.6697e+03	5e+01	5e-04	5e-14
7:	-2.6354e+03	-2.6570e+03	2e+01	2e-04	5e-14
8:	-2.6421e+03	-2.6493e+03	7e+00	6e-05	5e-14
9:	-2.6443e+03	-2.6469e+03	3e+00	1e-05	6e-14
10:	-2.6453e+03	-2.6458e+03	6e-01	8e-07	6e-14
11:	-2.6455e+03	-2.6456e+03	6e-02	7e-08	6e-14
12:	-2.6455e+03	-2.6455e+03	1e-03	5e-10	6e-14

Optimal solution found.

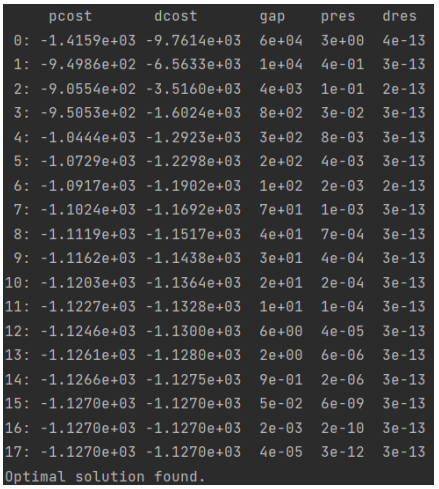


	pcost	dcost	gap	pres	dres
0:	-2.1684e+03	-9.2723e+03	4e+04	3e+00	6e-14
1:	-1.4956e+03	-6.2132e+03	6e+03	1e-01	6e-14
2:	-1.6217e+03	-2.4259e+03	9e+02	2e-02	5e-14
3:	-1.7787e+03	-2.0679e+03	3e+02	5e-03	5e-14
4:	-1.8099e+03	-2.0096e+03	2e+02	3e-03	5e-14
5:	-1.8368e+03	-1.9541e+03	1e+02	1e-03	5e-14
6:	-1.8566e+03	-1.9127e+03	6e+01	1e-04	6e-14
7:	-1.8612e+03	-1.9039e+03	4e+01	1e-04	5e-14
8:	-1.8682e+03	-1.8906e+03	2e+01	4e-05	5e-14
9:	-1.8710e+03	-1.8858e+03	1e+01	2e-05	5e-14
10:	-1.8737e+03	-1.8814e+03	8e+00	1e-05	5e-14
11:	-1.8750e+03	-1.8794e+03	4e+00	4e-06	5e-14
12:	-1.8757e+03	-1.8783e+03	3e+00	6e-07	6e-14
13:	-1.8760e+03	-1.8779e+03	2e+00	3e-07	5e-14
14:	-1.8763e+03	-1.8776e+03	1e+00	2e-07	5e-14
15:	-1.8765e+03	-1.8774e+03	9e-01	1e-07	5e-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	6e-14
19:	-1.8769e+03	-1.8769e+03	5e-02	2e-10	6e-14
20:	-1.8769e+03	-1.8769e+03	7e-03	2e-11	6e-14
21:	-1.8769e+03	-1.8769e+03	2e-04	1e-13	6e-14

Optimal solution found.
Acc: 0.6561546286876907
0.755056179775281
0.570673712021136
0.6832460732984293
macro-F1: 0.6696586550316154
micro-F1: 0.6561546286876907

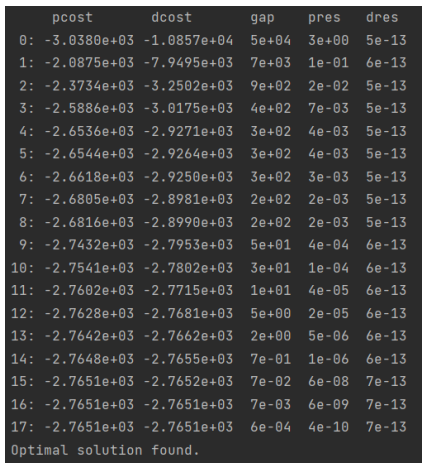
准确率约为 65.62%，macro-F1 指标为 0.6697，micro-F1 指标为 0.6562。

Linear 核：



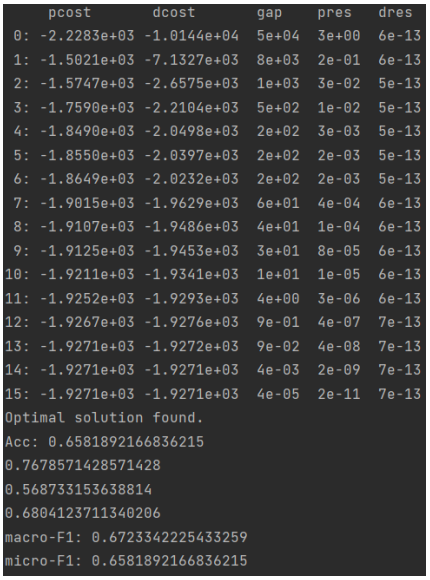
	pcost	dcost	gap	pres	dres
0:	-1.4159e+03	-9.7614e+03	6e+04	3e+00	4e-13
1:	-9.4986e+02	-6.5633e+03	1e+04	4e-01	3e-13
2:	-9.0554e+02	-3.5160e+03	4e+03	1e-01	2e-13
3:	-9.5053e+02	-1.6024e+03	8e+02	3e-02	3e-13
4:	-1.0444e+03	-1.2923e+03	3e+02	8e-03	3e-13
5:	-1.0729e+03	-1.2298e+03	2e+02	4e-03	3e-13
6:	-1.0917e+03	-1.1902e+03	1e+02	2e-03	2e-13
7:	-1.1024e+03	-1.1692e+03	7e+01	1e-03	3e-13
8:	-1.1119e+03	-1.1517e+03	4e+01	7e-04	3e-13
9:	-1.1162e+03	-1.1438e+03	3e+01	4e-04	3e-13
10:	-1.1203e+03	-1.1364e+03	2e+01	2e-04	3e-13
11:	-1.1227e+03	-1.1328e+03	1e+01	1e-04	3e-13
12:	-1.1246e+03	-1.1300e+03	6e+00	4e-05	3e-13
13:	-1.1261e+03	-1.1280e+03	2e+00	6e-06	3e-13
14:	-1.1266e+03	-1.1275e+03	9e-01	2e-06	3e-13
15:	-1.1270e+03	-1.1270e+03	5e-02	6e-09	3e-13
16:	-1.1270e+03	-1.1270e+03	2e-03	2e-10	3e-13
17:	-1.1270e+03	-1.1270e+03	4e-05	3e-12	3e-13

Optimal solution found.



	pcost	dcost	gap	pres	dres
0:	-3.0380e+03	-1.0857e+04	5e+04	3e+00	5e-13
1:	-2.0875e+03	-7.9495e+03	7e+03	1e-01	6e-13
2:	-2.3734e+03	-3.2502e+03	9e+02	2e-02	5e-13
3:	-2.5886e+03	-3.0175e+03	4e+02	7e-03	5e-13
4:	-2.6536e+03	-2.9271e+03	3e+02	4e-03	5e-13
5:	-2.6544e+03	-2.9264e+03	3e+02	4e-03	5e-13
6:	-2.6618e+03	-2.9250e+03	3e+02	3e-03	5e-13
7:	-2.6805e+03	-2.8981e+03	2e+02	2e-03	5e-13
8:	-2.6816e+03	-2.8990e+03	2e+02	2e-03	5e-13
9:	-2.7432e+03	-2.7953e+03	5e+01	4e-04	6e-13
10:	-2.7541e+03	-2.7802e+03	3e+01	1e-04	6e-13
11:	-2.7602e+03	-2.7715e+03	1e+01	4e-05	6e-13
12:	-2.7628e+03	-2.7681e+03	5e+00	2e-05	6e-13
13:	-2.7642e+03	-2.7662e+03	2e+00	5e-06	6e-13
14:	-2.7648e+03	-2.7655e+03	7e-01	1e-06	6e-13
15:	-2.7651e+03	-2.7652e+03	7e-02	6e-08	7e-13
16:	-2.7651e+03	-2.7651e+03	7e-03	6e-09	7e-13
17:	-2.7651e+03	-2.7651e+03	6e-04	4e-10	7e-13

Optimal solution found.



	pcost	dcost	gap	pres	dres
0:	-2.2283e+03	-1.0144e+04	5e+04	3e+00	6e-13
1:	-1.5021e+03	-7.1327e+03	8e+03	2e-01	6e-13
2:	-1.5747e+03	-2.6575e+03	1e+03	3e-02	5e-13
3:	-1.7590e+03	-2.2104e+03	5e+02	1e-02	5e-13
4:	-1.8490e+03	-2.0498e+03	2e+02	3e-03	5e-13
5:	-1.8550e+03	-2.0397e+03	2e+02	2e-03	5e-13
6:	-1.8649e+03	-2.0232e+03	2e+02	2e-03	5e-13
7:	-1.9015e+03	-1.9629e+03	6e+01	4e-04	6e-13
8:	-1.9107e+03	-1.9486e+03	4e+01	1e-04	6e-13
9:	-1.9125e+03	-1.9453e+03	3e+01	8e-05	6e-13
10:	-1.9211e+03	-1.9341e+03	1e+01	1e-05	6e-13
11:	-1.9252e+03	-1.9293e+03	4e+00	3e-06	6e-13
12:	-1.9267e+03	-1.9276e+03	9e-01	4e-07	7e-13
13:	-1.9271e+03	-1.9272e+03	9e-02	4e-08	7e-13
14:	-1.9271e+03	-1.9271e+03	4e-03	2e-09	7e-13
15:	-1.9271e+03	-1.9271e+03	4e-05	2e-11	7e-13

Optimal solution found.
Acc: 0.6581892166836215
0.7678571428571428
0.568733153638814
0.6804123711340206
macro-F1: 0.6723342225433259
micro-F1: 0.6581892166836215

准确率约为 65.82%，macro-F1 指标为 0.6723，micro-F1 指标为 0.6582。

Poly 核：

	pcost	dcost	gap	pres	dres
0:	-1.3453e+03	-1.0113e+04	6e+04	3e+00	3e-12
1:	-9.1009e+02	-6.9922e+03	1e+04	5e-01	2e-12
2:	-8.3485e+02	-3.6676e+03	4e+03	2e-01	2e-12
3:	-8.3788e+02	-2.2054e+03	2e+03	6e-02	2e-12
4:	-9.0835e+02	-1.3485e+03	5e+02	2e-02	2e-12
5:	-9.4480e+02	-1.2156e+03	3e+02	8e-03	2e-12
6:	-9.6805e+02	-1.1397e+03	2e+02	4e-03	2e-12
7:	-9.8196e+02	-1.0995e+03	1e+02	2e-03	2e-12
8:	-9.9427e+02	-1.0662e+03	8e+01	1e-03	2e-12
9:	-1.0006e+03	-1.0516e+03	5e+01	7e-04	2e-12
10:	-1.0058e+03	-1.0395e+03	3e+01	4e-04	2e-12
11:	-1.0091e+03	-1.0330e+03	2e+01	2e-04	2e-12
12:	-1.0130e+03	-1.0253e+03	1e+01	8e-05	2e-12
13:	-1.0145e+03	-1.0230e+03	9e+00	5e-05	2e-12
14:	-1.0161e+03	-1.0203e+03	4e+00	1e-05	2e-12
15:	-1.0170e+03	-1.0190e+03	2e+00	2e-06	2e-12
16:	-1.0177e+03	-1.0183e+03	6e-01	3e-07	2e-12
17:	-1.0179e+03	-1.0180e+03	8e-02	5e-08	2e-12
18:	-1.0180e+03	-1.0180e+03	9e-03	4e-09	2e-12
19:	-1.0180e+03	-1.0180e+03	1e-04	5e-11	2e-12

Optimal solution found.

	pcost	dcost	gap	pres	dres
0:	-2.9682e+03	-1.0603e+04	5e+04	3e+00	7e-12
1:	-2.1011e+03	-8.1536e+03	1e+04	4e-01	4e-12
2:	-2.1563e+03	-3.6288e+03	2e+03	3e-02	3e-12
3:	-2.5255e+03	-2.8971e+03	4e+02	6e-03	4e-12
4:	-2.5807e+03	-2.8329e+03	3e+02	3e-03	4e-12
5:	-2.6559e+03	-2.7388e+03	9e+01	1e-03	4e-12
6:	-2.6757e+03	-2.7154e+03	4e+01	4e-04	4e-12
7:	-2.6882e+03	-2.7008e+03	1e+01	1e-04	4e-12
8:	-2.6921e+03	-2.6963e+03	4e+00	3e-05	4e-12
9:	-2.6934e+03	-2.6948e+03	1e+00	9e-06	4e-12
10:	-2.6940e+03	-2.6941e+03	1e-01	1e-07	5e-12
11:	-2.6941e+03	-2.6941e+03	1e-02	1e-08	5e-12
12:	-2.6941e+03	-2.6941e+03	1e-03	1e-09	4e-12

Optimal solution found.

	pcost	dcost	gap	pres	dres
0:	-2.2173e+03	-1.1255e+04	6e+04	4e+00	4e-12
1:	-1.4797e+03	-8.5162e+03	1e+04	4e-01	4e-12
2:	-1.4684e+03	-3.1823e+03	2e+03	3e-02	3e-12
3:	-1.6725e+03	-2.2831e+03	6e+02	1e-02	3e-12
4:	-1.7472e+03	-2.1379e+03	4e+02	4e-03	3e-12
5:	-1.7940e+03	-2.0200e+03	2e+02	2e-03	3e-12
6:	-1.8084e+03	-1.9878e+03	2e+02	1e-03	3e-12
7:	-1.8282e+03	-1.9451e+03	1e+02	7e-04	3e-12
8:	-1.8358e+03	-1.9288e+03	9e+01	5e-04	3e-12
9:	-1.8393e+03	-1.9214e+03	8e+01	4e-04	3e-12
10:	-1.8486e+03	-1.8987e+03	5e+01	1e-04	4e-12
11:	-1.8545e+03	-1.8887e+03	3e+01	5e-05	4e-12
12:	-1.8580e+03	-1.8831e+03	3e+01	3e-05	4e-12
13:	-1.8601e+03	-1.8795e+03	2e+01	2e-05	4e-12
14:	-1.8629e+03	-1.8754e+03	1e+01	9e-06	4e-12
15:	-1.8642e+03	-1.8736e+03	9e+00	6e-06	4e-12
16:	-1.8667e+03	-1.8702e+03	4e+00	1e-06	4e-12
17:	-1.8679e+03	-1.8689e+03	1e+00	1e-07	4e-12
18:	-1.8683e+03	-1.8684e+03	8e-02	7e-09	4e-12
19:	-1.8683e+03	-1.8683e+03	1e-02	9e-10	4e-12
20:	-1.8683e+03	-1.8683e+03	4e-04	2e-11	4e-12

Optimal solution found.
Acc: 0.6449643947100712
0.750551876379691
0.5717948717948718
0.6575716234652115
macro-F1: 0.6599727905465914
micro-F1: 0.6449643947100712

准确率约为 64.50%，macro-F1 指标为 0.6600，micro-F1 指标为 0.6450。

注：

对比下来，Linear 核的效果要略好。而时间上，Gauss 核用时 323.78s，Linear 核用时 111.34s，Ploy 核用时 135.72 s。Linear 核最快，而 Gauss 核用时过长。

实际应用最广的是 Gauss 核，主要用于线性不可分的情形，无论是小样本还是大样本，高维还是低维等情况，Gauss 核均适用，它可以将一个样本映射到一个更高维的空间，而 Linear 核是 Gauss 核的一个特例。与 Ploy 核相比，Gauss 核需要确定的参数要少，核函数参数的多少直接影响函数的复杂程度。另外，当多项式的阶数比较高时，核矩阵的元素值将趋于无穷大或无穷小。

Linear 核主要用于线性可分的情形，参数少，速度快，对于一般数据，分类效果已经很理想了。并且如果特征维数很高，往往线性可分（SVM 解决非线性分类问题的思路就是将样本映射到更高维的特征空间中）。如果样本数量很多，由于求解最优化问题的时候，目标函数涉及两两样本计算内积，使用 Gauss 核计算量明显大于 Linear 核，所以手动添加一些特征，使得线性可分，然后可以用 Linear 核的 SVM

至于到底该采用哪种核，要根据具体问题，有的数据是线性可分的，有的不可分，需要多尝试不同核不同参数。如果特征的提取的好，包含的信息量足够大，很多问题都是线性可分的。当然，如果有足够的时间去寻找 Gauss 核参数，或许能达到更好的效果。

2.深度学习

2.1　手写感知机模型并进行反向传播

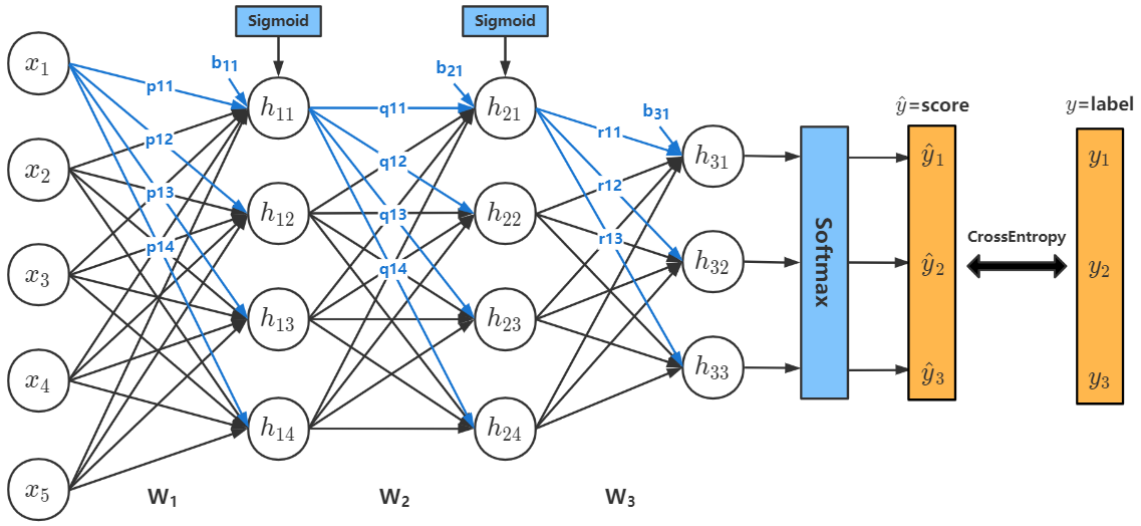
实验内容：

- 实现一个 4 层的感知机模型，神经元设置为 5-4-4-3，即输入的特征为 5，输出的类别个数的 3，激活函数设置为 sigmoid；
- 实现 BP 算法；
- 实现梯度下降算法。

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。

实现思路：



激活函数为 Sigmoid，输出用 Softmax 处理与转换为 One-hot 的 label 做交叉熵。

记σ为 Sigmoid，s为 Softmax，L为交叉熵，有

$$\sigma' = \sigma(1-\sigma) \quad s' = s(1-s)$$

逆向传播计算梯度：

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial h_3}\frac{\partial h_3}{\partial W_3} = -\frac{1}{N\hat{y}_k}(s(1-s))h_2^T = \frac{1}{N}(\text{score}-\text{label})h_2^T$$

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial h_3}\frac{\partial h_3}{\partial b_3} = -\frac{1}{N\hat{y}_k}(s(1-s))h_2^T = \frac{1}{N}(\text{score}-\text{label})\mathbf{1}^T$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial h_3}\frac{\partial h_3}{\partial W_2} = \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W_2}$$
$$= (W_3^T \frac{1}{N}(\text{score} - \text{label}) * (\sigma(1-\sigma)))h_1^T$$
$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial h_3}\frac{\partial h_3}{\partial b_2} = \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial b_2}$$
$$= (W_3^T \frac{1}{N}(\text{score} - \text{label}) * (\sigma(1-\sigma)))\mathbf{1^T}$$

同理，求出W_1和b_1的梯度：

$$\frac{\partial L}{\partial W_1} = (W_2^T(W_3^T \frac{1}{N}(\text{score} - \text{label}) * \sigma^{'}) * \sigma^{'})X^T$$

$$\frac{\partial L}{\partial b_1} = (W_2^T(W_3^T \frac{1}{N}(\text{score} - \text{label}) * \sigma^{'}) * \sigma^{'})\mathbf{1^T}$$

接下来使用梯度下降优化：

$$W_1 = W_1 - \alpha\frac{\partial L}{\partial W_1}$$
$$W_2 = W_2 - \alpha\frac{\partial L}{\partial W_2}$$
$$W_3 = W_3 - \alpha\frac{\partial L}{\partial W_3}$$
$$b_1 = b_1 - \alpha\frac{\partial L}{\partial b_1}$$
$$b_2 = b_2 - \alpha\frac{\partial L}{\partial b_2}$$
$$b_3 = b_3 - \alpha\frac{\partial L}{\partial b_3}$$

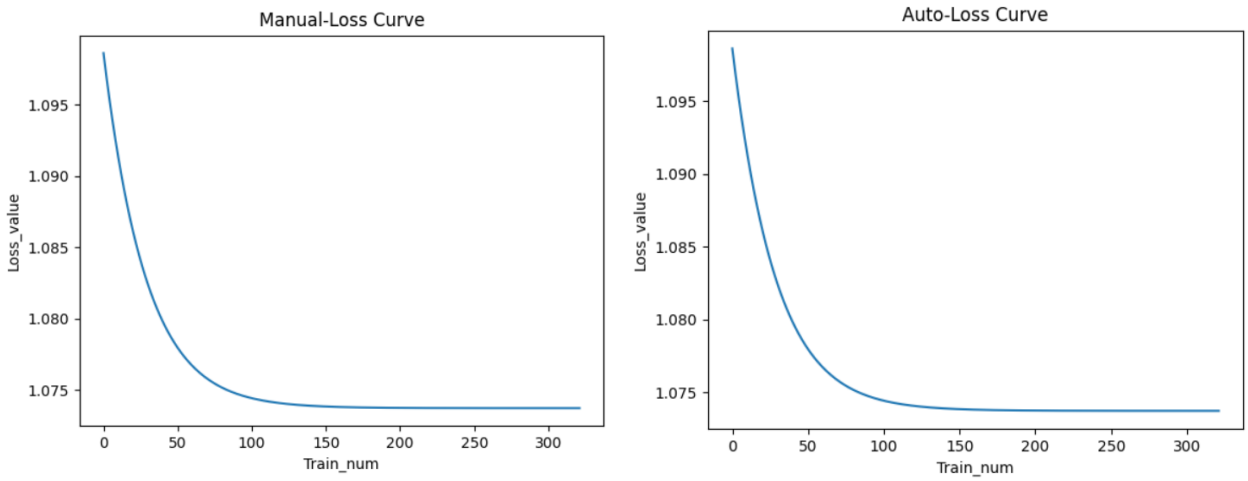
核心代码：

```
class MultiLayerPerceptron:
    def __init__(self):
        self.w1 = np.ones([4, 5])
        self.w2 = np.ones([4, 4])
        self.w3 = np.ones([3, 4])   # 权值 w
        self.b1 = np.ones([4, 1])
        self.b2 = np.ones([4, 1])
        self.b3 = np.ones([3, 1])   # 偏置 b
        self.h1 = None  # h1
        self.h2 = None  # h2
        self.h3 = None  # h3
        self.L = []  # 损失函数表
        self.alpha = 0.01  # 学习率
        self.epochs = 1000  # 最大训练次数
```

```
15.         self.epsilon = 1e-8  # 结束条件
16.
17.     def Sigmoid(self, data):  # 激活函数 Sigmoid
18.         '''...'''
19.     def Softmax(self, data):  # 激活函数 Softmax
20.         '''...'''
21.     def DSigmoid(self, data):   # Sigmoid 导数
22.         '''...'''
23.     def CrossEntropy(self, y, label):  # 交叉熵
24.         '''...'''
25.     def forword(self, data, w1, w2, w3):    # 前向传播
26.         self.h1 = self.Sigmoid(np.dot(w1, data.T) + self.b1)
27.         self.h2 = self.Sigmoid(np.dot(w2, self.h1) + self.b2)
28.         self.h3 = self.Softmax((np.dot(w3, self.h2) + self.b3).T)
29.         return self.h3
30.
31.     def BackPropagation(self, data, score, label):  # 逆向传播求梯度
32.         datanum, _ = np.shape(label)
33.         # w3 和 b3 的梯度
34.         temp0 = (score - label) / datanum
35.         j_b3 = np.dot(temp0.T, np.ones([datanum, 1]))
36.         j_w3 = np.dot(temp0.T, self.h2.T)
37.         # w2 和 b2 的梯度
38.         temp1 = np.dot(self.w3.T, temp0.T)
39.         temp2 = self.DSigmoid(np.dot(self.w2, self.h1) + self.b2)
40.         j_b2 = np.dot(temp1 * temp2, np.ones([datanum, 1]))
41.         j_w2 = np.dot(temp1 * temp2, self.h1.T)
42.         # w1 和 b1 的梯度
43.         temp3 = temp1 * temp2
44.         temp4 = self.DSigmoid(np.dot(self.w1, data.T) + self.b1)
45.         j_b1 = np.dot(temp3 * temp4, np.ones([datanum, 1]))
46.         j_w1 = np.dot(temp3 * temp4, data)
47.         return j_w1, j_w2, j_w3, j_b1, j_b2, j_b3
48.
49.     def GradientDescent(self, j_w1, j_w2, j_w3, j_b1, j_b2, j_b3):  # 梯度下降
50.         '''...'''
```

测试结果：

Loss 训练曲线：




```
手动
训练次数 100 损失函数 1.0744571631020223
训练次数 200 损失函数 1.0737548590139492
训练次数 300 损失函数 1.0737356070956614
训练次数 322 损失函数 1.0737352890613803
自动
训练次数 100 损失函数 1.0744571631020234
训练次数 200 损失函数 1.0737548590139498
训练次数 300 损失函数 1.0737356070956614
训练次数 322 损失函数 1.0737352890613794
```

手动实现与 pytorch 实现结果基本一致。

梯度对比：

```
手动
j_w1: [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
j_w2: [[-5.76626999e-19 -5.76626999e-19 -5.76626999e-19 -5.76626999e-19]
 [-5.76626999e-19 -5.76626999e-19 -5.76626999e-19 -5.76626999e-19]
 [-5.76626999e-19 -5.76626999e-19 -5.76626999e-19 -5.76626999e-19]
 [-5.76626999e-19 -5.76626999e-19 -5.76626999e-19 -5.76626999e-19]]
j_w3: [[ 0.05297638  0.05297638  0.05297638  0.05297638]
 [-0.01655512 -0.01655512 -0.01655512 -0.01655512]
 [-0.03642126 -0.03642126 -0.03642126 -0.03642126]]
j_b1: [[0.]
 [0.]
 [0.]
 [0.]]
j_b2: [[-5.76626999e-19]
 [-5.76626999e-19]
 [-5.76626999e-19]
 [-5.76626999e-19]]
j_b3: [[ 0.05333333]
 [-0.01666667]
 [-0.03666667]]
```

```
自动
j_w1: tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]], dtype=torch.float64)
j_w2: tensor([[-5.7663e-19, -5.7663e-19, -5.7663e-19, -5.7663e-19],
        [-5.7663e-19, -5.7663e-19, -5.7663e-19, -5.7663e-19],
        [-5.7663e-19, -5.7663e-19, -5.7663e-19, -5.7663e-19],
        [-5.7663e-19, -5.7663e-19, -5.7663e-19, -5.7663e-19]],
       dtype=torch.float64)
j_w3: tensor([[ 0.0530,  0.0530,  0.0530,  0.0530],
        [-0.0166, -0.0166, -0.0166, -0.0166],
        [-0.0364, -0.0364, -0.0364, -0.0364]], dtype=torch.float64)
j_b1: tensor([[0.],
        [0.],
        [0.],
        [0.]], dtype=torch.float64)
j_b2: tensor([[-5.7663e-19],
        [-5.7663e-19],
        [-5.7663e-19],
        [-5.7663e-19]], dtype=torch.float64)
j_b3: tensor([[ 0.0533],
        [-0.0167],
        [-0.0367]], dtype=torch.float64)
```

手动求导与 pytorch 求导结果基本一致。

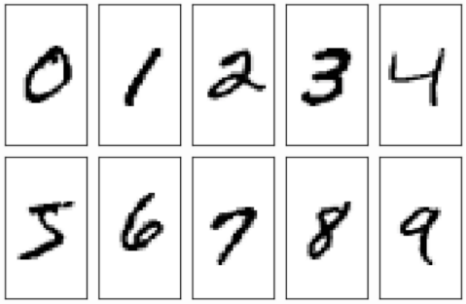
2.2 复现 MLP-Mixer

实验内容：

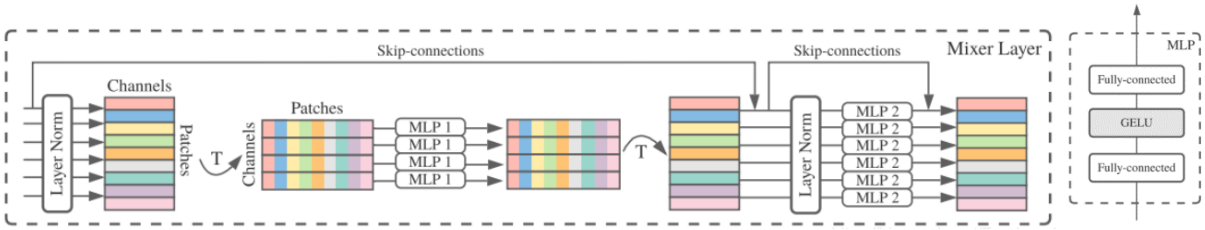
复现 MLP-Mixer 模型，并在 MNIST 数据集上进行测试。

数据集说明：

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)。



实现思路：



Mixer_Layer 部分负责一个 Mixer Layer 模块，包括 layer Norm、MLP1、Skip-connections、MLP2.

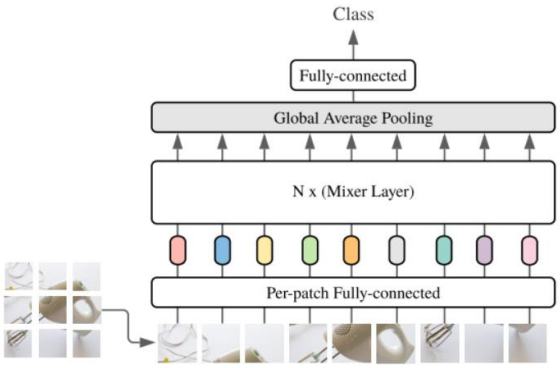
主要过程为：将通过 Mixer Layer 模块的输入矩阵，使用 MLP 先后对列、行进行映射，实现空间域和通道域的信息融合。与传统卷积不同的是，Mixer Layer 将空间域和通道域分开操作。

- 首先将输入归一化，嵌入到一系列通道中，构成一个矩阵 X。
- 将该矩阵转置通过 MLP1（Token-mixing MLPs）后再转置回原维度得到 X1，MLP1 作用在原矩阵 X 的列上。
- 接着将 X 与 X1 进行 Skip-connection 后得到 X3。将 X3 归一化，通过 MLP2（Channel-mixing MLPs）得到 X4，MLP2 作用在原矩阵 X 的行上。
- 最后将 X3 与 X4 进行 Skip-connection 后输出。

其中两个 MLP 模块采用了相同的结构，包含 2 个全连接层和一个 GELU 激活函数层 ，给模型融入非线性成分。

MLPMixer 部分负责 MLP 的整体结构，包括 Per-patch Fully-connected、N 个 Mixer Layer 模块、Global Average Pooling、Fully-connected。

主要过程为：先将输入图片拆分成多个 patches，然后用一个全连接网络对所有 patch 进行处理，提取出 tokens。将其送入 N 个 Mixer Layer 模块，输出后进行全局池化。最后通过 Fully-connected 进行分类。



核心代码：

Mixer_Layer 部分：

```
1.  class Mixer_Layer(nn.Module):
2.      def __init__(self, patch_size, hidden_dim):
3.          super(Mixer_Layer, self).__init__()
4.          self.token_dim = (28 // patch_size) ** 2  # 序列数（patch 个数）
5.          self.channel_dim = 14  # 通道数（嵌入维度）
6.          self.mlp1 = nn.Sequential(  # Token_mixing
7.              nn.Linear(self.token_dim, hidden_dim),  # Fully-connected
8.              nn.GELU(),  # 激活函数 GELU
9.              nn.Dropout(0),  # 防止或减轻过拟合
10.             nn.Linear(hidden_dim, self.token_dim),  # Fully-connected
11.             nn.Dropout(0)
12.         )
13.         self.mlp2 = nn.Sequential(  # Channel_mixing
14.             nn.Linear(self.channel_dim, hidden_dim),  # Fully-connected
15.             nn.GELU(),  # 激活函数 GELU
16.             nn.Dropout(0),
17.             nn.Linear(hidden_dim, self.channel_dim),  # Fully-connected
18.             nn.Dropout(0)
19.         )
20.         self.norm = nn.LayerNorm(self.channel_dim)
21.
22.     def forward(self, x):
23.         x = self.norm(x).transpose(1, 2)  # Layer Norm
24.         x_token = self.mlp1(x)  # MLP1-Token_mixing
25.         x_skip = self.norm((x + x_token).transpose(1, 2))  # Skip-connections&Layer Norm
26.         x_channel = self.mlp2(x_skip)  # MLP2-Channel_mixing
27.         x_mixer = x_skip + x_channel  # Skip-connections
28.         return x_mixer
```

MLPMixer 部分

```
1.  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'
        self.image_channel = 1  # MNIST 数据集图片维度
        self.channel_dim = 14  # 通道数（嵌入维度）
        self.class_num = 10  # 分类标签数（0~9）
        # Per-patch Fully-connected
        self.Pre_patch = nn.Conv2d(in_channels=self.image_channel, out_channels=self.channel_dim,  kernel_size=patch_size, stride=patch_size)
        # N * Mixer Layer
        self.Nmixer_layer = nn.Sequential(*[Mixer_Layer(patch_size, hidden_dim) for _ in range(depth)])
        self.norm = nn.LayerNorm(self.channel_dim)
        self.fully_connected = nn.Linear(self.channel_dim, self.class_num)
        self.global_pool = torch.mean

    def forward(self, data):
        x = self.Pre_patch(data)    # Per-patch Fully-connected
        x = x.flatten(2).transpose(1, 2)    # 展平
        x = self.Nmixer_layer(x)    # N * Mixer Layer
        x = self.norm(x)    # Pre_head Layer_norm
        x = self.global_pool(x, dim=1)  # Global Average Pooling
        x = self.fully_connected(x)     # Fully_connected
        return x
```

优化器和参数：

```
model = MLPMixer(patch_size=4, hidden_dim=128, depth=6).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adamax(model.parameters(), lr=10*learning_rate, betas=(0.5, 0.999))
```

测试结果：

准确率约 98%

```
Train Epoch: 0/5 [0/60000]  Loss: 2.311446
Train Epoch: 0/5 [12800/60000]  Loss: 0.456622
Train Epoch: 0/5 [25600/60000]  Loss: 0.325239
Train Epoch: 0/5 [38400/60000]  Loss: 0.333890
Train Epoch: 0/5 [51200/60000]  Loss: 0.205388
Train Epoch: 1/5 [0/60000]  Loss: 0.070794
Train Epoch: 1/5 [12800/60000]  Loss: 0.149972
Train Epoch: 1/5 [25600/60000]  Loss: 0.271728
Train Epoch: 1/5 [38400/60000]  Loss: 0.061849
Train Epoch: 1/5 [51200/60000]  Loss: 0.086888
Train Epoch: 2/5 [0/60000]  Loss: 0.122365
Train Epoch: 2/5 [12800/60000]  Loss: 0.073975
Train Epoch: 2/5 [25600/60000]  Loss: 0.175230
Train Epoch: 2/5 [38400/60000]  Loss: 0.217961
Train Epoch: 2/5 [51200/60000]  Loss: 0.122104
Train Epoch: 3/5 [0/60000]  Loss: 0.141113
Train Epoch: 3/5 [12800/60000]  Loss: 0.061265
Train Epoch: 3/5 [25600/60000]  Loss: 0.023571
Train Epoch: 3/5 [38400/60000]  Loss: 0.091374
Train Epoch: 3/5 [51200/60000]  Loss: 0.079833
Train Epoch: 4/5 [0/60000]  Loss: 0.084125
Train Epoch: 4/5 [12800/60000]  Loss: 0.025314
Train Epoch: 4/5 [25600/60000]  Loss: 0.093155
Train Epoch: 4/5 [38400/60000]  Loss: 0.020984
Train Epoch: 4/5 [51200/60000]  Loss: 0.252636
Test set: Average loss: 0.0711   Acc 0.98
```

注：

- 优化器尝试使用了 Adam、Adamax、SGD、ASGD、Adadelta。其中 ASGD 也称为 SAG，是用空间换时间的一种 SGD。Adadelta 是一种自适应优化方法，是自适应的为各个参数分配不同的学习率。这个学习率的变化，会受到梯度的大小和迭代次数的影响。梯度越大，学习率越小；梯度越小，学习率越大。其分母中采用距离当前时间点比较近的累计项，这可以避免在训练后期，学习率过小。Adam 是一种自适应学习率的优化方法，利用梯度的一阶矩估计和二阶矩估计动态的调整学习率。Adamax 是对 Adam 增加了一个学习率上限的概念。通过调参，使用这些优化器的准确率都可以达到 90%以上，而 Adamax 的效果最佳。
- 尝试调整各个参数，发现 patchsize 和 depth 对结果的影响较大，hidden_dim 对结果的影响较小，并且对于不同优化器，对学习率的需要也不一样。

思考：

- CV 任务主流网络结构完成了 MLP->CNN->Transformer->MLP 这样一次轮回，并不是说明这是一个闭环，早期的 MLP 并不完全等同于现在的 MLP，区别主要体现在算力和数据量上。早期人们放弃 MLP 而使用 CNN 是因为算力不足，CNN 更节省算力，训练模型更容易。现在算力提升了，数据完善了，就有了重新回到 MLP 的可能。并且 MLP-Mixer 的成功也基于使用了多种现代网络结构技巧，并不是只是用了 MLP，也使用了现在 SOTA 模型里的一些结构，比如 Layer Norm、GELU 层、ResNet 里的 Skip connection。MLP-Mixer 说明在分类这种简单的任务上是可以通过算力的堆砌来训练出比 CNN 更广义的 MLP 模型。所以我认为 MLP->CNN->Transformer->MLP 的技术是逐渐上升的。
- MLP-mixer 论文中声称没有使用卷积操作：

  We propose the *MLP-Mixer* architecture (or "Mixer" for short), a competitive but conceptually and technically simple alternative, that does not use convolutions or self-attention. Instead, Mixer's architecture is based entirely on multi-layer perceptrons (MLPs) that are repeatedly applied across either spatial locations or feature channels. Mixer relies only on basic matrix multiplication routines, changes to data layout (reshapes and transpositions), and scalar nonlinearities.

  但在模型讲述上又多次提到了“卷积”，例如此处提到，在极端情况下，MLP 架构可以看作是一个非常特殊的 CNN，它使用 1×1 卷积进行 channel-mixing，以及用于 toking-mixing 的完整接收域的单通道 depth-wise 卷积和参数共享：

  In the extreme case, our architecture can be seen as a very special CNN, which uses 1×1 convolutions for *channel mixing*, and single-channel depth-wise convolutions of a full receptive field and parameter sharing for *token mixing*. However, the converse is not true as typical CNNs are not special cases of

  并且在论文最后的官方实现里也使用了卷积函数：

  ```
  x = nn.Conv(self.hidden_dim, (s,s), strides=(s,s), name='stem')(x)
  ```

  原则上来说，卷积和全连接层可以按照如下的方式互相转化：如果卷积核的尺寸大到包含

了所有输入，以至于无法在输入上滑动，那么卷积就变成了全连接层。反之，如果全连接层足够稀疏，后一层的每个神经元只跟前一层对应位置附近的少数几个神经元连接，并且这些连接的权重在不同的空间位置都相同，那么全连接层也就变成了卷积层。

在 MLP-Mixer 的 per_patch 层，本实验选取 patchsize=4，所以第一步是把输入切分成若干 4x4 的 patch，然后对每个 patch 使用相同的投影。最简单的实现就是采用 4x4 的卷积核，stride 取 4x4，计算二维卷积。当然，这一步也可以按照全连接层来实现：首先把每个 4x4 的 patch 中的像素通过 reshape 等操作放在最后一维，然后再做一层线性变换。

在 Mixer-Layer 的 token-mixing 模块如果用 MLP 来实现，就是把同一个通道的像素值都放到最后一维，然后接一个线性变换即可。如果用卷积来实现，实质上是一个 depth-wise 卷积，并且各个通道要共享参数。

在 Mixer-Layer 的 channel-mixing 模块是对同一位置的不同通道进行融合。可以认为是一个逐点卷积。当然，也可以利用 permute 把相同位置不同通道的元素丢到最后一维去，然后统一做一个线性变换。

当 MLP-Mixer 对每个 patch 做相同的线性变换的时候，其实就已经在用卷积了。因为卷积的本质是局部连接+参数共享，而划分 patch=局部连接，对各个 patch 应用相同的线性变换=参数共享。只不过，它用的卷积核有一个 patch 那么大。而进行 token-mixing 和 channel-mixing 时，实际就是把普通的卷积拆成了 depth-wise 卷积和逐点卷积。

3.实验总结

本实验分为传统机器学习和深度学习两个部分。

在第一部分，手写了线性分类算法、朴素贝叶斯分类器、SVM 分类器，通过手写各分类器的训练部分，深刻理解了模型的理论细节和详细步骤。在线性分类算法里练习了梯度下降优化；在朴素贝叶斯分类器中学会了处理连续特征的不同方法，并熟悉了利用贝叶斯定理计算概率的步骤；在 SVM 分类器中，学会里利用 cvxopt 解凸二次规划问题，也理解了核函数的作用，比较了不同核函数的区别。

在第二部分，手写了四层 MLP，复现了 MLP-mixer。通过手写四层 MLP，一步步推算逆向传播计算梯度，练习了矩阵求导的步骤，并熟悉了各层连接的方法、激活函数的作用和交叉熵的算法，最后也熟悉了 pytorch 自动求导的方法和使用 pytorch 需要注意的地方。通过复现 MLP-mixer，阅读了论文原文并且浏览了大量相关文章，练习了检索资料的能力，也熟悉了 pytorch 在网络结构上的简便应用，最后也对比了若干 pytorch 优化器的差别。