AI Lab2

PB18111699

魏钊

linearClassification:

参考博客:

https://zhuanlan.zhihu.com/p/92764814

https://www.cnblogs.com/nowgood/p/lagrangemultipy1.html

https://blog.csdn.net/dou3516/article/details/78795721

$$\mathbf{X}=\begin{bmatrix}1 & x_{11} & \cdots & x_{1d}\\ \vdots & & \vdots & \\ 1 & x_{N1} & \cdots & x_{Nd}\end{bmatrix},\qquad \mathbf{y}=\begin{bmatrix}y_1\\ \vdots\\ y_N\end{bmatrix},\qquad \mathbf{w}=\begin{bmatrix}b\\ \vdots\\ w_d\end{bmatrix}$$

首先构造 x,w

X 为训练特征数据左接一列 1。

W 为随机生成。

```
x = np.c_[np.ones(train_features.shape[0]), train_features]  # 构建x矩阵
w = np.random.random(train_features.shape[1] + 1)  # 随机生成w, 其值在 (0, 1)
# print(w)
w = w.reshape(-1, 1)  # 转换为列向量
```

线性函数: $f(x)=w_0+w_1x_1+\ldots+w_nx_n=w^Tx$

损失函数: $J(w)=\sum_{i=1}^{n}(w^Tx^i-y^i)^2$

梯度: $\frac{\partial J(w)}{\partial w}=\sum_{i=1}^{n}2(w^Tx^i-y^i)*x_j^{(i)}$

更新规则: BGD: $w_j=w_j-2\alpha\sum_{i=1}^{n}(w^Tx^i-y^i)x_j^i$

$$J_{LS}(\theta) = \frac{1}{2}\|\Phi\theta - y\|^2$$
$$\min(J_{LS}(\theta)) \quad \text{约束条件} \ \|\theta\|^2 < R$$

该原始问题可以转化对偶问题

$$\max_{\lambda} \min_{\theta} \left[J_{LS}(\theta) + \frac{\lambda}{2}\left(\|\theta\|^2 - R\right) \right] \quad \text{约束条件} \lambda \geq 0 \tag{9}$$

lagrange 对偶问题的 拉格朗日乘子 λ 的解由 R 决定. 如果不根据 R 来决定 R, 而是直接指定的话, l_2 约束的最小二乘学习法的解 $\hat{\theta}$ 可以通过下式求得

$$\hat{\theta} = \arg\min_{\theta} \left[J_{LS}\theta) + \frac{\lambda}{2}\|\theta\|^2 \right] \tag{10}$$

$J_{LS}\theta)$ 表示的是训练样本的拟合程度, 与 $\frac{\lambda}{2}\|\theta\|^2$ 结合求最小值, 来防止训练样本的过拟合. l_2 正则化的最小二乘学习法也称为 岭回归 .

$$\frac{\partial(J_{LS}\theta) + \frac{\lambda}{2}\|\theta\|^2)}{\partial\theta} = \Phi^T(\Phi\theta - y) + \theta = 0$$
$$\hat{\theta} = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T y$$

再由上述公式开始进行迭代。

```
now_epochs = self.epochs  # 取训练轮数
while now_epochs > 0:
    xw = x.dot(w)  # 计算x*w
    xw_y = xw - train_labels  # 计算xw-y
    xw_y_T = xw_y.reshape(xw_y.shape[1], xw_y.shape[0])  # 转置
    gradient = 2*np.dot(xw_y_T, x)/len(train_features) + 2*self.Lambda * w.reshape(1, -1)  # 得到梯度
    w -= self.lr * gradient.reshape(-1, 1)
    now_epochs -= 1

self.w = w  # 训练结束得到w
```

根据下面公式进行预测:

- Prediction for $\mathbf{x}_0$

$$\hat{y} = \operatorname{sign}\left(\mathbf{w}^{*\top}\begin{bmatrix} 1 \\ \mathbf{x}_0 \end{bmatrix}\right) = \operatorname{sign}\left(\mathbf{y}^\top X^{+\top}\begin{bmatrix} 1 \\ \mathbf{x}_0 \end{bmatrix}\right)$$

```python
def predict(self, test_features):
    x = np.c_[np.ones(test_features.shape[0]), test_features]  # 构造x
    i = test_features.shape[0]  # 测试数据的数量
    Prediction = []  # 预测结果
    j = 0
    # 预测类别
    while j < i:
        y = x[j].dot(self.w)
        if y >= 2.5:
            Prediction.append(3)
        elif y >= 1.75:
            Prediction.append(2)
        else:
            Prediction.append(1)
        j += 1
    Prediction = np.array(Prediction).reshape(i, 1)  # 格式化
    return Prediction
```

这里 1.75 和 2.5 的选择是经过几次测试后发现，在这两个值的情况下正确率较高。

测试结果：

```
C:\Python_Anaconda\envs\myImpl.py\python.exe C:/Users/Lucifer.dark/Desktop/2021春/人工智能/LAB2/src1/linearclassification.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6408952187182095
0.754880694143167
0.5850673194614443
0.6308139534883721
macro-F1: 0.6569206556976611
micro-F1: 0.6408952187182095

Process finished with exit code 0
```

预测成功率可达到 64%左右。

nBayesClassifier：

依据下图进行训练：

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

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

这里对性别当成离散型计算，对其他属性当作连续属性计算，使用高斯分布估计条件概率。

先进行预处理：

```
#################预处理数据#################
D_c = {}  # 统计各类的总数
for i in range(1, 4):  # 初始化为0
    D_c[i] = 0
D_c_sex = {}  # 统计各类不同性别的数量
for i in range(1, 4):  # 初始化为0
    for j in range(1, 4):
        D_c_sex[i, j] = 0
column = {}  # 不同类的连续型属性的值集合
for i in range(traindata.shape[0]):  # 遍历数据，统计各种数量
    D_c[int(trainlabel[i])] += 1  # 各类的总数
    D_c_sex[(int(trainlabel[i]), int(traindata[i][0]))] += 1  # 各类不同性别的总数

    for j in range(1, 8):  # 统计连续型属性
        if (int(trainlabel[i]), j) not in column.keys():
            column[int(trainlabel[i]), j] = np.array(float(traindata[i][j]))  # 第一次遇到该类属性
        else:
            column[int(trainlabel[i]), j] = np.append(column[int(trainlabel[i]), j],
                                                      float(traindata[i][j]))  # 后续遇到进行加入即可
Sum_D = D_c[1] + D_c[2] + D_c[3]  # 总数
#################预处理完成#################
```

这里统计了给类别各属性的数量。

上图中包含完整注释。

预处理完成后，进行概率计算：

```
#################计算概率#################
for i in range(1, 4):  # 计算先验概率
    self.Pc[i] = math.log((D_c[i] + 1) / (Sum_D + 3))
    for j in range(0, 8):  # 计算条件概率
        if j == 0:  # 性别
            for m in range(1, 4):
                self.Pxc[i, j, m] = math.log((D_c_sex[i, m] + 1) / (D_c[i] + 3))
        else:  # 连续型属性，这里我使用高斯分布来表示
            avg = np.average(column[i, j])  # 计算平均值
            var = np.var(column[i, j])  # 计算方差
            self.Pxc[i, j] = (avg, var)
#################计算完毕#################
```

特别注意，各种概率都转换为了 log 型。

连续型使用了高斯分布表示。

下面进行预测：

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

即找到使目标函数最大的参数类别 c。

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

根据上面公式计算概率。

```python
y = []
for i in range(features.shape[0]):
    argmax = 0  # 初始化目标最大值
    c = 0  # 初始化类别
    for j in range(1, 4):
        h = self.Pc[j] + self.Pxc[j, 0, int(features[i][0])]  # 因为都转换为log形式了，所以这里使用加法，先验概率加性别概率
        for m in range(1, 8):
            (avg, var) = self.Pxc[j, m]
            std = np.sqrt(var)
            #####计算高斯分布概率######
            t = 1 / (((2 * math.pi) ** 0.5) * std)
            e = math.exp(-0.5 * ((features[i][m] - avg) ** 2) / var)
            h += math.log(t * e)
            #####计算结束######
        if h > argmax:  # 比较预测结果
            argmax = h
            c = j
    y.append(c)
y = np.array(y).reshape(features.shape[0], 1)
```

因为都转化为了 log 形式，这里进行的是加法而不是乘法。

训练结果：

```
C:\Python_Anaconda\envs\myImpl.py\python.exe C:/Users/Lucifer.dark/Desktop/2021春/人工智能/LAB2/src1/nBayesClassifier.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.5839267548321465
0.7038461538461537
0.4725111441307578
0.6244952893674293
macro-F1: 0.6002841957814469
micro-F1: 0.5929752066115703

Process finished with exit code 0
```

准确率在 60%左右。

SVM：

参考博客：

https://blog.csdn.net/QW_sunny/article/details/79793889

https://blog.csdn.net/weixin_35755640/article/details/113660632

原问题等价于如下：

$$\min_{\alpha} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j x_i^T x_j - \sum_{i=1}^{m} \alpha_i$$

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

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

将其化为如下形式即可求解：

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & Gx \preceq h \\ & Ax = b \end{array}$$

首先进行预处理计算所需的 p,q,G,h,A,b：

```python
#####首先构造矩阵P######
p = np.ones((train_data.shape[0], train_data.shape[0]))
for i in range(train_data.shape[0]):
    for j in range(train_data.shape[0]):
        p[i][j] = train_label[i] * train_label[j] * self.KERNEL(train_data[i], train_data[j], self.kernel)
#####构造q,即为全为-1的列向量#####
q = -1 * np.ones((train_data.shape[0], 1))
#####构造h，即为C的列向量和0的列向量的拼接#####
h = self.C * np.ones((train_data.shape[0], 1))
h = np.r_[h, np.zeros((train_data.shape[0], 1))]
#####构造G，要同时满足小于等于h且大于0,即为单位对角阵和负单位对角阵的拼接#####
G = np.eye(train_data.shape[0], dtype=int)
G = np.r_[G, -1 * G]
####构造A，即为y的行向量形式####
A = train_label.reshape(1, train_data.shape[0])
####构造b，即为0向量####
b = np.zeros((1, 1))
```

上述代码有详尽注释。

之后利用线性规划求解器求解 alpha:

```
####使用线性规划求解器求解####
#####先进行类型统一#####
p = p.astype(np.double)
q = q.astype(np.double)
G = G.astype(np.double)
h = h.astype(np.double)
A = A.astype(np.double)
b = b.astype(np.double)
####进行求解####
solver = cvxopt.solvers.qp(cvxopt.matrix(p), cvxopt.matrix(q), cvxopt.matrix(G), cvxopt.matrix(h),
                           cvxopt.matrix(A),
                           cvxopt.matrix(b))
alpha = np.array(solver['x'])  # 求解得到alpha
```

利用如下公式恢复 b:

$$b = y_i - \sum_{j=1}^{n} \alpha_j y_j K\left(\mathbf{x}_i, \mathbf{x}_j\right) \qquad \text{for any } i \text{ that } \alpha_i \neq 0$$

```
index = np.where(alpha >= self.Epsilon)[0]  # 找到所有值不低于阈值的index
# 利用alpha计算b
b = np.mean(
    [train_label[i] - sum(
        [train_label[i] * alpha[i] * self.KERNEL(x, train_data[i], self.kernel) for x in train_data[index]])
     for i in index])
```

最后利用如下公式进行预测:

$$y^* = \operatorname{sign}\left(\sum_{i \in SV} \alpha_i y_i K\left(\mathbf{x}_i, \mathbf{x}'\right) + b\right)$$

```
####进行预测####
predictions = []
for j in range(test_data.shape[0]):
    y = b + sum(
        [train_label[i] * alpha[i] * self.KERNEL(test_data[j], train_data[i], self.kernel) for i in index])
    predictions.append(y)
y = np.array(predictions).reshape(test_data.shape[0], 1)
return y
```

测试结果如下:

```
C:\Python_Anaconda\envs\myImpl.py\python.exe C:/Users/Lucifer.dark/Desktop/2021春/人工智能/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
Optimal solution found.
     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  4e-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  3e-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.
     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  4e-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.
Acc: 0.6581892166836215
0.7678571428571428
0.568733153638814
0.6804123711340206
macro-F1: 0.6723342225433259
micro-F1: 0.6581892166836215

Process finished with exit code 0
```

准确率在 65.8%左右。

MLP_manual:

首先随机生成各种所需的参数:

```
# 输入数据，随机生成
x = torch.rand(size=(100, 5), requires_grad=True)
y = torch.randint(3, size=(100, 1))
w1 = torch.rand(size=(4, 5), requires_grad=True)
w2 = torch.rand(size=(4, 4), requires_grad=True)
w3 = torch.rand(size=(3, 4), requires_grad=True)
```

初始化:

```
def __init__(self, x, y, w1, w2, w3, lr=0.01, epochs=500):
    self.x = x
    self.y = y
    self.lr = lr
    self.epochs = epochs
    self.w1 = w1
    self.w2 = w2
    self.w3 = w3
```

激活函数如下:

Sigmoid函数由下列公式定义

$$S(x) = \frac{1}{1+e^{-x}}$$

前向传播部分：

第一层到第二次（5-4）:

首先计算：WX

```
wx = torch.mm(self.w1, torch.transpose(self.x, 0, 1))  # 计算 wx12
```

使用激活函数计算 y2:

```
self.y2 = torch.transpose(torch.div(1, 1 + torch.exp(-wx)), 0, 1)  # 第二层的激活函数输出
```

4-4:

```
#####4-4#####
wx = torch.mm(self.w2, torch.transpose(self.y2, 0, 1))  # 计算 wx23
self.y3 = torch.transpose(torch.div(1, 1 + torch.exp(-wx)), 0, 1)  # 第三层的激活函数输出
```

4-3：

特别注意最后一次 Softmax 函数如下：

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

```
#####4-3#####
wx = torch.mm(self.w3, torch.transpose(self.y3, 0, 1))  # 计算 wx34
y4 = torch.exp(torch.transpose(wx, 0, 1))
s = y4.sum(1)  # 求分母上的和
self.y4 = torch.div(y4, s.reshape(-1, 1))  # 输出结果
```

Loss 的计算如下：

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

```
####求 loss####
self.loss = torch.zeros(1)  # 初始化
for j in range(self.x.shape[0]):
    self.loss = self.loss - torch.log(self.y4[j][self.y[j]])
self.loss = self.loss / self.x.shape[0]
```

反向传播部分：

从 W3 开始一直到 W1，根据以下公式完成计算：

$$(\ell' s_3')_i = \begin{cases} \hat{y}_i - 1, i = y \\ \hat{y}_i, i \neq y \end{cases}$$

$$\frac{\partial L}{\partial W_1} = (W_2^{\mathrm{T}}(W_3^{\mathrm{T}}(\ell' s_3')\odot s_2')\odot s_1')x^{\mathrm{T}}$$

$$\frac{\partial L}{\partial W_2} = (W_3^{\mathrm{T}}(\ell' s_3')\odot s_2')h_1^{\mathrm{T}}$$

$$\frac{\partial L}{\partial W_3} = (\ell' s_3')h_2^{\mathrm{T}}$$

$$f'(\mathbf{u}_l) = sigmoid'(\mathbf{u}_l) = sigmoid(\mathbf{u}_l)(1 - sigmoid(\mathbf{u}_l)) = \mathbf{y}_l(1-\mathbf{y}_l)$$

```
self.WL3 = self.y4
for j in range(self.y4.shape[0]):
    self.WL3[j][self.y[j]] = self.WL3[j][self.y[j]] - 1
self.WL2 = torch.mm(self.WL3, self.w3)
self.WL3 = torch.mm(torch.transpose(self.WL3, 0, 1), self.y3)
```

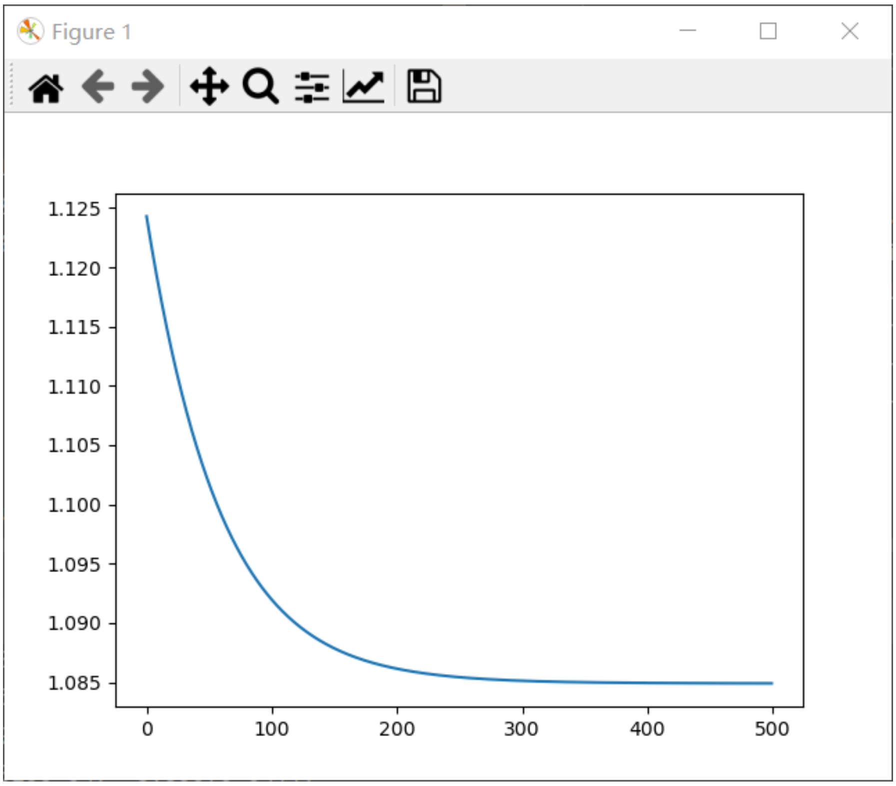
```
self.WL1 = torch.mm(self.WL2 * (self.y3 * (1 - self.y3)), self.w2)
self.WL2 = torch.mm(torch.transpose(self.WL2 * (self.y3 * (1 - self.y3)), 0, 1),
self.y2)
self.WL1 = torch.mm(torch.transpose(self.WL1 * (self.y2 * (1 - self.y2)), 0, 1),
self.x)
```

梯度下降部分：

梯度下降算法

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

```
loss = []
for i in range(self.epochs):
    self.Forwarding()
    temp = 0
    for j in range(self.x.shape[0]):
        temp = temp - math.log(self.y4[j][self.y[j]])
    temp = temp / self.x.shape[0]
    loss.append(temp)
    self.Backwarding()
    # 梯度下降
    self.w1 = self.w1 - self.lr * (self.WL1 / self.x.shape[0])
    self.w2 = self.w2 - self.lr * (self.WL2 / self.x.shape[0])
    self.w3 = self.w3 - self.lr * (self.WL3 / self.x.shape[0])
plt.plot(loss)
plt.show()
```



以上为 loss 曲线。

各参数矩阵的梯度如下（方便起见进行了一轮比较）：

```
C:\Python_Anaconda\python.exe C:/Users/Lucifer.dark/Desktop/2021春/人工智能/LAB2/src2/MLP_manual.py
自动计算W3：
tensor([[ 0.0201,  0.0184,  0.0174,  0.0207],
        [ 0.0836,  0.0772,  0.0711,  0.0851],
        [-0.1037, -0.0955, -0.0884, -0.1057]])
手动计算W3：
tensor([[ 0.0201,  0.0184,  0.0174,  0.0207],
        [ 0.0836,  0.0772,  0.0711,  0.0851],
        [-0.1037, -0.0955, -0.0884, -0.1057]], grad_fn=<DivBackward0>)
自动计算W2：
tensor([[ 0.0034,  0.0030,  0.0032,  0.0030],
        [ 0.0014,  0.0012,  0.0013,  0.0012],
        [-0.0067, -0.0059, -0.0062, -0.0059],
        [ 0.0028,  0.0025,  0.0026,  0.0025]])
手动计算W3：
tensor([[ 0.0034,  0.0030,  0.0032,  0.0030],
        [ 0.0014,  0.0012,  0.0013,  0.0012],
        [-0.0067, -0.0059, -0.0062, -0.0059],
        [ 0.0028,  0.0025,  0.0026,  0.0025]], grad_fn=<DivBackward0>)
自动计算W1：
tensor([[ 2.8901e-04,  8.6058e-05,  2.8591e-04,  5.0822e-04,  5.2968e-04],
        [-1.6404e-04, -8.9121e-05, -1.7803e-04, -1.8509e-04, -1.7736e-04],
        [ 2.7710e-04,  1.4336e-04,  3.0345e-04,  3.8993e-04,  4.5165e-04],
        [ 5.7307e-04,  2.5546e-04,  5.9403e-04,  8.0813e-04,  8.8359e-04]])
手动计算W1：
tensor([[ 2.8901e-04,  8.6058e-05,  2.8591e-04,  5.0822e-04,  5.2968e-04],
        [-1.6404e-04, -8.9121e-05, -1.7803e-04, -1.8509e-04, -1.7736e-04],
        [ 2.7710e-04,  1.4336e-04,  3.0345e-04,  3.8993e-04,  4.5165e-04],
        [ 5.7307e-04,  2.5546e-04,  5.9403e-04,  8.0813e-04,  8.8359e-04]],
       grad_fn=<DivBackward0>)

Process finished with exit code 0
```

可以看到手动计算的和自动计算得到的一样。

MLP_Mixer:

参考:

https://blog.csdn.net/u013468614/article/details/117220561?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522162604716016780271527677%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=162604716016780271527677&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_v2~rank_v29-7-117220561.first_rank_v2_pc_rank_v29&utm_term=mlp-mixer&spm=1018.2226.3001.4187

https://github.com/lucidrains/mlp-mixer-pytorch/blob/main/mlp_mixer_pytorch/mlp_mixer_pytorch.py

https://github.com/rishikksh20/MLP-Mixer-pytorch/blob/master/mlp-mixer.py

https://github.com/920232796/MlpMixer-pytorch/blob/master/MlpMixer/model.py

mixer_layer 部分:

```
# 这里需要写Mixer_Layer（layernorm，mlp1，mlp2，skip_connection）
self.patch_size = (28 // patch_size) ** 2
self.hidden_dim = hidden_dim
self.layernorm = nn.LayerNorm(self.hidden_dim)
# 行列交替两种类型的MLP
self.fn1 = nn.Sequential(
    nn.Linear(self.patch_size, self.hidden_dim * 3),
    nn.GELU(),
    nn.Dropout(0),
    nn.Linear(self.hidden_dim * 3, self.patch_size),
    nn.Dropout(0)
)
self.fn2 = nn.Sequential(
    nn.Linear(self.hidden_dim, self.hidden_dim * 3),
    nn.GELU(),
    nn.Dropout(0),
    nn.Linear(self.hidden_dim * 3, self.hidden_dim),
    nn.Dropout(0)
)
```

```
def forward(self, x):
    ##################################################
    temp1 = torch.transpose(self.layernorm(x), 1, 2)
    temp1 = self.fn1(temp1)
    temp1 = torch.transpose(temp1, 1, 2) + x
    temp2 = self.fn2(self.layernorm(temp1))
    return temp2 + temp1
```

MLPMixer 部分:

利用卷积实现:

```
# 对图片进行拆分
self.folding = nn.Conv2d(kernel_size=patch_size, stride=patch_size, in_channels=1,
                     out_channels=hidden_dim)
self.mixer_layer = nn.ModuleList(
    [Mixer_Layer(patch_size=patch_size, hidden_dim=hidden_dim) for i in range(depth)])
self.layernorm = nn.LayerNorm(hidden_dim)
self.Classifier = nn.Linear(hidden_dim, 10)
```

```
def forward(self, data):
    ##########################################################################
    # 注意维度的变化
    temp = self.folding(data)
    temp = torch.transpose(temp.view(temp.shape[0], temp.shape[1], temp.shape[2] *
temp.shape[3]), 1, 2)
    for f in self.mixer_layer:
        temp = f(temp)
    return self.Classifier(self.layernorm(temp).mean(dim=1))
```

train:

```
optimizer.zero_grad()
loss = criterion(model(data), target)
loss.backward()
optimizer.step()
```

test:

```
        for i in range(model(data).shape[0]):
                if torch.max(model(data), 1)[1][i] == target[i]:
                        num_correct = num_correct + 1
        test_loss = test_loss + criterion(model(data), target)
accuracy = num_correct / len(test_loader.dataset)
test_loss = test_loss / len(test_loader.dataset)
```

main:

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

测试:

```
C:\Python_Anaconda\python.exe C:/Users/Lucifer.dark/Desktop/2021春/人工智能/LAB2/src2/MLP_Mixer.py
Train Epoch: 0/5 [0/60000]  Loss: 2.356595
Train Epoch: 0/5 [12800/60000]  Loss: 1.010329
Train Epoch: 0/5 [25600/60000]  Loss: 0.561464
Train Epoch: 0/5 [38400/60000]  Loss: 0.439387
Train Epoch: 0/5 [51200/60000]  Loss: 0.262845
Train Epoch: 1/5 [0/60000]  Loss: 0.214819
Train Epoch: 1/5 [12800/60000]  Loss: 0.155095
Train Epoch: 1/5 [25600/60000]  Loss: 0.189344
Train Epoch: 1/5 [38400/60000]  Loss: 0.253254
Train Epoch: 1/5 [51200/60000]  Loss: 0.106105
Train Epoch: 2/5 [0/60000]  Loss: 0.179545
Train Epoch: 2/5 [12800/60000]  Loss: 0.091536
Train Epoch: 2/5 [25600/60000]  Loss: 0.189698
Train Epoch: 2/5 [38400/60000]  Loss: 0.159349
Train Epoch: 2/5 [51200/60000]  Loss: 0.123501
Train Epoch: 3/5 [0/60000]  Loss: 0.266544
Train Epoch: 3/5 [12800/60000]  Loss: 0.154652
Train Epoch: 3/5 [25600/60000]  Loss: 0.125598
Train Epoch: 3/5 [38400/60000]  Loss: 0.103392
Train Epoch: 3/5 [51200/60000]  Loss: 0.107305
Train Epoch: 4/5 [0/60000]  Loss: 0.114238
Train Epoch: 4/5 [12800/60000]  Loss: 0.148439
Train Epoch: 4/5 [25600/60000]  Loss: 0.102706
Train Epoch: 4/5 [38400/60000]  Loss: 0.132604
Train Epoch: 4/5 [51200/60000]  Loss: 0.237750
Test set: Average loss: 0.0011   Acc 0.96

Process finished with exit code 0
```

准确率达到了 96%。