中国科学技术大学计算机学院
《人工智能基础》实验报告

2021.07.14

实验题目：传统机器学习与深度学习

学生姓名：胡毅翔

学生学号：PB18000290

计算机实验教学中心制

2019 年 9 月

目录

1 实验目的

1. 实现线性分类器算法；
2. 实现朴素贝叶斯分类器；
3. 实现 SVM 分类器；
4. 手写实现感知机模型并进行反向传播；
5. 实现 MLP-Mixer。

2 实验环境

1. PC 一台;
2. Windows 10 操作系统;
3. Python 3.8.1;
4. Pytorch 1.8.1+cpu。

3 传统机器学习

3.1 线性分类器算法

3.1.1 最小二乘问题

$$\begin{aligned} Loss &= \min_w (Xw - y)^2 + \lambda \|w\|^2 \\ \frac{\partial Loss}{\partial w} &= 2(\mathbf{X}\mathbf{w} - \mathbf{y})^\top \mathbf{X} + 2\lambda \mathbf{I} w^\top = 0 \\ w^* &= (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{\top -1} \mathbf{X}^\top y \end{aligned} \tag{1}$$

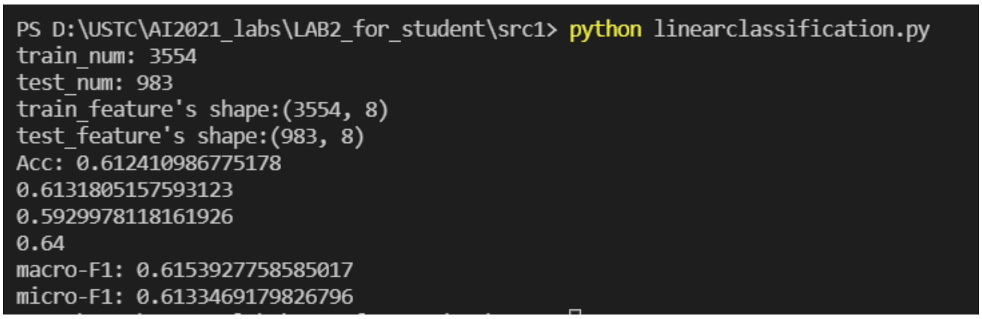
3.1.2 算法部分

基于上述推导的结果，完成的代码如下：

```
'''根据训练数据train_features,train_labels计算梯度更新参数W'''

def fit(self, train_features, train_labels):
# 参数weight, bias初始化
self.weight = np.array(np.random.rand(
train_features.shape[1], train_labels.shape[1]))
self.bias = np.zeros(train_labels.shape[1])
# 计算更新时用的矩阵
x_T = train_features.transpose()
```

```
a = x_T.dot(train_features)
a = a + self.Lambda * \
np.identity(train_features.shape[1], dtype=np.float32)
a = a.transpose()
a = np.linalg.inv(a)
a = a.dot(x_T)
# 训练 迭代
for i in range(self.epochs):
# 预测
y_pred = self.predict(train_features)
# 计算梯度
d_w = a@(y_pred - train_labels)
d_b = np.mean(y_pred-train_labels)
# 更新参数
self.weight = self.weight - self.lr * d_w
self.bias = self.bias - self.lr * d_b

'''根据训练好的参数对测试数据test_features进行预测，返回预测结果
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 predict(self, test_features):
y_pred_proba = test_features.dot(self.weight)+self.bias
y_pred = y_pred_proba
# 预测
for k in range(y_pred_proba.shape[0]):
y_pred[k][0] = math.floor(y_pred_proba[k][0])
return y_pred
```

运行结果截图如下:



3.2 朴素贝叶斯分类器

3.2.1 实验原理

1. 使用拉普拉斯平滑计算条件概率和先验概率:

$$\begin{aligned}\hat{P}(c) &= \frac{|D_c|+1}{|D|+N} \\ \hat{P}\left(x_i \mid c\right) &= \frac{|D_{c,x_i}|+1}{|D_c|+N_i}\end{aligned} \tag{2}$$

   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。

2. 判定准则为

$$h_{n\ b}(x) = \operatorname{argmax}_{c\in\mathcal{Y}} P(c)\prod_{i=1}^{d} P\left(x_i \mid c\right)$$

对于连续变量，假设服从高斯分布，用训练数据估计对应于每个类的均值 μ 和方差 σ^2。

3.2.2 算法部分

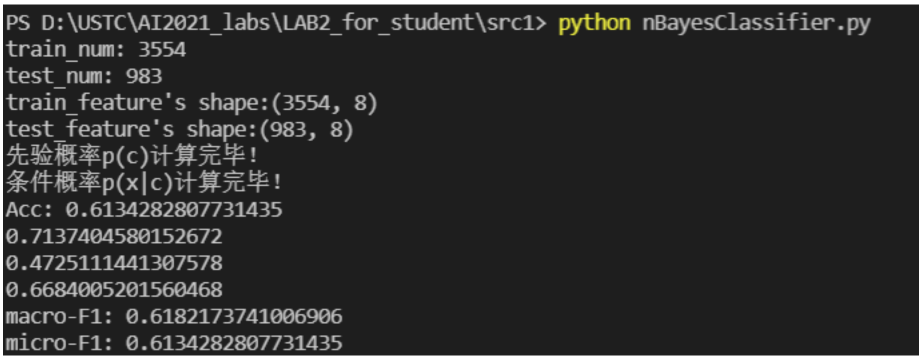
基于上述推导的结果，完成的代码如下:

```
'''
通过训练集计算先验概率分布p(c)和条件概率分布p(x|c)
建议全部取log，避免相乘为0
'''

def fit(self, traindata, trainlabel, featuretype):
# 计算先验概率
Py = {}
yi = {}
ySet = np.unique(trainlabel)
for i in ySet:
# 统计类别为i的数据量
Py[i] = (sum(trainlabel == i)+1)/(trainlabel.shape[0]+len(ySet))
yi[i] = sum(trainlabel == i)
# 保存Pc: P(c) 每个类别c的概率分布
self.Pc = Py
ySet = yi
print("先验概率p(c)计算完毕！")
# 计算条件概率
Pxy = {}

for xIdx in range(len(featuretype)):
# 第一层不同的属性/特征
Xarr = traindata[:, xIdx]
if featuretype[xIdx] == 0:
```

```
# 离散型特征
categoryParams = {}
XiSet = np.unique(Xarr)
XiSetCount = XiSet.size
for yj, yiCount in ySet.items():
# 第二层是不同的分类标签
categoryParams[yj] = {}
Xiyi = Xarr[np.nonzero(trainlabel == yj)[0]]
for xi in XiSet:
# 第三层是变量X的不同值类型
tmp = (sum(Xiyi == xi)+1)/(Xiyi.size+XiSetCount)
categoryParams[yj][xi] = tmp
Pxy[xIdx] = categoryParams
else:
# 连续型特征
continuousParams = {}
for yk, yiCount in ySet.items():
# 第二层是不同的分类标签
Xiyi = Xarr[np.nonzero(trainlabel == yk)[0]]
continuousParams[yk] = (Xiyi.mean(), Xiyi.std())
Pxy[xIdx] = continuousParams
print("条件概率p(x|c)计算完毕！")
# 保存Pxc: P(c|x) 每个特征的条件概率
self.Pxc = Pxy
return

'''
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):
m, n = features.shape
log_proba = np.zeros((m, len(self.Pc)))
for i in range(m):
# 第一层每个样本
for idx, (yi, Py) in enumerate(self.Pc.items()):
# 第二层不同类别
# 取对数计算
log_proba_idx = 0
for xIdx in range(n):
# 第三层每种特征
xi = features[i, xIdx]
if featuretype[xIdx] == 0:
log_proba_idx += np.log(self.Pxc[xIdx][yi][xi])
else:
miu = self.Pxc[xIdx][yi][0]
sigma = self.Pxc[xIdx][yi][1]
t = np.exp(-(xi-miu)**2/(2*sigma**2)) / \
```

```
    (np.power(2*np.pi, 0.5)*sigma)
    log_proba_idx += np.log(t)
    log_proba[i, idx] = log_proba_idx+np.log(Py)
    # 选择可能性最大的
    a = np.argmax(log_proba, axis=1)
    a = a + 1
    return a.reshape(m, 1)
```

运行结果截图如下:



```
PS D:\USTC\AI2021_labs\LAB2_for_student\src1> python nBayesClassifier.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
先验概率p(c)计算完毕!
条件概率p(x|c)计算完毕!
Acc: 0.6134282807731435
0.7137404580152672
0.4725111441307578
0.6684005201560468
macro-F1: 0.6182173741006906
micro-F1: 0.6134282807731435
```

3.3 SVM 分类器

3.3.1 实验原理

课堂中所学的 SVM 分类器用于解决的时二分类问题。本次实验需要解决的是 K 分类问题。

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。

因此在预测时,计算的返回值不需要经过 sign() 函数。

在求解时,需要用到凸优化包 cvxopt,求解线性规划问题,调用 Cvxopt.solvers.qp(P,q,G,h,A,b),求解的问题格式如下:

$$
\begin{aligned}
&\text{minimize} \quad (1/2)x^T P x + q^T x \\
&\text{subject to } Gx \preceq h \\
&Ax = b
\end{aligned}
\tag{3}
$$

或

$$
\begin{array}{ll}
\text{maximize} & -(1/2)\left(q + G^T z + A^T y\right)^T P^{\dagger}\left(q + G^T z + A^T y\right) - h^T z - b^T y \\
\text{subject to} & q + G^T z + A^T y \in \operatorname{range}(P) \\
& z \succeq 0
\end{array}
\tag{4}
$$

例如：

$$\begin{aligned}
\min \quad & 2x_1^2 + x_2^2 + x_1x_2 + x_1 + x_2 \\
\text{s.t.} \quad & x_1 \geq 0 \\
& x_2 \geq 0 \\
& x_1 + x_2 = 1
\end{aligned} \tag{5}$$

或

$$\begin{aligned}
\min \quad & 2x_1^2 + x_2^2 + x_1x_2 + x_1 + x_2 \\
\text{s.t.} \quad & -x_1 \leq 0 \\
& -x_2 \leq 0 \\
& x_1 + x_2 = 1
\end{aligned} \tag{6}$$

3.3.2 算法部分

基于上述推导的结果，完成的代码如下：

```
def fit(self, train_data, train_label, test_data):
# 构造核矩阵K, Kij = <yixi,yjxj>
train_num = train_data.shape[0]
K = np.zeros((train_num, train_num))
temp = train_data
for i in range(train_num):
for j in range(train_num):
K[i][j] = self.KERNEL(
temp[i], temp[j], self.kernel)*train_label[i]*train_label[j]

# 构造q
q = np.ones((train_num, 1))
q = -1*q
# 构造G
G1 = np.eye(train_num, dtype=int)
G2 = np.eye(train_num, dtype=int)
G2 = -1*G2
G = np.r_[G1, G2]
# 构造h
h1 = np.zeros((train_num, 1))
for i in range(train_num):
h1[i] = self.C
h2 = np.zeros((train_num, 1))
h = np.r_[h1, h2]
# 构造A
A = train_label.reshape(1, train_num)
# 构造b
b = np.zeros((1, 1))
# 统一数据类型
K = K.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)
# 化成cvxopt.matrix格式
K_1 = cvxopt.matrix(K)
q_1 = cvxopt.matrix(q)
G_1 = cvxopt.matrix(G)
h_1 = cvxopt.matrix(h)
A_1 = cvxopt.matrix(A)
b_1 = cvxopt.matrix(b)
# 用cvxopt（凸优化包）求解约束方程
sol = cvxopt.solvers.qp(K_1, q_1, G_1, h_1, A_1, b_1)
sol_x = sol['x']
alpha = np.array(sol_x)
indices = np.where(alpha > self.Epsilon)[0]
bias = np.mean(
[train_label[i] - sum([train_label[i] * alpha[i] * self.KERNEL(x, train_data[i], self.kernel) for x
                                                    in train_data[indices]]) for i in indices])
test_num = test_data.shape[0]
# 预测
predictions = []
for j in range(test_num):
prediction = bias + sum([train_label[i] * alpha[i] * self.KERNEL(
test_data[j], train_data[i], self.kernel) for i in indices])
predictions.append(prediction)
prediction = np.array(predictions).reshape(test_num, 1)
return prediction
```

核函数类型为 Linear 时，运行结果截图如下：

```
 5: -1.8550e+03 -2.0397e+03  2e+02  2e-03  4e-13
 6: -1.8649e+03 -2.0232e+03  2e+02  2e-03  4e-13
 7: -1.9015e+03 -1.9629e+03  6e+01  4e-04  5e-13
 8: -1.9107e+03 -1.9486e+03  4e+01  1e-04  5e-13
 9: -1.9125e+03 -1.9453e+03  3e+01  8e-05  4e-13
10: -1.9211e+03 -1.9341e+03  1e+01  1e-05  5e-13
11: -1.9252e+03 -1.9293e+03  4e+00  3e-06  5e-13
12: -1.9267e+03 -1.9276e+03  9e-01  4e-07  5e-13
13: -1.9271e+03 -1.9272e+03  9e-02  4e-08  5e-13
14: -1.9271e+03 -1.9271e+03  4e-03  2e-09  5e-13
15: -1.9271e+03 -1.9271e+03  4e-05  2e-11  5e-13
Optimal solution found.
Acc: 0.6581892166836215
0.7678571428571428
0.568733153638814
0.6804123711340206
macro-F1: 0.6723342225433259
micro-F1: 0.6581892166836215
```

核函数类型为 Poly 时，运行结果截图如下：

```
10: -1.8486e+03 -1.8987e+03  5e+01  1e-04  3e-12
11: -1.8545e+03 -1.8887e+03  3e+01  5e-05  3e-12
12: -1.8580e+03 -1.8831e+03  3e+01  3e-05  3e-12
13: -1.8601e+03 -1.8795e+03  2e+01  2e-05  3e-12
14: -1.8629e+03 -1.8754e+03  1e+01  9e-06  3e-12
15: -1.8642e+03 -1.8736e+03  9e+00  6e-06  3e-12
16: -1.8667e+03 -1.8702e+03  4e+00  1e-06  3e-12
17: -1.8679e+03 -1.8689e+03  1e+00  1e-07  3e-12
18: -1.8683e+03 -1.8684e+03  8e-02  7e-09  3e-12
19: -1.8683e+03 -1.8683e+03  1e-02  9e-10  3e-12
20: -1.8683e+03 -1.8683e+03  4e-04  2e-11  3e-12
Optimal solution found.
Acc: 0.6449643947100712
0.750551876379691
0.5717948717948718
0.6575716234652115
macro-F1: 0.6599727905465914
micro-F1: 0.6449643947100712
```

核函数类型为 Gauss 时，运行结果截图如下：

```
17: -1.8768e+03 -1.8771e+03  3e-01  1e-08  5e-14
18: -1.8769e+03 -1.8769e+03  1e-01  1e-09  5e-14
19: -1.8769e+03 -1.8769e+03  5e-02  2e-10  5e-14
20: -1.8769e+03 -1.8769e+03  7e-03  2e-11  5e-14
21: -1.8769e+03 -1.8769e+03  2e-04  1e-13  5e-14
Optimal solution found.
Acc: 0.6561546286876907
0.755056179775281
0.570673712021136
0.6832460732984293
macro-F1: 0.6696586550316154
micro-F1: 0.6561546286876907
```

4 深度学习

4.1 感知机模型

4.1.1 实验原理

多层感知机由感知机推广而来，最主要的特点是有多个神经元层，因此也叫深度神经网络 (DNN: Deep Neural Networks)。

多层感知机的一个重要特点就是多层，我们将第一层称之为输入层，最后一层称之有输出层，中间的层称之为隐层。MLP 并没有规定隐层的数量，因此可以根据各自的需求选择合适的隐层层数。且对于输出层神经元的个数也没有限制。

首先进行 MLP 梯度推导。

单输出感知器模型的运算法则：

$$
\begin{aligned}
y &= XW + b \\
y &= \sum x_i * w_i + b
\end{aligned} \tag{7}
$$

输入 X 乘以权重 W 得到 y，再通过激活函数得到输出 (O)。在这里，激活函数是 sigmoid 函数。

$$\sigma(y) = \frac{1}{1+e^{-y}} \tag{8}$$

E 是 loss 函数值，这里是输出值 (output) 与真实值 (target) 的欧式距离。

$$E = \frac{1}{2}\left(O_0^1 - t\right)^2 \tag{9}$$

E 的大小是评价感知器模型好坏的指标之一，w 权重是描述这个感知器模型的参数，通过计算 E 来优化感知器模型，即优化 w 的值。w_{jk}^I 表示第 I 层，第 j 个输入链接第 k 个输出的权值 w。以下先对一个权重 (值)w 求得感知器模型的梯度。

$$\begin{aligned}
\frac{\partial E}{\partial w_{j0}^1} &= \left(O_0^1 - t\right)\frac{\partial O_0^1}{\partial w_{j0}^1} \\
\frac{\partial E}{\partial w_{j0}^1} &= \left(O_0^1 - t\right)\frac{\partial \sigma\left(x_0^1\right)}{\partial w_{j0}^i} \\
\frac{\partial E}{\partial w_{j0}^1} &= \left(O_0^1 - t\right)\frac{\partial \sigma\left(x_0^1\right)}{\partial x_0^1}\frac{\partial x_0^1}{\partial w_{j0}^1}
\end{aligned} \tag{10}$$

红色部分使用了链式法则，链式法则其实就是连续求导，这里将求得 sigmoid 函数的导数。sigmoid 导数如何求，将在下一环节展示。求得的 sigmoid 导数是 $s\prime(x) = s(x)(1 - s(x))$。

$$\begin{aligned}
\frac{\partial E}{\partial w_{j0}^1} &= \left(O_0^1 - t\right)\sigma\left(x_0^1\right)\left(1 - \sigma\left(x_0^1\right)\right)\frac{\partial x_0^1}{\partial w_{j0}^1} \\
\frac{\partial E}{\partial w_{j0}^1} &= \left(O_0^1 - t\right)O_0^1\left(1 - O_0^1\right)\frac{\partial x_0^1}{\partial w_{j0}^1} \\
\frac{\partial E}{\partial w_{j0}^1} &= \left(O_0^1 - t\right)O_0^1\left(1 - O_0^1\right)x_j^0
\end{aligned} \tag{11}$$

现在把单个输出的感知器模型推广成多输出感知器模型。按照上面的思路，求 w 的梯度：

$$\begin{aligned}
E &= \frac{1}{2}\sum\left(O_i^1 - t_i\right)^2 \\
\frac{\partial E}{\partial u_{jk}^1} &= \left(O_k^1 - t_k\right)\frac{\partial Q_k^1}{\partial w_{jk}^1}
\end{aligned} \tag{12}$$

W_{jk} 只对 O_k 值有贡献，对 O_i(i 不等于 k) 没有贡献。因此 O_i(i 不等于 k) 对 W_{jk} 的偏导为 0。换

句话说，相关梯度与输入结点有关。

$$\begin{aligned}
&\frac{\partial E}{\partial w_{jk}^{1}} = \left(O_{k}^{1} - t_{k}\right) \frac{\partial \sigma\left(x_{k}^{1}\right)}{\partial w_{jk}^{1}} \\
&\frac{\partial E}{\partial w_{jk}^{1}} = \left(O_{k}^{1} - t_{k}\right) \sigma\left(x_{k}^{1}\right)\left(1 - \sigma\left(x_{k}^{1}\right)\right) \frac{\partial x_{k}^{1}}{\partial w_{jk}^{1}} \\
&\frac{\partial E}{\partial w_{jk}^{1}} = \left(O_{k}^{1} - t_{k}\right) O_{k}^{1}\left(1 - O_{k}^{1}\right) x_{j}^{0}
\end{aligned} \tag{13}$$

下面推导 MLP 链式法则。

求导过程：

$$\begin{aligned}
&\left(\frac{f}{g}\right)^{\prime} = \frac{f^{\prime} g - f g^{\prime}}{g^{2}} \\
&\sigma(y) = \frac{1}{1+e^{-y}} \\
&\frac{\partial \sigma(y)}{\partial y} = \frac{0 *\left(1+e^{-y}\right) - 1 *\left(-e^{-y}\right)}{1+2 e^{-y}+e^{-2 y}} \\
&= \frac{e^{-y}}{\left(1+e^{-y}\right)^{2}} = \frac{1+e^{-y}-1}{\left(1+e^{-y}\right)^{2}} = \frac{1}{1+e^{-y}} - \frac{1}{\left(1+e^{-y}\right)^{2}} \\
&= \frac{1}{1+e^{-y}}\left(1 - \frac{1}{1+e^{-y}}\right) = \sigma(y)(1-\sigma(y))
\end{aligned} \tag{14}$$

链式法则如下：

$$\frac{\partial E}{\partial w_{jk}^{1}} = \frac{\partial E}{\partial O_{k}^{1}} \frac{\partial O_{k}^{1}}{\partial w_{jk}^{1}} = \frac{\partial E}{\partial O_{k}^{2}} \frac{\partial O_{k}^{2}}{\partial O_{k}^{1}} \frac{\partial O_{k}^{1}}{\partial w_{jk}^{1}} \tag{15}$$

最后推导 MLP 反向传播。

参考上面的公式推导，现在完成从 1 到 K 的泛化处理。首先根据前面的结论，梯度推导为：

$$\frac{\partial E}{\partial w_{jk}^{K}} = \left(O_{k}^{K} - t_{k}\right) O_{k}^{K}\left(1 - O_{k}^{K}\right) x_{j}^{K-1} \tag{16}$$

针对多层感知器，x 是该层的输入，即是上一层的输出 O。如果这里用 x 表示会有歧义，因为 O = sigmoid(x)，x 必须经过激活函数之后才会得到 O，所以对多层感知器而言，梯度推导为：

$$\frac{\partial E}{\partial w_{jk}^{K}} = \left(O_{k}^{K} - t_{k}\right) O_{k}^{K}\left(1 - O_{k}^{K}\right) O_{j}^{K-1} \tag{17}$$

现在，我们对结合上述结论，以及链式法则，完成下一层的梯度求导，即对 w_{ij}^J 求导。

$$
\begin{aligned}
\frac{\partial E}{\partial w_{ij}^J} &= \frac{\partial}{\partial w_{ij}^J}\frac{1}{2}\sum_{k\in K}\left(O_k^K - t_k\right)^2 \\
\frac{\partial E}{\partial w_{ij}^J} &= \sum_{k\in K}\left(O_k^K - t_k\right)\frac{\partial}{\partial w_{ij}^J}O_k^K \\
\frac{\partial E}{\partial w_{ij}^J} &= \sum_{k\in K}\left(O_k^K - t_k\right)\frac{\partial}{\partial w_{ij}^J}\sigma\left(x_k^K\right) \\
\frac{\partial E}{\partial w_{ij}^J} &= \sum_{k\in K}\left(O_k^K - t_k\right)\frac{\partial\sigma\left(x_k^K\right)}{\partial x_k^K}\frac{\partial x_k^K}{\partial w_{ij}^J} \\
\frac{\partial E}{\partial w_{ij}^J} &= \sum_{k\in K}\left(O_k^K - t_k\right)\sigma\left(x_k^K\right)\left(1-\sigma\left(x_k^K\right)\right)\frac{\partial x_k^K}{\partial w_{ij}^J}
\end{aligned}
\tag{18}
$$

先求到 K 层的梯度，继续使用链式法则，在 K 层的梯度上继续求第 J 层。

$$
\begin{aligned}
\frac{\partial E}{\partial w_{ij}^J} &= \sum_{k\in K}\left(O_k^K - t_k\right)O_k^K\left(1-O_k^K\right) \mid \frac{\partial x_k^K}{\partial O_j^J}\frac{\partial O_j^j}{\partial w_{ij}^J} \\
\frac{\partial E}{\partial w_{ij}^J} &= \sum_{k\in K}\left(O_k^K - t_k\right)O_k^K\left(1-O_k^K\right)\sqrt{w_{jk}^K}\partial\frac{\partial O_j^J}{\partial w_{ij}^J} \\
\frac{\partial E}{\partial w_{ij}^J} &= \frac{\partial O_j^J}{\partial w_{ij}^J}\sum_{k\in K}\left(O_k^K - t_k\right)O_k^K\left(1-O_k^K\right)w_{jk}^K \\
\frac{\partial E}{\partial w_{ij}^J} &= \frac{\partial\sigma\left(x_j^J\right)}{\partial x_j^J}\frac{\partial x_j^J}{\partial w_{ij}^J}\sum_{k\in K}\left(O_k^K - t_k\right)O_k^K\left(1-O_k^K\right)w_{jk}^K \\
\frac{\partial E}{\partial w_{ij}^J} &= O_j^J\left(1-O_j^J\right)\frac{\partial x_j^J}{\partial w_{ij}^J}\sum_{k\in K}\left(O_k^K - t_k\right)O_k^K\left(1-O_k^K\right)w_{jk}^K \\
\frac{\partial E}{\partial w_{ij}^J} &= O_j^J\left(1-O_j^J\right)O_i^I\sum_{k\in K}\left(O_k^K - t_k\right)O_k^K\left(1-O_k^K\right)w_{jk}^K
\end{aligned}
\tag{19}
$$

I，J，K 是连续的 3 层，即 J = I + 1，K = J + 1。现在开始对其简化。对第 K 层的第 k 结点有:

$$
\frac{\partial E}{\partial w_{jk}^K} = O_j^J\delta_k^K \tag{20}
$$

其中:

$$
\delta_k^K = \left(O_k^K - t_k\right)O_k^K\left(1-O_k^K\right) \tag{21}
$$

同理，对第 J(K = J + 1) 层的第 j 个结点有:

$$
\frac{\partial E}{\partial w_{ij}^J} = O_i^l\delta_j^J \tag{22}
$$

其中：

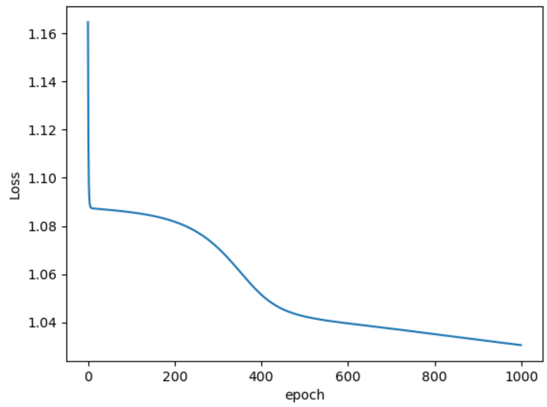
$$\delta_j^J = O_j^J\left(1 - O_j^J\right)\sum_{k\in K}\delta_k^K w_{jk}^K \tag{23}$$

现在开始说反向传播。如果通过前向通路求得个个参数的梯度十分困难和繁琐。比如需要先对第 I 层求得梯度，然后又求得第 J 层，最后是第 K 层。这样会重复计算，会造成资源浪费。如果使用反向传播的话，可以一次计算所有参数。如同上述推导过程一样，如果要对第 J 层（第 J 层不是输出层，第 K 层是输出层）求得梯度，那么必须要经过求得第 K 层（输出层）梯度基础上才行，第 I 层，I-1，...,0 层（输入层）同理。

4.1.2 算法部分

代码详见 MLP_manual.py。

运行结果截图如下：



4.2 MLPMixer

4.2.1 实验原理

MLP-Mixer 主要包括三部分：Per-patch Fully-connected、Mixer Layer、分类器。其中分类器部分采用传统的全局平均池化（GAP）+ 全连接层（FC）+Softmax 的方式构成，故不进行更多介绍，下面主要针对前两部分进行解释。

首先介绍全连接部分。

FC 相较于 Conv，并不能获取局部区域间的信息，为了解决这个问题，MLP-Mixer 通过 Per-patch Fully-connected 将输入图像转化为 2D 的 Table，方便在后面进行局部区域间的信息融合。

具体来说，MLP-Mixer 将输入图像相邻无重叠地划分为 S 个 Patch，每个 Patch 通过 MLP 映射为一维特征向量，其中一维向量长度为 C，最后将每个 Patch 得到的特征向量组合得到大小为 S*C 的 2D Table。需要注意的时，每个 Patch 使用的映射矩阵相同，即使用的 MLP 参数相同。

实际上，Per-patch Fully-connected 实现了（W，H，C）的向量空间到（S，C）的向量空间的映射。

例如，假设输入图像大小为 240*240*3，模型选取的 Patch 为 16*16，那么一张图片可以划分为（240*240）/（16*16）= 225 个 Patch。结合图片的通道数，每个 Patch 包含了 16*16*3 = 768 个值，把这 768 个值做 Flatten 作为 MLP 的输入，其中 MLP 的输出层神经元个数为 128。这样，每个 Patch 就可以得到长度的 128 的特征向量，组合得到 225*128 的 Table。MLP-Mixer 中 Patch 大小和 MLP 输出单元个数为超参数。

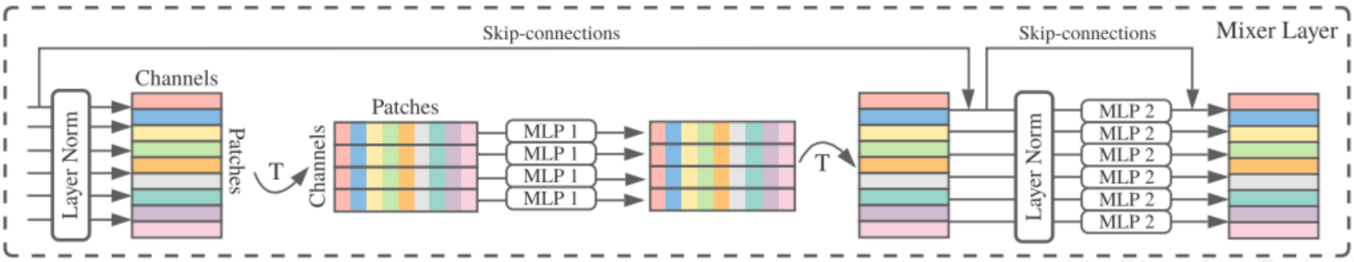
再介绍 Mixer Layer 部分。

观察 Per-patch Fully-connected 得到的 Table 会发现，Table 的行代表了同一空间位置在不同通道上的信息，列代表了不同空间位置在同一通道上的信息。换句话说，对 Table 的每一行进行操作可以实现通道域的信息融合，对 Table 的每一列进行操作可以实现空间域的信息融合。

在传统 CNN 中，可以通过 1*1 Conv 来实现通道域的信息融合，如果使用更大一点的卷积核，可以同时实现空间域和通道域的信息融合。

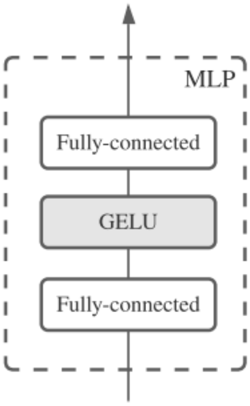
在 Transformer 中，通过 Self-Attention 实现空间域的信息融合，通过 MLP 同时实现空间域和通道域的信息融合。

而在 MLP-Mixer 中，通过 Mixer Layer 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，这种思想与 Xception 和 MobileNet 中的深度可分离卷积相似。

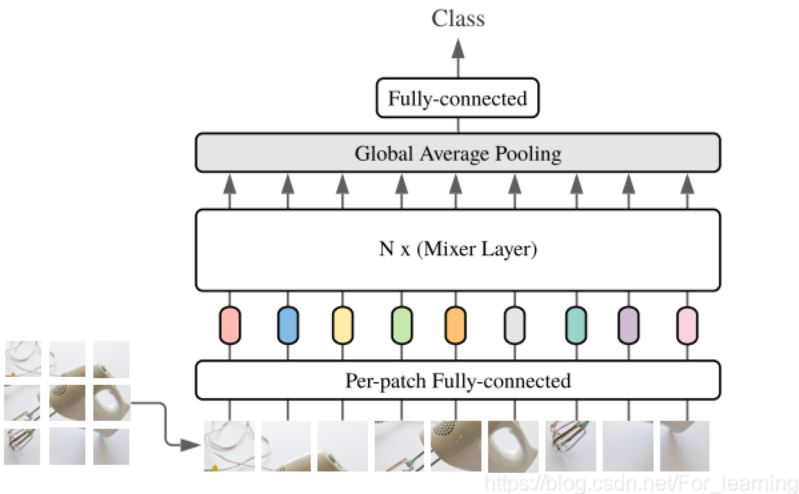


根据上述内容，MLP-Mixer 在 Mixer Layer 中使用分别使用 token-mixing MLPs（图中 MLP1）和 channel-mixing MLPs（图中 MLP2）对 Table 的列和行进行映射，与 Per-patch Fully-connected 相同，MLP1 和 MLP2 在不同列、行中的映射过程中共享权重。除此之外，Mixer Layer 还加入了 LN 和跳接来提高模型性能。

另外，MLP1 和 MLP2 都采用了相同的结构，如下图。



整体结构如下：



4.2.2 算法部分

基于上述原理，完成的代码如下：

```
class MLP(nn.Module):
    def __init__(self, dims, multiplxer=4):
        super(MLP, self).__init__()
        hidden = int(dims * multiplxer)

        self.out = nn.Sequential(
            nn.Linear(dims, hidden),
            nn.GELU(),
            nn.Linear(hidden, dims)
        )

    def forward(self, x):
        return self.out(x)
```

```python
class MixerLayer(nn.Module):
    def __init__(self, patch_size, hidden_dim):
        super(MixerLayer, self).__init__()
        seq = patch_size
        dims = hidden_dim
        # LayerNorm1
        self.layer_norm1 = nn.LayerNorm(dims)
        # mlp1
        self.mlp1 = MLP(seq, multiplxer=0.5)
        # LayerNorm2
        self.layer_norm2 = nn.LayerNorm(dims)
        # mlp2
        self.mlp2 = MLP(dims)

    def forward(self, x):
        out = self.layer_norm1(x).transpose(1, 2)
        out = self.mlp1(out).transpose(1, 2)
        out += x
        out2 = self.layer_norm2(out)
        out2 = self.mlp2(out2)
        out2 += out
        return out2


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'
        # 图片大小
        in_dims = 28
        # 维度
        dims = hidden_dim
        # 深度
        N = depth
        # 目标类别数
        n_classes = 10
        self.embedding = nn.Linear(in_dims, dims)
        self.layers = nn.ModuleList()
        for _ in range(N):
            self.layers.append(MixerLayer(in_dims, dims))
        self.gap = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(dims, n_classes)
        self.dims = dims

    def forward(self, x):
        out = self.embedding(x)
        out = out.permute(0, 2, 3, 1).view(x.size(0), -1, self.dims)
        for layer in self.layers:
```

```
            out = layer(out)
        out = out.mean(dim=1)
        out = self.fc(out)
        return out
# 定义训练函数


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):
            batch_size_train = data.shape[0]
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            pre_out = model(data)
            targ_out = torch.nn.functional.one_hot(target, num_classes=10)
            targ_out = targ_out.view((batch_size_train, 10)).float()
            loss = criterion(pre_out, targ_out)
            loss.backward()
            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()))

# 定义测试函数


def test(model, test_loader, criterion):
    model.eval()
    test_loss = 0
    num_correct = 0
    total = 0
    with torch.no_grad():
        for data, target in test_loader:
            batch_size_test = data.shape[0]
            data, target = data.to(device), target.to(device)
            pre_out = model(data)
            targ_out = torch.nn.functional.one_hot(target, num_classes=10)
            targ_out = targ_out.view((batch_size_test, 10)).float()
            test_loss += criterion(pre_out, targ_out)  # 将一批的损失相加
            t = pre_out.argmax(dim=1)
            num_correct += sum(t == target)
            total += batch_size_test
    # 准确率
    accuracy = num_correct/total
    # 平均损失
    test_loss /= len(test_loader.dataset)
    print("Test set: Average loss: {:.4f}\t Acc {:.2f}".format(
        test_loss, accuracy))
```

```
if __name__ == '__main__':
    n_epochs = 5
    batch_size = 128
    learning_rate = 0.001

    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)

    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)

    # device = torch.device("cpu")
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    #n = (28 * 28) // 4 ** 2
    model = MLPMixer(patch_size=4, hidden_dim=14, depth=12)
    model.to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    mse = nn.MSELoss()

    train(model, train_loader, optimizer, n_epochs, mse)

    test(model, test_loader, mse)
```

运行结果截图如下:

```
PS D:\USTC\AI2021_labs\LAB2_for_student\src2> python MLP_Mixer.py
Train Epoch: 0/5 [0/60000]      Loss: 0.142960
Train Epoch: 0/5 [12800/60000]  Loss: 0.049485
Train Epoch: 0/5 [25600/60000]  Loss: 0.031894
Train Epoch: 0/5 [38400/60000]  Loss: 0.021583
Train Epoch: 0/5 [51200/60000]  Loss: 0.019155
Train Epoch: 1/5 [0/60000]      Loss: 0.013864
Train Epoch: 1/5 [12800/60000]  Loss: 0.010951
Train Epoch: 1/5 [25600/60000]  Loss: 0.008961
Train Epoch: 1/5 [38400/60000]  Loss: 0.007933
Train Epoch: 1/5 [51200/60000]  Loss: 0.008113
Train Epoch: 2/5 [0/60000]      Loss: 0.002999
Train Epoch: 2/5 [12800/60000]  Loss: 0.008601
Train Epoch: 2/5 [25600/60000]  Loss: 0.001316
Train Epoch: 2/5 [38400/60000]  Loss: 0.005195
Train Epoch: 2/5 [51200/60000]  Loss: 0.006033
Train Epoch: 3/5 [0/60000]      Loss: 0.007407
Train Epoch: 3/5 [12800/60000]  Loss: 0.004103
Train Epoch: 3/5 [25600/60000]  Loss: 0.004994
Train Epoch: 3/5 [38400/60000]  Loss: 0.005090
Train Epoch: 3/5 [51200/60000]  Loss: 0.007462
Train Epoch: 4/5 [0/60000]      Loss: 0.001658
Train Epoch: 4/5 [12800/60000]  Loss: 0.003426
Train Epoch: 4/5 [25600/60000]  Loss: 0.004487
Train Epoch: 4/5 [38400/60000]  Loss: 0.004975
Train Epoch: 4/5 [51200/60000]  Loss: 0.002877
Test set: Average loss: 0.0000   Acc 0.97
```