人工智能第二次实验

PB18051113 林成渊 2021/7/15

实验内容

总体描述

本次实验包含传统机器学习与深度学习两部分。

实验目的

- 传统机器学习部分实现
  - 线性分类算法
  - 朴素贝叶斯分类器
  - SVM算法
- 深度学习部分实现
  - 手写感知机模型并进行反向传播
  - 复现MLP-Mixer
- 针对实现进行分析并撰写报告

实验环境

- Anaconda 3
- Python 3.6

实验过程

传统机器学习部分

实现一个线性分类算法

推导： 求解如下优化问题

$$\min_{w}(Xw-y)^2+\lambda\|w\|^2$$

令 $E_{\hat{\omega}}=(Xw-y)^2+\lambda\|w\|^2$ ，对 $\hat{\omega}$ 求导得到

$$\frac{\partial E_{\hat{\omega}}}{\partial\hat{\omega}}=2X^T(X\hat{\omega}-y)+2\lambda\hat{\omega}$$

令导数为 0 ，移项得到

$$(X^TX+\lambda)\hat{\omega}=X^Ty \qquad (1)$$

令 $(X^TX+\lambda)$ 的伪逆矩阵为 A ，那么可以解出

$$\hat{\omega}=AX^Ty$$

即所需要的解，实际上这一步就是求解 (1) 得到一组满足的值。

实际代码实现中，对 (1) 式子迭代求解 $\hat{\omega}$ ，根据实验源码说明，采用梯度下降方法

$$\omega_{k+1} = \omega_k - lr \times \frac{\partial E_{\hat{\omega}}}{\partial \hat{\omega}}$$

实际代码实现（第一轮）： 只需按照迭代式按部就班计算即可

```
'''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):
    # 采用梯度下降算法， x 是数据集扩展来的 X 矩阵， w 是扩展后的被迭代的矩阵
    x = np.c_[np.ones(train_features.shape[0]),train_features]
    w = np.zeros(train_features.shape[1] + 1)
    # 开始执行 epochs 次迭代
    for t in range(self.epochs):
        grad = 2 * np.dot(x.T, (np.dot(x, w.T) - train_labels.T[0]).T).T + 2 *
self.Lambda * w.T  # 计算梯度
        w = w - self.lr * grad  # 根据梯度下降
    self.w = w
......
'''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):
        test_num = test_features.shape[0]  # 测试数据的数目
        X = np.c_[np.ones(test_features.shape[0]), test_features]  # X 作为测试数据
        prediction = []  # 预测结果
        for temp in X:
            y_predict = np.dot(temp, self.w)
            # 将各个数据分到最近的一个类
            if y_predict < 1.5:
                prediction.append(1)
            elif y_predict < 2.5:
                prediction.append(2)
            else:
                prediction.append(3)
        prediction = np.array(prediction).reshape(test_num, 1)  # 封装结果
        return prediction
```

实验结果（第一轮）：

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.612410986775178
0.6217008797653959
0.6044071353620146
0.6190476190476191
macro-F1: 0.6150518780583432
micro-F1: 0.612410986775178
```

可见测试通过

分析： 这一写法对于参数的选取有比较严苛的要求，需要取非常低的 lr 值才能收敛（如图中所示运行，采纳了0.000005的lr值，而当lr = 0.05时是无法收敛的）。初步猜测可能是迭代的步长越过了收敛点从而越迭代越发散。注意到此次的代码才用的是批量梯度下降（BGD），考虑小批量梯度下降的方法，进入第二轮

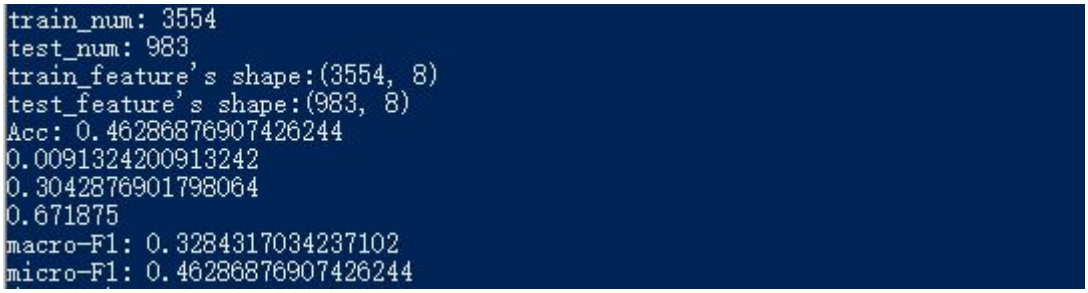
实际代码实现（第二轮）： 与第一轮的唯一不同在于，每次迭代对于每个数据均算一次梯度，如果数据有 n 个，迭代 m 次，那么就相当于计算了 nm 次梯度。

```
'''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):
```

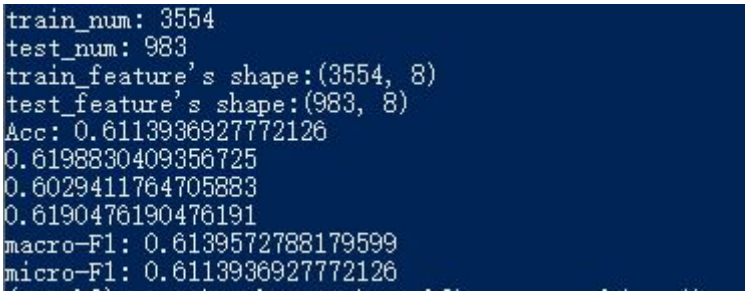
```
# 采用梯度下降算法， x 是数据集扩展来的 X 矩阵， w 是扩展后的被迭代的矩阵
    x = np.c_[np.ones(train_features.shape[0]),train_features]
    w = np.zeros(train_features.shape[1] + 1)
# 开始执行 epochs 次迭代
    for t in range(self.epochs):
        for row in range(train_features.shape[0]):
            grad = 2 * np.dot(x[row].T, (np.dot(x[row], w.T) - train_labels.T[0]
[row]).T).T + 2 * self.Lambda * w.T  # 计算梯度
            w = w - self.lr * grad  # 根据梯度下降
    self.w = w
......
'''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):
    test_num = test_features.shape[0]  # 测试数据的数目
    X = np.c_[np.ones(test_features.shape[0]), test_features]  # X 作为测试数据
    prediction = []  # 预测结果
    for temp in X:
        y_predict = np.dot(temp, self.w)
        # 将各个数据分到最近的一个类
        if y_predict < 1.5:
            prediction.append(1)
            elif y_predict < 2.5:
                prediction.append(2)
                else:
                    prediction.append(3)
                    prediction = np.array(prediction).reshape(test_num, 1)  # 封
装结果
                    return prediction
```

实验结果（第二轮）：

采用 lr = 0.5，运行得到结果如下，相对于第一轮，这种算法对于高 lr 值的适配性更好了



而取 lr = 0.000005，运行得到结果



同样得到了收敛，并且准略率与第一轮实现相近。

分析： 第二轮的实现运行时间非常长，但是在收敛性能上更佳。故可以此作为最终的线性分类-梯度下降实现方案。另外，对于更进一步的优化，如果仅仅考虑这个规模的数据集，事实上除了梯度下降以外，直接通过迭代求解也许会是更好的方案。即

$$X^T X\hat{\omega} = X^T y - \lambda\hat{\omega}$$

令 $X^T X$ 的伪逆矩阵为 A，于是可以通过迭代求出

$$\hat{\omega}_{k+1} = A(X^T y - \lambda\hat{\omega}_k)$$

如此设计是考虑到 λ 值较小，为了减小数据精度带来的损失，故而尽量规避容易将损失放大的迭代法（λ 作为除数）。基于该迭代式实现如下

```
def fit(self,train_features,train_labels):
    x = np.c_[np.ones(train_features.shape[0]),train_features]
    w = np.zeros(train_features.shape[1] + 1)
    for t in range(self.epochs):
        temp = np.dot(X.T,train_labels) - self.Lambda * w
        grad = np.dot(np.linalg.inv(np.dot(X.T,X)), temp) - w
        w = w + self.lr*grad.T
    self.w = w
```

运行结果

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
(3554,) (3554, 8)
Acc: 0.6266531027466938
0.6629526462395543
0.6015367727771679
0.6408045977011494
macro-F1: 0.6350980055726239
micro-F1: 0.6266531027466938
```

这并非梯度下降算法。事实上这一算法在本次实验规模的数据集上表现非常好，但是大规模数据集下计算伪逆矩阵会付出很大的开销，此时更适合采用梯度下降的算法（如上面的两种）

实现一个朴素贝叶斯分类器

思路：数据集中，对离散的属性，使用拉普拉斯平滑计算其条件概率和先验概率。而对于其他的取值连续的属性，使用高斯分布来表示它的类条件概率分布，即使用训练数据估计对应于每个类的均值 μ 和方差 σ。由于实验重点关注的是同一属性的概率的相对大小，因此可以直接使用概率密度函数来替代进行计算。

实际代码实现：首先按照朴素贝叶斯分类器的定义按部就班实现，具体细节已在注释中阐述

```
    '''
    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):
        for i in np.unique(trainlabel):  # 对所有的可能分类结果循环
            self.Pc[i] = (trainlabel[np.where(trainlabel == i)].shape[0] +
1)/(trainlabel.shape[0] + np.unique(trainlabel).shape[0])
            tmp = traindata[np.where(trainlabel == i),:][0]  # 取出数据集中对应分类
的所有组
            for j in range(traindata.shape[1]):  # 对所有的属性循环
                if featuretype[j] == 0:  # 属性为离散的情况
                    for k in np.unique(traindata[:,j]):  # 对离散属性的每一种取值可能
循环
                        self.Pxc[(i, j, k)] = (tmp[np.where(tmp[:,0] ==
k)].shape[0] + 1)/(tmp.shape[0] + np.unique(traindata[:,0]).shape[0])
```

```
                else:  # 属性为连续的情况
                    self.Pxc[(i, j)] = (np.average(tmp.T[j]),
np.sqrt(np.var(tmp.T[j])))


    '''
    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):
        prediction = []
        test_num = features.shape[0]
        for k in range(test_num):
            probabilities = []  # 存放概率的集合
            for c in range(1, 4):  # 对所有分类可能循环
                temp = math.log(self.Pc[c])  # 利用temp的值累乘计算特定分类的相对概率值
                for i in range(features.shape[1]):
                    if featuretype[i] == 0:  # 属性为离散的情况
                        temp += math.log(self.Pxc[(c, i, int(features[k][i]))])
                    else:  # 属性为连续的情况
                        (mean, sigma) = self.Pxc[(c, i)]  # 取出高斯分布的两个参数
                        temp += math.log(math.exp(-((features[k][i] - mean) ** 2
/ 2) / (sigma ** 2)) /(((2 * math.pi) ** 0.5) * sigma))
                probabilities.append(temp)
            prediction.append(np.argmax(probabilities) + 1)  # 求出最大概率对应的分
类
        return np.array(prediction).reshape(test_num, 1)
```

实验结果：

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6134282807731435
0.7137404580152672
0.4725111441307578
0.6684005201560468
macro-F1: 0.6182173741006906
micro-F1: 0.6134282807731435
```

可见测试通过

实现SVM分类器

思路： SVM 的目标是要找到一个超平面实现最大间距的分类。形式化后为

$$
\begin{aligned}
&min_{\omega,b} \quad \frac{1}{2}||\omega||^2 \\
&s.t.\ y_i(\omega^T x_i + b) \geq 1 \quad ,i = 1,2,\ldots,m
\end{aligned}
$$

这是一个凸二次规划问题，考虑使用拉格朗日乘子法，得到其对偶问题

$$
\begin{aligned}
&\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \left(\mathbf{x}_i^{\top} \mathbf{x}_j\right) \\
&s.t. \ \sum_{i=1}^{n} \alpha_i y_i = 0 \\
&\qquad \alpha \geq 0 \quad ,i = 1,2,\ldots,n
\end{aligned}
$$

考虑到松弛条件，修正为

$$\begin{aligned}
&\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \left(\mathbf{x}_i^{\top} \mathbf{x}_j\right) \\
&s.t. \ \sum_{i=1}^{n} \alpha_i y_i = 0 \\
&\quad 0 \leq \alpha_i \leq C \quad , i = 1, 2, \ldots, n
\end{aligned}$$

对应到本问题，结合核函数转化即为

$$\begin{aligned}
&\min_{\alpha} \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j k\left(\mathbf{x}_i, \mathbf{x}_j\right) - \sum_{i=1}^{n} \alpha_i \\
&s.t. \ \sum_{i=1}^{n} \alpha_i y_i = 0 \\
&\quad 0 \leq \alpha_i \leq C \quad , i = 1, 2, \ldots, n
\end{aligned}$$

可以使用标准的二次规划进行求解

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

对应过来即为

$$\begin{aligned}
&b = 0 \\
&x = (\alpha_1, \alpha_2, \ldots, \alpha_n)^T \\
&q = (-1, -1, \ldots, -1) \\
&h = (C, C, \ldots, C, 0, 0, \ldots, 0)^T \\
&A = (y_1, y_2, \ldots, y_n) \\
&P = [y_i y_j k(x_i, x_j)]_{i \times j} \\
&G = \begin{bmatrix}
1 & & & & \\
& 1 & & & \\
& & \ddots & & \\
& & & 1 & \\
-1 & & & & \\
& -1 & & & \\
& & \ddots & & \\
& & & -1 &
\end{bmatrix}_{2N \times N}
\end{aligned}$$

实际代码实现： 按照上述思路，直接构造各参数调库实现求解即可

实验结果： 分别对三种核函数运行测试。为了显示方便，图中显示的是cvxopt关闭显示过程后输出的结果。而打开显示后可以观察到迭代次数。首先测试线性核（实测迭代15次）

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6581892166836215
0.7678571428571428
0.568733153638814
0.6804123711340206
macro-F1: 0.6723342225433259
micro-F1: 0.6581892166836215
```

随后测试多项式和（实测迭代20次）

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6449643947100712
0.750551876379691
0.5717948717948718
0.6575716234652115
macro-F1: 0.6599727905465914
micro-F1: 0.6449643947100712
```

最后测试高斯核（实测迭代21次）

```
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6561546286876907
0.755056179775281
0.570673712021136
0.6832460732984293
macro-F1: 0.6696586550316154
micro-F1: 0.6561546286876907
```

迭代次数高斯核 > 多项式核 > 线性核，本次训练的精度上 线性核 > 高斯核 > 多项式核

深度学习部分

手写感知机模型并进行反向传播

思路： 模仿 Pytorch 的写法，为模型定义一个类，__init__函数定义所有结构，train 函数内完成训练操作。

训练操作分三步：前向计算结果 ⇒ 反向传播得到梯度 ⇒ 更新网络。与调库不同，计算梯度的操作根据链式求导法则经过后向传播由自己手动书写。

代码实现： 根据感知机模型按部就班书写即可，此处为模型结构

```
def __init__(self, lr=0.05, epochs=200):
self.LLayer1_omega = torch.normal(0, 0.1, (5, 4), requires_grad=True)  # 第一个线
性层
self.LLayer1_b = torch.normal(0, 0.1, (1, 4), requires_grad=True)
self.LLayer2_omega = torch.normal(0, 0.1, (4, 4), requires_grad=True)  # 第二个线
性层
self.LLayer2_b = torch.normal(0, 0.1, (1, 4), requires_grad=True)
self.LLayer3_omega = torch.normal(0, 0.1, (4, 3), requires_grad=True)  # 第三个线
性层
self.LLayer3_b = torch.normal(0, 0.1, (1, 3), requires_grad=True)
self.Sigmoid1 = None  # 第一个激活函数
self.Sigmoid2 = None  # 第二个激活函数
self.lr = lr
self.epochs = epochs
```

前向部分

```
# 前向计算结果
X = torch.tensor([train_data[i].tolist()], requires_grad=True)  # 源数据
out1 = torch.matmul(X, self.LLayer1_omega) + self.LLayer1_b  # 经过第一层
self.Sigmoid1 = 1 / (1 + torch.exp(-out1))  # 经过第一个激活函数
out2 = torch.matmul(self.Sigmoid1, self.LLayer2_omega) + self.LLayer2_b  # 经过第
二层
self.Sigmoid2 = 1 / (1 + torch.exp(-out2))  # 经过第二个激活函数
out3 = torch.matmul(self.Sigmoid2, self.LLayer3_omega) + self.LLayer3_b  # 经过第
三层
y = torch.exp(out3) / torch.sum(torch.exp(out3))
```

反向传播

```
# 反向计算梯度
loss = -torch.log(y[0][train_label[i]-1])
loss_sum.append(float(loss))
loss.backward()
grad = y
grad[0][train_label[i]-1] -= 1
temp3_grad_omega = torch.matmul(self.Sigmoid2.T, grad)  # 反向传播第三层
temp3_grad_b = torch.matmul(grad.T, torch.ones((self.Sigmoid2.shape[0], 1),
requires_grad=True)).T
grad = torch.matmul(grad, self.LLayer3_omega.T)
grad = grad * self.Sigmoid2 * (1 - self.Sigmoid2)  # 反向传播第二个激活函数
temp2_grad_omega = torch.matmul(self.Sigmoid1.T, grad)  # 反向传播第二层
temp2_grad_b = torch.matmul(grad.T, torch.ones((self.Sigmoid1.shape[0], 1),
requires_grad=True)).T
grad = torch.matmul(grad, self.LLayer2_omega.T)
grad = grad * self.Sigmoid1 * (1 - self.Sigmoid1)  # 反向传播第一个激活函数
temp1_grad_omega = torch.matmul(X.T, grad)
temp1_grad_b = torch.matmul(grad.T, torch.ones((X.shape[0], 1),
requires_grad=True)).T
grad = torch.matmul(grad, self.LLayer1_omega.T)  # 反向传播第一层
```

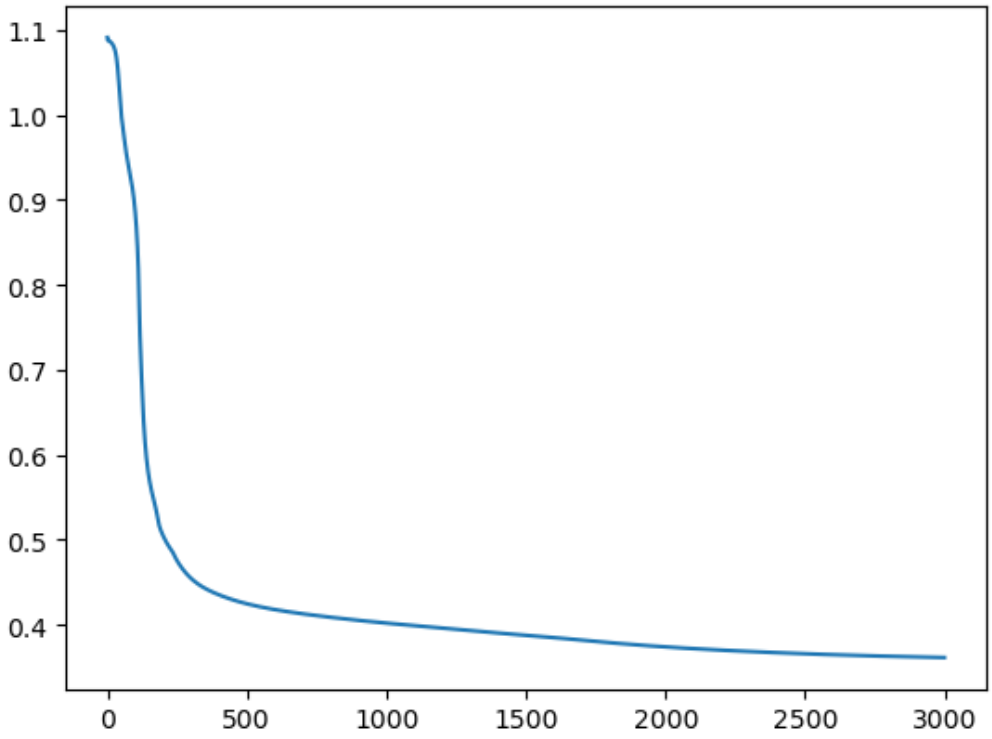
更新

```
# 更新网络
self.LLayer1_omega = torch.tensor((self.LLayer1_omega - self.lr *
temp1_grad_omega).tolist(),
requires_grad=True)  # 更新第一层
self.LLayer1_b = torch.tensor((self.LLayer1_b - self.lr *
temp1_grad_b).tolist(), requires_grad=True)
self.LLayer2_omega = torch.tensor((self.LLayer2_omega - self.lr *
temp2_grad_omega).tolist(),
requires_grad=True)  # 更新第二层
self.LLayer2_b = torch.tensor((self.LLayer2_b - self.lr *
temp2_grad_b).tolist(), requires_grad=True)
self.LLayer3_omega = torch.tensor((self.LLayer3_omega - self.lr *
temp3_grad_omega).tolist(),
requires_grad=True)  # 更新第三层
self.LLayer3_b = torch.tensor((self.LLayer3_b - self.lr *
temp3_grad_b).tolist(), requires_grad=True)
```

实验结果： 验证梯度

```
        [ 0.0140,  0.0044, -0.0035, -0.0016]])
我的 b 梯度: tensor([[ 0.0280,  0.0089, -0.0069, -0.0033]], grad_fn=<PermuteBackward>)
自动算的 b 梯度: tensor([[ 0.0280,  0.0089, -0.0069, -0.0033]])
线性层1:
我的 omega 梯度:
 tensor([[-1.1955e-04,  1.7804e-03,  3.4780e-04, -1.6416e-03],
        [ 2.1209e-04, -3.1585e-03, -6.1701e-04,  2.9123e-03],
        [ 8.9653e-05, -1.3351e-03, -2.6082e-04,  1.2311e-03],
        [-2.1086e-04,  3.1402e-03,  6.1344e-04, -2.8954e-03],
        [-1.2695e-04,  1.8906e-03,  3.6933e-04, -1.7432e-03]],
       grad_fn=<MmBackward>)
自动算的 omega 梯度:
 tensor([[-1.1955e-04,  1.7804e-03,  3.4780e-04, -1.6416e-03],
        [ 2.1209e-04, -3.1585e-03, -6.1701e-04,  2.9123e-03],
        [ 8.9653e-05, -1.3351e-03, -2.6082e-04,  1.2311e-03],
        [-2.1086e-04,  3.1402e-03,  6.1344e-04, -2.8954e-03],
        [-1.2695e-04,  1.8906e-03,  3.6933e-04, -1.7432e-03]])
我的 b 梯度: tensor([[-4.5479e-05,  6.7730e-04,  1.3231e-04, -6.2450e-04]],
       grad_fn=<PermuteBackward>)
自动算的 b 梯度: tensor([[-4.5479e-05,  6.7730e-04,  1.3231e-04, -6.2450e-04]])
```

可见梯度计算无误。执行训练并且绘制 loss 图如下

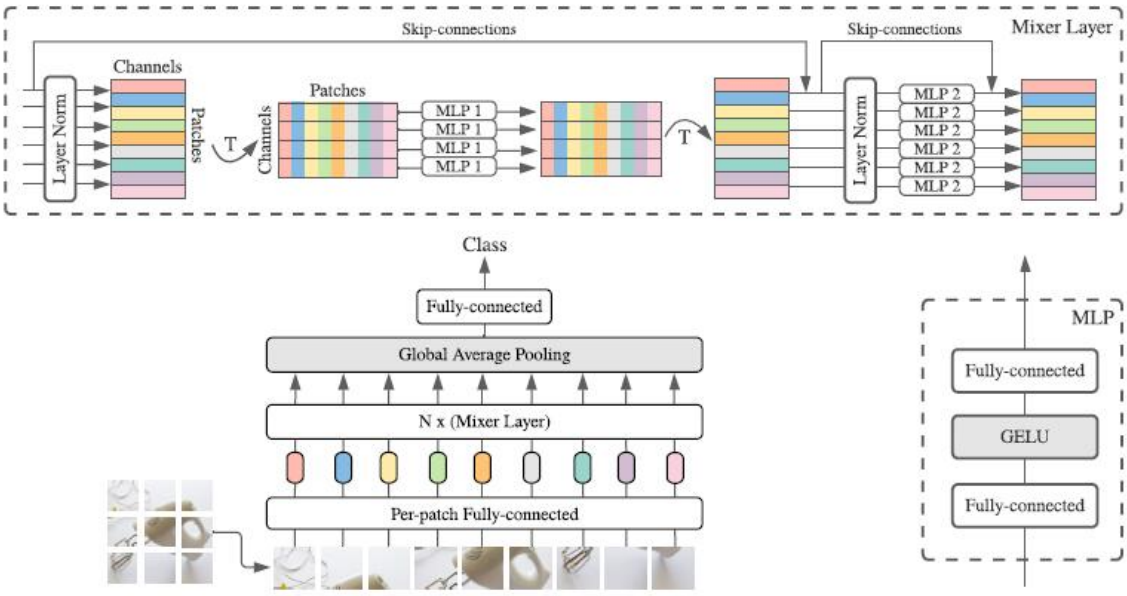


复现MLP-Mixer

思路： MLP-Mixer是一种基于感知机的图像识别模型。它以一系列图像块的线性投影(其形状为`patches x channels`)作为输入。Mixer采用了两种类型的MLP层(注:这两种类型的层交替执行以促进两个维度见的信息交互):

- `channel-mixing` MLP：用于不同通道前通讯，每个token独立处理，即采用每一行作为输入；
- `token-mixing` MLP：用于不同空域位置通讯，每个通道图例处理，即采用每一列作为输入。

Mixer的架构示意图如下



Mixer 先将图像分割成不重叠的等大的 S 个 patch，如果原始输入图像分辨率为 $H \times W$，每个 patch 的分辨率为 $P \times P$，有 $S = HW/P^2$，随后以之为输出，每个 patch 采用相同的投影矩阵进行线性投影，并投影到期望的 hidden_dim C。这将得到一个二维实值输入表$X \in R^{S \times C}$。

Mixer中包含多个等尺寸的层，每个层包含两种MLP模块：

- token-mixing MLP：它作用于 X 的列，从 R^S 映射到 R^S
- Channel-mixing MLP：它作用于 X 的行，从R^C 映射到 R^C

每个MLP模块包含两个全连接层与一个非线性层，它们可以定义如下：

$$U_{*,i} = X_{*,i} + W_2\sigma\left(W_1 \text{ Layer } \text{Norm}(X)_{*,i}\right), \text{ for } i = 1, \cdots, C$$
$$Y_{j,*} = U_{j,*} + W_4\sigma\left(W_3 \text{ Layer } \text{Norm}(U)_{j,*}\right), \text{ for } j = 1, \cdots, S$$

其中 σ 表示GELU激活函数，D_S, D_C 分别表示token-mixing与channel-mixing MLP中隐层宽度。需要注意的是，D_S的选择独立于输入图像块的数量。因此，网络的计算复杂度与输入块的数量成线性关系。而由于 D_C 独立于块尺寸，因此，整体计算量与图像的像素数成线性关系。

除了MLP外，Mixer还采用其他标准架构成分：跳过连接、LayerNorm，并采用了标注分类头，即全局均值池化+线性分类器。

代码实现： 参照代码框架进行填空即可，注意到本次实验的框架里没有给出 channel 数，但是实现的时候又需要这一参量，所以给所有涉及到这一参量的类、方法加了一个变量 channel_n

```
#########################################################################
# 这里需要写Mixer_Layer（layernorm，mlp1，mlp2，skip_connection）
patch_n = (28 // patch_size) ** 2
channel_n = 512
self.layer_norm_1 = nn.LayerNorm(channel_n)
self.layer_norm_2 = nn.LayerNorm(channel_n)
self.mlp_1 = nn.Sequential(
    nn.Linear(patch_n, hidden_dim),
    nn.GELU(),
    nn.Linear(hidden_dim, patch_n))
self.mlp_2 = nn.Sequential(
    nn.Linear(channel_n, hidden_dim),
    nn.GELU(),
    nn.Linear(hidden_dim, channel_n))
#########################################################################
```

构造MLP部分，主要为几个线性映射网络，调库实现即可

```
#########################################################################
forward_data = self.layer_norm_1(x).transpose(1, 2)
forward_data = self.mlp_1(forward_data).transpose(1, 2)
token_mixing = forward_data + x
forward_data = self.layer_norm_2(token_mixing)
forward_data = self.mlp_2(forward_data)
channel_mixing = forward_data + token_mixing
return channel_mixing
#########################################################################
```

同为MLP部分，定义了其前向部分，后向将由 torch 自动处理

```
##########################################################################
# 这里写Pre-patch Fully-connected, Global average pooling, fully connected
channel_n = 512
self.patch_size = patch_size
self.prepatch_fully_connected = nn.Linear(patch_size ** 2, channel_n)
self.mixer_layers = nn.Sequential(*[Mixer_Layer(patch_size, hidden_dim) for _ in
range(depth)])
self.layer_norm = nn.LayerNorm(channel_n)
self.fully_connected = nn.Linear(channel_n, 10)
##########################################################################
```

MLP_Mixer的整体，定义了预处理和最后汇总的部分。

```
##########################################################################
# 注意维度的变化
data = data.reshape(len(data), -1, self.patch_size,
self.patch_size).transpose(1, 2).reshape(len(data), -1, self.patch_size ** 2)
data = self.prepatch_fully_connected(data)
data = self.mixer_layers(data)
data = self.layer_norm(data)
data = torch.mean(data, dim=1)
data = self.fully_connected(data)
return data
##########################################################################
```

首先将图片拆包，随后调用MLP进行训练，最后对每一个 channel，先对所有patch的特征进行汇总，最后各个channel的汇总统一到fully connected来决定分类结果。

```
##########################################################################
# 计算loss并进行优化
optimizer.zero_grad()
result = model(data)
loss = criterion(result, target)
loss.backward()
optimizer.step()
##########################################################################
```

直接使用pytorch提供的优化器进行优化

```
    ##########################################################################
    # 需要计算测试集的loss和accuracy
    if 'num_data' not in vars().keys():
        num_data = 0
    num_data += len(data)
    result = model(data)
    test_loss += criterion(result, target)
    pred = result.argmax(dim=1, keepdim=True)
    num_correct += pred.eq(target.view_as(pred)).sum().item()
test_loss /= num_data
accuracy = 100. * num_correct / num_data
##########################################################################
```

计算loss和accuracy，由于代码框架不允许在外面做修改，所以此处利用 `vars().keys()` 在第一次循环时定义变量 `num_data` 用于统计数据量。

实验结果： 运行程序

```
Train Epoch: 1/5 [0/60000]  Loss: 0.097697
Train Epoch: 1/5 [12800/60000]  Loss: 0.066703
Train Epoch: 1/5 [25600/60000]  Loss: 0.153292
Train Epoch: 1/5 [38400/60000]  Loss: 0.102633
Train Epoch: 1/5 [51200/60000]  Loss: 0.134715
Train Epoch: 2/5 [0/60000]  Loss: 0.102050
Train Epoch: 2/5 [12800/60000]  Loss: 0.043837
Train Epoch: 2/5 [25600/60000]  Loss: 0.134959
Train Epoch: 2/5 [38400/60000]  Loss: 0.043928
Train Epoch: 2/5 [51200/60000]  Loss: 0.053298
Train Epoch: 3/5 [0/60000]  Loss: 0.087126
Train Epoch: 3/5 [12800/60000]  Loss: 0.042480
Train Epoch: 3/5 [25600/60000]  Loss: 0.100434
Train Epoch: 3/5 [38400/60000]  Loss: 0.021793
Train Epoch: 3/5 [51200/60000]  Loss: 0.180304
Train Epoch: 4/5 [0/60000]  Loss: 0.058873
Train Epoch: 4/5 [12800/60000]  Loss: 0.034295
Train Epoch: 4/5 [25600/60000]  Loss: 0.026188
Train Epoch: 4/5 [38400/60000]  Loss: 0.049796
Train Epoch: 4/5 [51200/60000]  Loss: 0.056216
Test set: Average loss: 0.0006   Acc 97.29
```

可见收敛效果和精确度不错。

实验感想和遇到的困难

- **线性分类算法：** 实现按部就班即可。值得注意的是有几种不同的方法。
  - BGD 法实现，在较大数据规模下运算速度较快，但是需要较低的 lr 值来避免其跳过收敛点
  - SGD 法实现，运算较慢，相对于BGD而言实际上执行了远多于它的梯度更新次数，在较高的lr值下有不错的表现
  - 最小二乘法迭代，并非梯度下降算法，在本实验的数据规模下运算速度最快，收敛性能最好，但是随着数据规模的增大，计算伪逆矩阵的开销非常大导致运算很慢
  - 此外，还有随机步长、随着迭代减小步长等方法。

  实现方法的多样带来了不同的可能，权衡彼此之间的性能和使用环境也成为了必要。
- **朴素贝叶斯分类器：** 实现按部就班即可。对于概率的计算涉及到大量的分组聚合，使用 `numpy` 中的 where 或者 `pandas` 会是不错的选择，甚至可以尝试连接 SQL 。
- **SVM分类器：** 根据算法调库实现即可。
- **手写感知机模型：** 按部就班实现即可。本次采用了一个类描述整个模型的方法，事实上写完后分析一番，代码有不少可复用的地方，再考虑到结构清晰，将其细分为几个类之间的嵌套可能会是更好的选择。
- **复现MLP-Mixer：** 主要时间都花在熟悉 pytorch 用法和调研各类资料、博客、仓库上了。由于是调用pytorch官方的优化器，按照其规范书写即可。

总体感觉而言，在课程本身并没有对机器学习做太多详细讲解的情况下，本次实验量有些大，许多的时间花费在学习工具的使用上。而实验的一部分算法每次运行的时间很长（在没有比较合适的机器配置情况下），导致很难做非常详细的测试，考虑到模型本身也很大，因此测试的手段主要还是运行一小部分后对于各个局部变量进行考察，尽可能用理论做出一些比较宽泛的解释。