人工智能lab2

实验目标

- 完成传统机器学习部分和深度学习部分
  - 最小二乘法+L2 规范化项的线性分类器
  - 朴素贝叶斯分类器
  - 支持软间隔和核函数的SVM分类器
  - 手写感知机模型并进行反向传播
  - 复现MLP-Mixer
- 熟悉Pytorch的使用

实验内容

机器学习部分

线性分类算法

完善linearClassification.py的代码以实现线性分类器。

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

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

(2) 基于 (1) 中的结果，实现linearClassification.py中未完成的代码部分。

要求最后在报告中贴上输出的截图。

- 推导过程

$$\begin{gathered}
min_w (Xw-y)^2 + \lambda\|w\|^2 \\
\text{等价于} min_w (Xw-y)^T(Xw-y) + \lambda\|w\|^2 \\
\text{对} w \text{求导，并使得导数值为} 0 \text{，可得} \\
2(Xw-y)^T X + 2\lambda w^T = 0 \\
X^T(Xw-y) + \lambda w = 0 \\
X^T Xw - X^T y + \lambda w = 0 \\
(X^T X + \lambda I)w = X^T y \\
w = (X^T X + \lambda I)^{-1} X^T y
\end{gathered}$$

- 基于以上推导，直接根据传入的训练数据 X、传入的参数 Lambda、传入的训练数据的标签 y，根据上面推导的得到的闭式解计算 w即可

  ```
  w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X) + self.Lambda * I), X.T), y)
  ```

  这里需要注意的是需要在原有的训练数据 X 前加上一列 1，用来代表 bias

  ```
  one_col = np.ones(n)
  X = np.c_[one_col, train_features]
  ```

- 然后基于计算得到的 w 进行预测即可

```
label_pre = np.dot(X, self.w)
```

同样需要加上一列 1

```
one_col = np.ones(n)
X = np.c_[one_col, test_features]
```

注意由于标签是离散化的，而预测得到的值是连续的，所以需要对其离散化，用相应的离散值代表某个连续区间内的所有值

```
for i in label_pre:
    if i[0] <= 1.5:
        result.append(1)
    elif i[0] > 1.5 and i[0] <= 2.5:
        result.append(2)
    else:
        result.append(3)
```

- 特别需要注意要将最终结果装换成(n, 1)大小的向量，否则后续的evaluation会出错

```
np_result = np.array(result).reshape(n, -1)
```

- 结果截图

```
(pytorch2) D:\study\AI\src1>python linearclassification.py
train_num: 3554
test_num: 983
train_feature's shape:(3554, 8)
test_feature's shape:(983, 8)
Acc: 0.6266531027466938
0.6629526462395543
0.6015367727771679
0.6408045977011494
macro-F1: 0.6350980055726239
micro-F1: 0.6266531027466938
```

朴素贝叶斯分类器

- 训练部分
  - 统计每个类别的数据量$|D_c|$以及每个类别中第 i 个属性值为 x 的数据量 $|D_{c,x_i}|$

```
self.Dc = {1: 0, 2: 0, 3:0}
        self.Dcx = {(1,1): 0, (1,2): 0, (1,3): 0, (2,1): 0, (2,2): 0,
(2,3): 0, (3,1): 0, (3,2): 0, (3, 3): 0}

        for data, label in zip(traindata, trainlabel):
            if label[0] == 1:
                self.Dc[1] += 1
                # M
                if data[0] == 1:
                    self.Dcx[(1,1)] += 1
                # F
                elif data[0] == 2:
                    self.Dcx[(1,2)] += 1
```

```
                # I
                elif data[0] == 3:
                    self.Dcx[(1,3)] += 1
            elif label[0] == 2:
                self.Dc[2] += 1
                # M
                if data[0] == 1:
                    self.Dcx[(2,1)] += 1
                # F
                elif data[0] == 2:
                    self.Dcx[(2,2)] += 1
                # I
                elif data[0] == 3:
                    self.Dcx[(2,3)] += 1
            elif label[0] == 3:
                self.Dc[3] += 1
                # M
                if data[0] == 1:
                    self.Dcx[(3,1)] += 1
                # F
                elif data[0] == 2:
                    self.Dcx[(3,2)] += 1
                # I
                elif data[0] == 3:
                    self.Dcx[(3,3)] += 1
```

- 计算每个类别的先验概率$P(c)$

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

```
def cal_prior_P(D_c):
    # （当前类别下的样本数 + 1）/（总样本数 + 类别总数）
    return (len(D_c) + 1) / (n + class_num)
```

- 计算每个类别中 8 个特征维度的平均值和方差

```
# 计算各个类别，8个特征维度的平均值
def cal_X_mean(X):
    X_mean = []
    for i in range(X.shape[1]):
        X_mean.append(np.mean(X[:, i]))
    return X_mean
# 计算各个类别，8个特征维度的方差
def cal_X_var(X):
    X_var = []
    for i in range(X.shape[1]):
        X_var.append(np.var(X[:, i]))
    return X_var
```

- 提取各个类别的数据，字典的键为类别名，值为对应的分类数据

```
data_dict={}
for i in range(1, 4):
    if i in full_data[:,-1]:
        data_dict[i]=full_data[full_data[:, -1]==i]
for i in range(1, 4):
    class_data = data_dict[i]
    X_class = class_data[:, :-1]
    y_class = class_data[:, -1]
    self.Pc[i] = cal_prior_P(y_class)
    self.X_mean[i] = cal_X_mean(X_class)
    self.X_var[i] = cal_X_var(X_class)
```

- 预测部分
    - 假设连续变量服从高斯分布，使用训练数据估计分布的参数，即用训练数据估计对应于每个类的均值和方差

```
def cal_Gaussian(X, XMean, XVar):
    prob = []
    d = len(X)
    for i in range(d):
        a = 1 / np.sqrt(2 * np.pi * XVar[i+1])
        b = np.exp(-(X[i] - XMean[i+1]) ** 2 / (2 * XVar[i+1]))
        prob.append(a * b)
    return prob
```

    - 计算第一个属性（唯一的离散属性）对应的后验概率$\hat{P}(x_i|c)$（使用拉普拉斯平滑条件）

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

```
def cal_discrete_P(X, i):
    return (self.Dcx[i, X] + 1) / (self.Dc[i] + 3)
```

    - 预测

```
res = []
# 对每一个样本进行如下计算
for i in range(features.shape[0]):
    X = features[i, :]
    post_P = []
    # 计算该样本输入每个类别的后验概率
    for j in range(1, 4):
        gaussian_res = cal_Gaussian(X[1:], self.X_mean[j],
self.X_var[j])
        discrete_res = cal_discrete_P(X[0], j)
        post_P.append(np.log(self.Pc[j]) + sum(np.log(gaussian_res)) +
np.log(discrete_res))
        max_index = np.argmax(post_P)
    res.append([max_index + 1])
```

其中判定准则

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

对应代码中的

```
post_P.append(np.log(self.Pc[j]) + sum(np.log(gaussian_res)) +
np.log(discrete_res))
max_index = np.argmax(post_P)
```

这里使用了对原式求了对数

- 结果截图

```
(pytorch2) D:\study\AI\src1>python nBayesClassifier.py
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
```

SVM分类器

- 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。
- 函数SupportVectorMachine.fit()返回值应为svm预测的分数，即 $y = wx + b$
- 核心部分：
  - 随机生成输入

```
X = np.random.randint(5, 10, (100, 5))
X_label = np.random.randint(0, 3, (100, 1))
```

  - 按照 cvxopt 求解问题的要求计算对应的矩阵P、q、G、h、A、b

$$arg\max_{\alpha}[\sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}y_iy_j\alpha_i\alpha_j\langle x_i, x_j\rangle]$$

subject to $0 \leq \alpha_i \leq C$, $\sum_{i=1}^{m}\alpha_i y_i = 0$

```python
n = train_data.shape[0]
test_num = test_data.shape[0]
K = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        K[i][j] = self.KERNEL(train_data[i], train_data[j], self.kernel)
        # P[i][j] = K_i_j * train_label[i] * train_label[j]
P = cvxopt.matrix(np.outer(train_label, train_label) * K)
q = cvxopt.matrix(-1 * np.ones(n))
temp1 = np.diag(-1 * np.ones(n))
temp2 = np.identity(n)
G = cvxopt.matrix(np.vstack((temp1, temp2)))
temp3 = np.zeros(n)
temp4 = np.ones(n) * self.C
h = cvxopt.matrix(np.hstack((temp3, temp4)))
A = cvxopt.matrix(train_label.astype(np.double), (1, n))
b = cvxopt.matrix(0.0)

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

- 得到支持向量对应的 index

```python
alpha = np.array(solution['x'])
sup_idx = np.where(alpha > self.Epsilon)[0]
```

- 计算 bias（为了和前面的 b 区分）

$bias = \frac{1}{|S|}\sum_{s\in S}(y_s - \sum_{i\in S}\alpha_i y_i k(x_i, x_s))$

```python
bias = np.mean([train_label[s] - sum([alpha[i] * train_label[i] *
self.KERNEL(train_data[i], train_data[s], self.kernel) for i in
sup_idx]) for s in sup_idx])
```

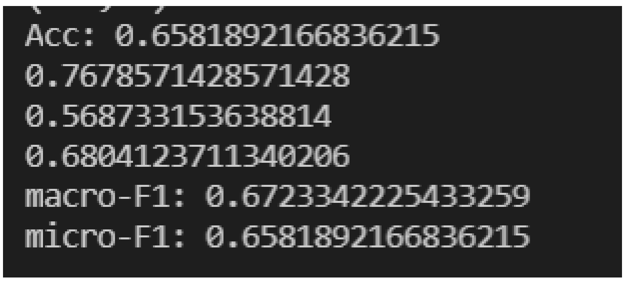
- 计算预测的 y 值

$y = \sum_{i=1}^{m}\alpha_i y_i k(x, x_i) + bias$

```python
res = []
for k in range(test_num):
    pred = sum([alpha[i] * train_label[i] * self.KERNEL(test_data[k],
train_data[i], self.kernel) for i in sup_idx]) + bias
    res.append([pred])

res_ret = np.array(res).reshape(test_num, 1)
```

- 实验结果
  - 线性核



```
Acc: 0.6581892166836215
0.7678571428571428
0.568733153638814
0.6804123711340206
macro-F1: 0.6723342225433259
micro-F1: 0.6581892166836215
```

- 多项式核

```
Acc: 0.6449643947100712
0.7505518763796911
0.5717948717948718
0.6575716234652115
macro-F1: 0.6599727905465914
micro-F1: 0.6449643947100712
```

- 高斯核

```
Acc: 0.6561546286876907
0.7550561797752810
0.5706737120211360
0.6832460732984293
macro-F1: 0.6696586550316154
micro-F1: 0.6561546286876907
```

评价指标

- Accuracy（准确率），即正确预测的样本占所有测试样本的比重。
- F1 score = 2 * P * R / (P + R)，其中准确率 P = TP / (TP + FP)，召回率 R = TP / (TP + FN)。
  - 真正例（True Positive， TP）：真实类别为正例，预测类别为正例。
  - 假正例（False Positive， FP）：真实类别为负例，预测类别为正例。
  - 假负例（False Negative， FN）：真实类别为正例，预测类别为负例。
  - 真负例（True Negative， TN）：真实类别为负例，预测类别为负例
- Macro F1：将 n 分类的评价拆成n 个二分类的评价，计算每个二分类的 F1 score， n 个F1 score 的平均值即为 Macro F1。
- Micro F1：将 n 分类的评价拆成n 个二分类的评价，将 n 个二分类评价的 TP、 FP、 RN对应相加，计算评价准确率和召回率，由这 2 个准确率和召回率计算的 F1 score 即为Micro F1。

深度学习部分

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（隐层神经元设置为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
- 感知机模型
  - 初始化

```
def __init__(self, lr, epochs):
        self.num_layers = 4
        self.w1 = np.random.randn(4, 5)
        self.w2 = np.random.randn(4, 4)
        self.w3 = np.random.randn(3, 4)
        self.lr = lr
        self.epochs = epochs
```

  - sigmoid 函数

```
def sigmoid(self, z):
    return 1.0 / (1.0 + torch.exp(-z))
```

- softmax 函数

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

```python
def softmax(self, X):
    X_exp = X.exp()
    partition = X_exp.sum(axis=0, keepdims=True)
    return X_exp / partition
```

- cross_entropy 函数

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

```python
def cross_entropy(self, y_hat, y):
    return - torch.log(y_hat.gather(1, y.view(-1, 1)))
```

这里使用了 gather 函数，根据 y 的值去索引对应位置的 y_hat

- 前向传播

$$\begin{aligned} \boldsymbol{h}_1 &= s_1(\boldsymbol{W}_1\boldsymbol{x}) \\ \boldsymbol{h}_2 &= s_2(\boldsymbol{W}_2\boldsymbol{h}_1) \\ \hat{\boldsymbol{y}} &= s_3(\boldsymbol{W}_3\boldsymbol{h}_2) \\ L &= \ell(y,\hat{\boldsymbol{y}}) \end{aligned}$$

```python
# Forward Pass
in_data = train_data.t()
z1 = W1.mm(in_data)
h1 = self.sigmoid(z1)
z2 = W2.mm(h1)
h2 = self.sigmoid(z2)
z3 = W3.mm(h2)
y_hat = self.softmax(z3)
```

- 计算损失函数

```python
# Compute Loss
loss = self.cross_entropy(y_hat.t(), train_label)
```

- 反向传播

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

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

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

```
ls3 = torch.rand(3, 100)
for i in range(100):
    for j in range(3):
        if j == train_label[i]:
            ls3[j][i] = y_hat[j][i] - 1
        else:
            ls3[j][i] = y_hat[j][i]

delta3 = ls3 @ h2.t() / n
# (4,3) @ (3,100)
tmp = (W3.t() @ ls3) * h2 * (1 - h2)
# (4, 100) @ (100, 4) = (4, 4)
delta2 = tmp @ h1.t() / n

delta1 = ((W2.t() @ tmp) * h1 * (1 - h1)) @ in_data.t() / n
```

- 梯度下降算法

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

```
W1 = W1 - self.lr * delta1
W2 = W2 - self.lr * delta2
W3 = W3 - self.lr * delta3
```

- 使用Pytorch自动求导

```
    def train_auto(self, X, Y):
        train_data = Variable(torch.from_numpy(X).type(dtype), requires_grad
= False)
        train_label = Variable(torch.from_numpy(Y).type(torch.LongTensor),
requires_grad = False)
        W1 = Variable(torch.from_numpy(self.W1).type(dtype), requires_grad =
True)
```

```
        W2 = Variable(torch.from_numpy(self.W2).type(dtype), requires_grad=
True)
        W3 = Variable(torch.from_numpy(self.W3).type(dtype), requires_grad =
True)

        loss1 = np.zeros([self.epochs,1])
        for epoch in range(self.epochs):
            n = train_data.shape[0]
            # Forward Pass
            in_data = train_data.t()
            z1 = W1.mm(in_data)
            h1 = self.sigmoid(z1)
            z2 = W2.mm(h1)
            h2 = self.sigmoid(z2)
            z3 = W3.mm(h2)
            y_hat = self.softmax(z3)
            # Compute Loss
            loss = self.cross_entropy(y_hat.t(), train_label)

            loss.mean().backward()

            loss1[epoch] = loss.mean().item()
            print(loss1[epoch])

            delta1 = W1.grad.data
            delta2 = W2.grad.data
            delta3 = W3.grad.data

            W1.data = W1.data - self.lr * delta1
            W2.data = W2.data - self.lr * delta2
            W3.data = W3.data - self.lr * delta3

            W1.grad.data.zero_()
            W2.grad.data.zero_()
            W3.grad.data.zero_()

        plt.figure()
        ix = np.arange(self.epochs)
        plt.plot(ix, loss1)
        plt.show()
```

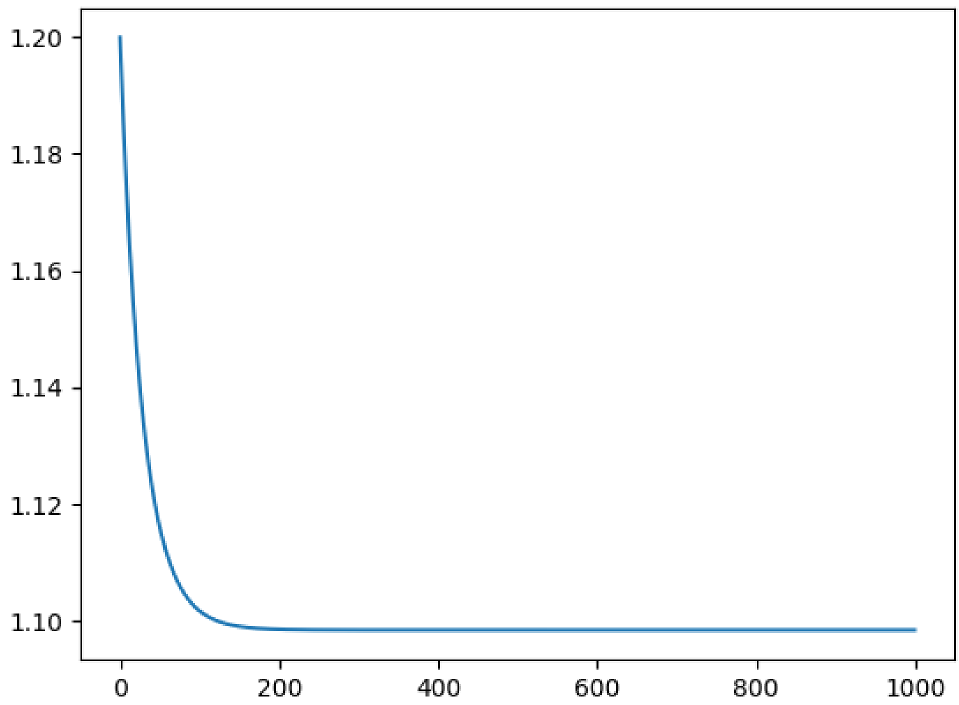
- 执行一轮，对比手动求导和自动求导的结果（从上到下分别是L关于W3、W2、W1的梯度）

```
Manual Grad
tensor([[-0.0493, -0.1399, -0.0374, -0.0366],
        [ 0.0545,  0.1545,  0.0413,  0.0404],
        [-0.0051, -0.0146, -0.0039, -0.0038]])
tensor([[ 8.2686e-02,  6.1613e-10,  8.2687e-02,  3.3822e-08],
        [ 2.0286e-02,  1.7344e-11,  2.0286e-02,  5.5328e-09],
        [ 1.1929e-02,  4.2306e-10,  1.1929e-02,  1.1784e-08],
        [-4.9224e-02,  1.1837e-10, -4.9225e-02, -1.0111e-08]])
tensor([[ 1.7569e-05,  1.1640e-05,  1.7261e-05,  1.0710e-05,  1.1918e-05],
        [-5.4996e-10, -1.4593e-09, -1.0281e-09, -1.5742e-09, -2.3905e-09],
        [-5.2691e-07, -1.1650e-08, -4.7618e-07, -5.2441e-07, -5.6683e-07],
        [-1.7437e-07, -1.0479e-07, -1.1549e-07, -1.3901e-07, -1.5415e-07]])
Auto Grad
tensor([[-0.0493, -0.1399, -0.0374, -0.0366],
        [ 0.0545,  0.1545,  0.0413,  0.0404],
        [-0.0051, -0.0146, -0.0039, -0.0038]])
tensor([[ 8.2686e-02,  6.1613e-10,  8.2687e-02,  3.3822e-08],
        [ 2.0286e-02,  1.7344e-11,  2.0286e-02,  5.5328e-09],
        [ 1.1929e-02,  4.2306e-10,  1.1929e-02,  1.1784e-08],
        [-4.9224e-02,  1.1837e-10, -4.9225e-02, -1.0111e-08]])
tensor([[ 1.7577e-05,  1.1651e-05,  1.7269e-05,  1.0719e-05,  1.1925e-05],
        [-5.4996e-10, -1.4593e-09, -1.0281e-09, -1.5742e-09, -2.3905e-09],
        [-5.3841e-07, -2.3234e-08, -4.8822e-07, -5.3487e-07, -5.8105e-07],
        [-1.7437e-07, -1.0479e-07, -1.1549e-07, -1.3901e-07, -1.5415e-07]])
```

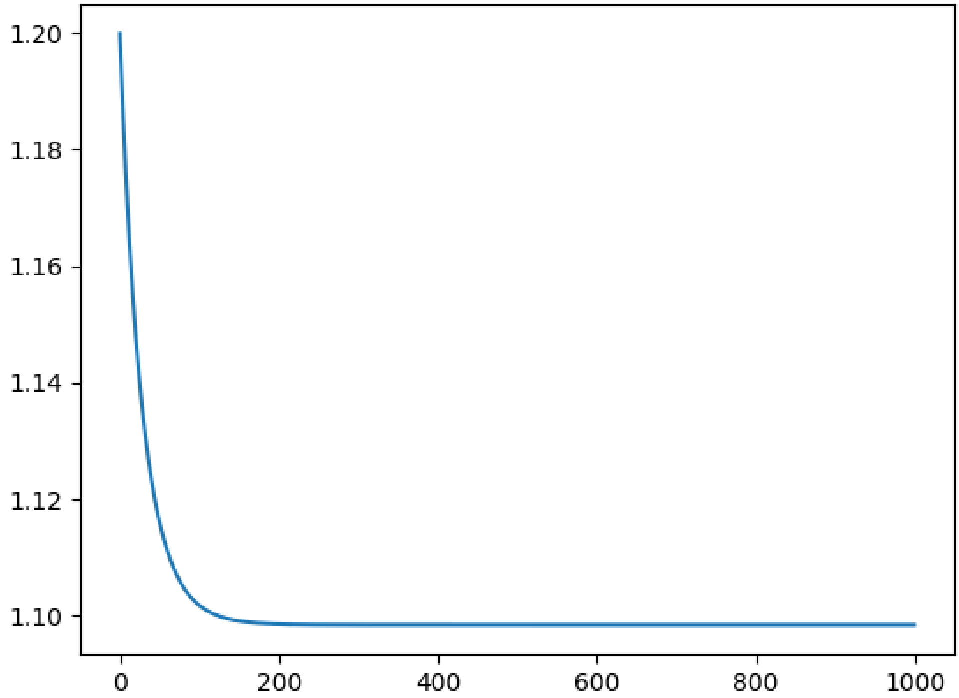
可以看出二者结果相同（关于W1的导数，由于数值太小，存在舍入误差，所以二者稍有区别）

- 迭代 1000 轮，设置学习率为 0.05，对比二者 Loss 的训练曲线

  手动求导



自动求导



可以看出二者均收敛到 1.099左右

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（模型可以自行搜索各种博客，论文） 。参考如下
    - https://arxiv.org/abs/2105.01601
    - https://github.com/d-li14/mlp-mixer.pytorch
    - https://blog.csdn.net/guzhao9901/article/details/116494592
- 数据集介绍：数据集由60000行的训练数据集（trainset）和10000行的测试数据集（testset）组成，包含从0到9的手写数字图片，如下图所示，分辨率为28*28。每一个MNIST数据单元有两部分组成：一张包含手写数字的图片和一个对应的标签（对应代码文件中的data和target）

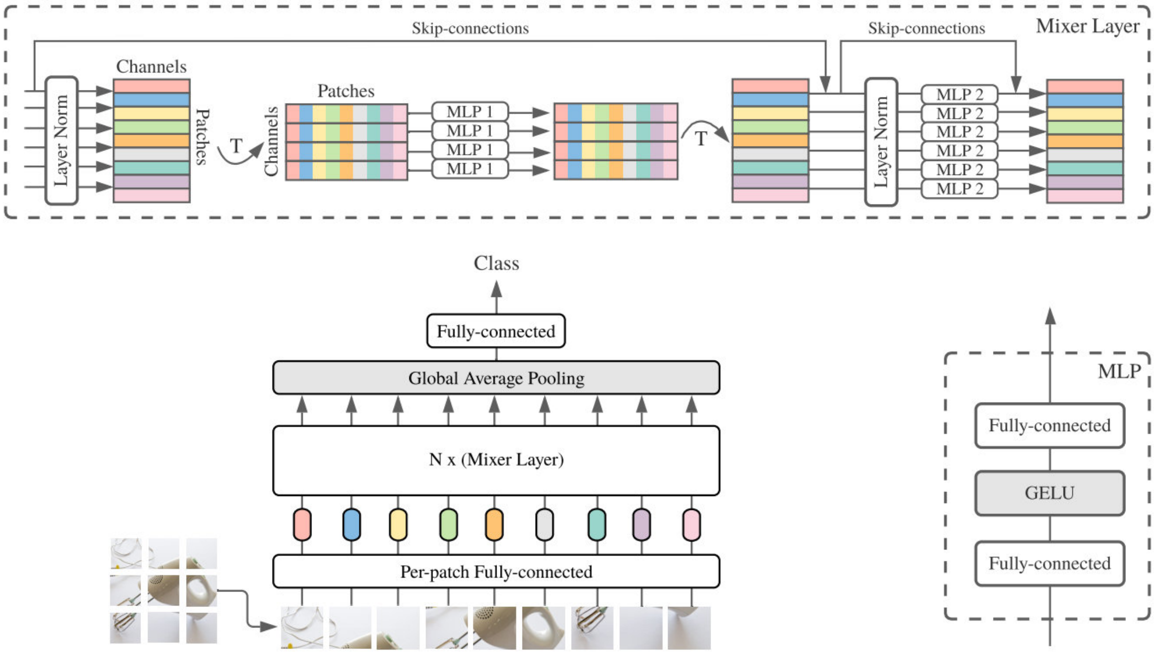


Figure 1: MLP-Mixer consists of per-patch linear embeddings, Mixer layers, and a classifier head. Mixer layers contain one token-mixing MLP and one channel-mixing MLP, each consisting of two fully-connected layers and a GELU nonlinearity. Other components include: skip-connections, dropout, layer norm on the channels, and linear classifier head.

- 模型整体思路：将输入图片拆分成多个不重叠的patches，通过Per-patch Fully-connected层将每个patch转换成feature embedding，送入N个Mixer Layer。最后，经过Global Average Pooling，并使用Fully-connected层进行分类。
- Mixer 架构采用两种不同类型的 MLP 层：token-mixing MLP 和 channel-mixing MLP。
    - token-mixing MLP 允许不同空间位置（token）进行通信，“混合”空间信息（对应图中的MLP1）
    - channel-mixing MLP 允许不同通道（channel）之间进行通信，“混合”每个位置特征（对应图中的MLP2）
- Mixer_Layer：根据论文中给出的模型定义Mixer_Layer。这里自行设定如下超参数

```
        tokens_mlp_dim = 16
        channels_mlp_dim = 128
```

```
class Mixer_Layer(nn.Module):
    def __init__(self, patch_size, hidden_dim):
        super(Mixer_Layer, self).__init__()

 ########################################################################
        #这里需要写Mixer_Layer（layernorm，mlp1，mlp2，skip_connection）
        # 这里自己设定的
        tokens_mlp_dim = 16
        channels_mlp_dim = 128

        class Mlp_Block(nn.Module):
            def __init__(self, hidden_dim, mlp_dim):
                super(Mlp_Block, self).__init__()
                self.mlp = nn.Sequential(
                    nn.Linear(hidden_dim, mlp_dim),
                    nn.GELU(),
                    nn.Linear(mlp_dim, hidden_dim)
```

```
            )

        def forward(self, x):
            return self.mlp(x)

    self.layer_norm_token = nn.LayerNorm(hidden_dim)
    self.token_mix = Mlp_Block(patch_size, tokens_mlp_dim)
    self.layer_norm_channel = nn.LayerNorm(hidden_dim)
    self.channel_mix = Mlp_Block(hidden_dim, channels_mlp_dim)

 ########################################################################

    def forward(self, x):

 ########################################################################
        out = self.layer_norm_token(x).transpose(1, 2)
        x = x + self.token_mix(out).transpose(1, 2)
        out = self.layer_norm_channel(x)
        x = x + self.channel_mix(out)
        return x

 ########################################################################
```

- MLPMixer：根据论文中给出的模型定义MLPMixer

```
class MLPMixer(nn.Module):
    def __init__(self, patch_size, hidden_dim, depth):
        super(MLPMixer, self).__init__()
        assert 28 % patch_size == 0, 'image_size must be divisible by
patch_size'
        assert depth > 1, 'depth must be larger than 1'

 ########################################################################
        #这里写Pre-patch Fully-connected, Global average pooling, fully
connected
        num_classes = 10
        num_tokens = (28 // patch_size) ** 2

        self.patch_emb = nn.Conv2d(1, hidden_dim, kernel_size=patch_size,
stride=patch_size, bias=False)
        self.mlp = nn.Sequential(*[Mixer_Layer(num_tokens, hidden_dim) for _
in range(depth)])
        self.layer_norm = nn.LayerNorm(hidden_dim)
        self.fully_connected = nn.Linear(hidden_dim, num_classes)

 ########################################################################


    def forward(self, data):

 ########################################################################
        #注意维度的变化
        data = self.patch_emb(data)
        data = data.flatten(2).transpose(1, 2)
        data = self.mlp(data)
        data = self.layer_norm(data)
        data = data.mean(dim=1)
```

```
        data = self.fully_connected(data)
        return data

    ########################################################################
```

- 训练模型

```
def train(model, train_loader, optimizer, n_epochs, criterion):
    model.train()
    for epoch in range(n_epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to(device), target.to(device)

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

- 测试

```
def test(model, test_loader, criterion):
    model.eval()
    test_loss = 0.
    num_correct = 0 #correct的个数
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)

    ########################################################################
        #需要计算测试集的loss和accuracy
            pred = model(data)
            loss = criterion(pred, target)
            test_loss += loss
            num_correct += (pred.argmax(dim=1) == target).float().sum()
        test_num = len(test_loader.dataset)
        print(test_num)
        accuracy = num_correct / test_num
        test_loss /= 25 # 5 个 epoch, 每个 epoch 会得到 5 个 loss

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

在循环的每一轮都统计 test_loss 和 num_correct。最后再求平均并输出

- 使用 `(pred.argmax(dim=1) == target).float().sum()` 来判断当前预测正确的测试样本的数量

- 在主函数中定义设备、优化器、criterion（交叉熵）

```
    ########################################################################
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = MLPMixer(patch_size = 7, hidden_dim = 14, depth = 3).to(device)
# 参数自己设定，其中depth必须大于1
    # 这里需要调用optimizer, criterion(交叉熵)
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    criterion = nn.CrossEntropyLoss()
    ########################################################################
```

- 实验结果

```
(pytorch2) D:\study\AI\src2>python MLP_Mixer.py
Train Epoch: 0/5 [0/60000]      Loss: 2.335708
Train Epoch: 0/5 [12800/60000]  Loss: 0.933379
Train Epoch: 0/5 [25600/60000]  Loss: 0.468637
Train Epoch: 0/5 [38400/60000]  Loss: 0.417751
Train Epoch: 0/5 [51200/60000]  Loss: 0.398164
Train Epoch: 1/5 [0/60000]      Loss: 0.221205
Train Epoch: 1/5 [12800/60000]  Loss: 0.129968
Train Epoch: 1/5 [25600/60000]  Loss: 0.243077
Train Epoch: 1/5 [38400/60000]  Loss: 0.317956
Train Epoch: 1/5 [51200/60000]  Loss: 0.207682
Train Epoch: 2/5 [0/60000]      Loss: 0.142281
Train Epoch: 2/5 [12800/60000]  Loss: 0.177089
Train Epoch: 2/5 [25600/60000]  Loss: 0.151986
Train Epoch: 2/5 [38400/60000]  Loss: 0.151511
Train Epoch: 2/5 [51200/60000]  Loss: 0.193290
Train Epoch: 3/5 [0/60000]      Loss: 0.084438
Train Epoch: 3/5 [12800/60000]  Loss: 0.310377
Train Epoch: 3/5 [25600/60000]  Loss: 0.278845
Train Epoch: 3/5 [38400/60000]  Loss: 0.177865
Train Epoch: 3/5 [51200/60000]  Loss: 0.112498
Train Epoch: 4/5 [0/60000]      Loss: 0.122214
Train Epoch: 4/5 [12800/60000]  Loss: 0.143348
Train Epoch: 4/5 [25600/60000]  Loss: 0.104190
Train Epoch: 4/5 [38400/60000]  Loss: 0.106465
Train Epoch: 4/5 [51200/60000]  Loss: 0.127762
10000
Test set: Average loss: 0.4206   Acc 0.96
```

实验总结

通过本次实验，我对机器学习和深度学习有了更加深入的了解。

通过机器学习部分，我对朴素贝叶斯的原理有了更加深入的理解。并学会了如何调用cvxopt求解SVM对应的二次规划问题

最有挑战的就是手动实现感知机模型，其中的求导部分较为复杂。通过对比自动求导和手动求导的结果，我也对Pytorch自动求导有了更加深入的理解。另外通过阅读MLP-Mixer的论文并实现，我也对深度学习前沿有了一定的了解。