使用MLP模型進行乳癌分類

組別: 乙班第11組
組長:鄭翔文 組員:李捷新,鄒博森
2023/01/14

實驗介紹

- 使用pytorch做一個分類器，用來辨識乳癌病患，定義MLP模型，並使用訓練資料來訓練模型，最後使用測試資料來測試模型的準確率。

模型(對照組)

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(X_train.shape[1], 64)#建立全連接層，將輸入的特徵數量映射成64個元素
        self.fc2 = nn.Linear(64, 128)#將64個元素映射成128個元素
        self.fc3 = nn.Linear(128, 64)#將128個元素映射成64個元素
        self.fc4 = nn.Linear(64, 2)#將64個元素映射成2個元素
        self.dropout = nn.Dropout(0.2)#建立Dropout層，每次訓練隨機丟棄20%的神經元

    def forward(self, x):
        x = F.relu(self.fc1(x))#將輸入資料x經過第一層全連接層轉換成64個元素
        x = self.dropout(x)#對第一層全連接層的輸出進行Dropout
        x = F.relu(self.fc2(x))#將經過Dropout的輸出經過第二層全連接層轉換成128個元素
        x = self.dropout(x)#進行Dropout
        x = F.relu(self.fc3(x))# 將經過Dropout的輸出經過第三層全連接層轉換成64個元素
        x = self.fc4(x)#將經過第三層全連接層的輸出經過第四層全連接層轉換成2個元素
        return x
```

訓練模型(對照組)

```python
def train(model, device, train_loader, optimizer, epoch):
    model.train()#將模型設置為訓練模式
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)#將數據和標籤發送到指定的裝置上
        optimizer.zero_grad()#對優化器進行參數更新
        output = model(data)#通過模型進行前向傳播
        loss = F.cross_entropy(output, target.argmax(1))#計算輸出和標籤之間的交叉熵損失
        loss.backward()#計算梯度
        optimizer.step()#更新模型參數
        if batch_idx % 500 == 0:#每500次迭代輸出訓練狀態
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

測試模型(對照組)

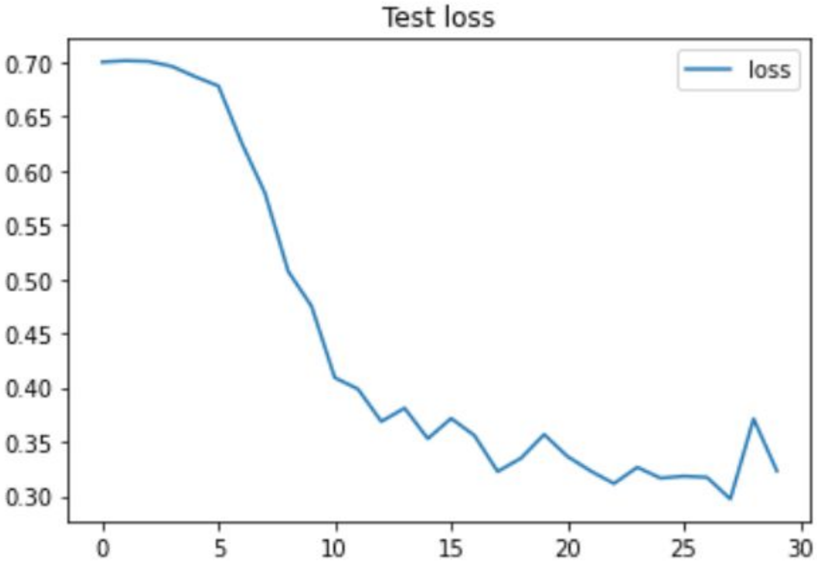
```
def test(model,device,optimizer,epoch,test_loader):
    model.eval()    #將模型設置為驗證模式
    test_loss = 0  #初始化測試損失和正確預測數量
    correct = 0
    with torch.no_grad():    #設置 torch.no_grad()避免計算梯度
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)#將data和target發送到指定的裝置上
            output = model(data)#通過模型進行前向傳播
            test_loss += F.cross_entropy(output, target.argmax(1), reduction='sum').item() #計算輸出和目標之間的交叉損失
            pred = output.argmax(1, keepdim=True)#獲取最高概率預測類的索引
            correct += pred.eq(target.argmax(1, keepdim=True).view_as(pred)).sum().item()#與真實類比較並更新正確預測數

    test_loss /= len(test_loader.dataset)#計算平均測試損失

    print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
    return test_loss, (100. * correct / len(test_loader.dataset))
```

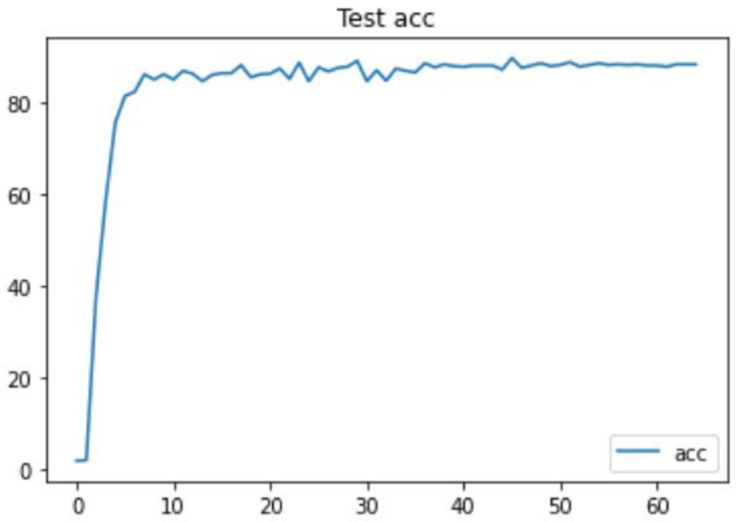
結果輸出(對照組)

```
Train Epoch: 1 [0/12952 (0%)]	Loss: 0.692962
Train Epoch: 1 [4000/12952 (31%)]	Loss: 0.694854
Train Epoch: 1 [8000/12952 (62%)]	Loss: 0.687239
Train Epoch: 1 [12000/12952 (93%)]	Loss: 0.687952
Test set: Average loss: 0.6997, Accuracy: 161/784 (21%)
Train Epoch: 2 [0/12952 (0%)]	Loss: 0.682079
Train Epoch: 2 [4000/12952 (31%)]	Loss: 0.687796
Train Epoch: 2 [8000/12952 (62%)]	Loss: 0.674626
Train Epoch: 2 [12000/12952 (93%)]	Loss: 0.680619
Test set: Average loss: 0.7009, Accuracy: 231/784 (29%)
Train Epoch: 3 [0/12952 (0%)]	Loss: 0.695877
Train Epoch: 3 [4000/12952 (31%)]	Loss: 0.677742
Train Epoch: 3 [8000/12952 (62%)]	Loss: 0.687146
Train Epoch: 3 [12000/12952 (93%)]	Loss: 0.679410
Test set: Average loss: 0.7003, Accuracy: 311/784 (40%)
Train Epoch: 4 [0/12952 (0%)]	Loss: 0.683452
Train Epoch: 4 [4000/12952 (31%)]	Loss: 0.668055
Train Epoch: 4 [8000/12952 (62%)]	Loss: 0.668880
Train Epoch: 4 [12000/12952 (93%)]	Loss: 0.670486
Test set: Average loss: 0.6957, Accuracy: 399/784 (51%)
Train Epoch: 5 [0/12952 (0%)]	Loss: 0.675256
Train Epoch: 5 [4000/12952 (31%)]	Loss: 0.648800
Train Epoch: 5 [8000/12952 (62%)]	Loss: 0.670612
Train Epoch: 5 [12000/12952 (93%)]	Loss: 0.688033
Test set: Average loss: 0.6862, Accuracy: 467/784 (60%)
Train Epoch: 6 [0/12952 (0%)]	Loss: 0.655304
Train Epoch: 6 [4000/12952 (31%)]	Loss: 0.638326
Train Epoch: 6 [8000/12952 (62%)]	Loss: 0.626996
Train Epoch: 6 [12000/12952 (93%)]	Loss: 0.607070
Test set: Average loss: 0.6776, Accuracy: 493/784 (63%)
Train Epoch: 7 [0/12952 (0%)]	Loss: 0.675759
Train Epoch: 7 [4000/12952 (31%)]	Loss: 0.685754
Train Epoch: 7 [8000/12952 (62%)]	Loss: 0.589613
```

```
Train Epoch: 26 [0/12952 (0%)]	Loss: 0.408550
Train Epoch: 26 [4000/12952 (31%)]	Loss: 0.302761
Train Epoch: 26 [8000/12952 (62%)]	Loss: 0.332849
Train Epoch: 26 [12000/12952 (93%)]	Loss: 0.495928
Test set: Average loss: 0.3185, Accuracy: 667/784 (85%)
Train Epoch: 27 [0/12952 (0%)]	Loss: 0.357678
Train Epoch: 27 [4000/12952 (31%)]	Loss: 0.316366
Train Epoch: 27 [8000/12952 (62%)]	Loss: 0.262755
Train Epoch: 27 [12000/12952 (93%)]	Loss: 0.165093
Test set: Average loss: 0.3174, Accuracy: 666/784 (85%)
Train Epoch: 28 [0/12952 (0%)]	Loss: 0.178725
Train Epoch: 28 [4000/12952 (31%)]	Loss: 0.281948
Train Epoch: 28 [8000/12952 (62%)]	Loss: 0.267072
Train Epoch: 28 [12000/12952 (93%)]	Loss: 0.528032
Test set: Average loss: 0.2974, Accuracy: 674/784 (86%)
Train Epoch: 29 [0/12952 (0%)]	Loss: 0.105551
Train Epoch: 29 [4000/12952 (31%)]	Loss: 0.298669
Train Epoch: 29 [8000/12952 (62%)]	Loss: 0.532183
Train Epoch: 29 [12000/12952 (93%)]	Loss: 0.159237
Test set: Average loss: 0.3713, Accuracy: 644/784 (82%)
Train Epoch: 30 [0/12952 (0%)]	Loss: 0.348338
Train Epoch: 30 [4000/12952 (31%)]	Loss: 0.176229
Train Epoch: 30 [8000/12952 (62%)]	Loss: 0.370880
Train Epoch: 30 [12000/12952 (93%)]	Loss: 0.209403
Test set: Average loss: 0.3233, Accuracy: 662/784 (84%)
```

結果輸出(對照組)


Test loss
loss
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0
5
10
15
20
25
30


Test acc
acc
80
60
40
20
0
0
10
20
30
40
50
60

模型(實驗組)

```
#建立MLP模型
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, hidden_size2, hidden_size3, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)  #建立全連接層，輸入大小為input_size，輸出大小為hidden_size，並將其存入fc1中
        self.relu = nn.ReLU()   #建立ReLU激活函數，並將其存入relu 中
        self.fc2 = nn.Linear(hidden_size, hidden_size2)   #輸入大小為hidden_size，輸出大小為hidden_size2，並將其存入fc2中
        self.relu2 = nn.ReLU()    #將其存入relu2中
        self.fc3 = nn.Linear(hidden_size2, hidden_size3)    #輸入大小為hidden_size2，輸出大小為hidden_size3，並將其存入fc3中
        self.relu3 = nn.ReLU()    #將其存入relu3中
        self.fc4 = nn.Linear(hidden_size3,output_size)    #輸入大小為hidden_size3，輸出大小為hidden_size4，並將其存入fc4中

    def forward(self, x):
        x = self.fc1(x)  #使用self.fc1對x進行全連接運算
        x = self.relu(x)  #使用self.relu對上一步的運算結果進行ReLU激活函數運算
        x = self.fc2(x)   #使用self.fc2對上一步的運算結果進行全連接運算
        x = self.relu2(x)   #使用self.relu2對上一步的運算結果進行ReLU激活函數運算
        x = self.fc3(x)   #使用self.fc3對上一步的運算結果進行全連接運算
        x = self.relu3(x)   #使用self.relu3對上一步的運算結果進行ReLU激活函數運算
        x = self.fc4(x)   #使用self.fc4對上一步的運算結果進行全連接運算
        return x
```

訓練模型(實驗組)

```python
#訓練模型
for epoch in range(epoch_t):  #按照指定的epoch進行
    for i in range(num_batches):

        outputs = model(X_batches[i])#處理輸入資料X_batches[i]並產生outputs
        loss = criterion(outputs, Y_batches[i])#通過比較模型的輸出與預期輸出Y_batches[i]來計算損失。


        optimizer.zero_grad()   #清除現有的梯度
        loss.backward()   #計算梯度
        optimizer.step()  #更新參數

    # 每5個epoch輸出一次當前的loss
    if (epoch+1) % 5 == 0:
      print(f'Epoch [{epoch+1}/{epoch_t}], Loss: {loss.item():.3f}')
      test_loss.append(loss.item())
      with torch.no_grad():   #包裹程式塊，避免在測試時計算梯度
        correct = 0
        total = 0
        outputs = model(X2)   #對測試集X2進行前向運算
        _,predicted = torch.max(outputs.data, 1)    #取出outputs中每一行最大值的索引值
        total += Y2.size(0)
        accuracy = 100*(predicted == Y2).sum() / total    #計算出當前模型在測試集上的準確度
        #每5個epoch輸出一次Accuracy
        print(f'Epoch [{epoch+1}/{epoch_t}], Accuracy: {accuracy:.2f} %')
        test_acc.append(accuracy)
```

測試模型(實驗組)

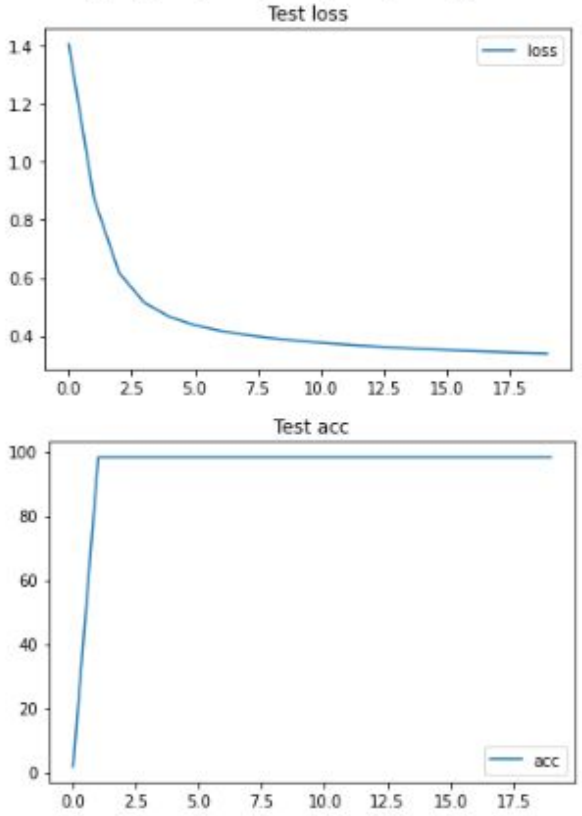
```python
#測試模型
with torch.no_grad():   #禁用自動微分功能，可以減少記憶體使用量
    correct = 0
    total = 0
    outputs = model(X2)   #對測試集 X2 進行前向運算
    _,predicted = torch.max(outputs.data, 1)  #取出outputs中每一行最大值的索引值
    total += Y2.size(0)
    correct += (predicted == Y2).sum()
    print(f'總共:{total}')
    print(f'答對數:{correct}')
    print(f'正確率: {100*correct / total} %')
    conf_matrix = confusion_matrix(Y2, predicted) #計算預測結果與實際結果之間的混淆矩陣
    print('Confusion matrix:')
    print(conf_matrix)
    conf_matrix = torch.zeros((output_size, output_size), dtype=torch.int64)   #將conf_matrix初始化為一個全為0的矩陣，用於存儲預測結果與實際結果之間的混淆矩陣
    prediction = predicted.to(torch.int64)    #將predicted轉換為int64的張量
    for t, p in zip(Y2.view(-1), prediction.view(-1)):  #用zip將Y2和prediction中每一對元素打包成一個tuple
      conf_matrix[t, p] += 1
    print('Confusion matrix:')
    print(conf_matrix)
```

結果輸出(實驗組)

```
Epoch [5/100], Loss: 1.405
Epoch [5/100], Accuracy: 1.79 %
Epoch [10/100], Loss: 0.874
Epoch [10/100], Accuracy: 98.21 %
Epoch [15/100], Loss: 0.617
Epoch [15/100], Accuracy: 98.21 %
Epoch [20/100], Loss: 0.515
Epoch [20/100], Accuracy: 98.21 %
Epoch [25/100], Loss: 0.466
Epoch [25/100], Accuracy: 98.21 %
Epoch [30/100], Loss: 0.437
Epoch [30/100], Accuracy: 98.21 %
Epoch [35/100], Loss: 0.418
Epoch [35/100], Accuracy: 98.21 %
Epoch [40/100], Loss: 0.404
Epoch [40/100], Accuracy: 98.21 %
Epoch [45/100], Loss: 0.393
Epoch [45/100], Accuracy: 98.21 %
Epoch [50/100], Loss: 0.384
Epoch [50/100], Accuracy: 98.21 %
Epoch [55/100], Loss: 0.376
Epoch [55/100], Accuracy: 98.21 %
Epoch [60/100], Loss: 0.370
Epoch [60/100], Accuracy: 98.21 %
Epoch [65/100], Loss: 0.365
Epoch [65/100], Accuracy: 98.21 %
Epoch [70/100], Loss: 0.360
Epoch [70/100], Accuracy: 98.21 %
Epoch [75/100], Loss: 0.355
Epoch [75/100], Accuracy: 98.21 %
```

```
Epoch [80/100], Loss: 0.351
Epoch [80/100], Accuracy: 98.21 %
Epoch [85/100], Loss: 0.348
Epoch [85/100], Accuracy: 98.21 %
Epoch [90/100], Loss: 0.345
Epoch [90/100], Accuracy: 98.21 %
Epoch [95/100], Loss: 0.342
Epoch [95/100], Accuracy: 98.21 %
Epoch [100/100], Loss: 0.339
Epoch [100/100], Accuracy: 98.21 %
總共:784
答對數:770
正確率: 98.21428680419922 %
Confusion matrix:
[[770   0]
 [ 14   0]]
Confusion matrix:
tensor([[770,   0,   0,   0,   0,   0],
        [ 14,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0]])
```

結果輸出(實驗組)



Test loss
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Test acc
acc
100
80
60
40
20
0

線性回歸

```
#線性回歸
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
train_data = pd.read_csv('BC_Train.csv').values

n_samples = train_data.shape[0]    #樣本數量
n_attributes = train_data.shape[1]  #特徵數量
for k in range(n_attributes-1):
  col_data = train_data[:,k]  #獲取當前特徵的數據
  min_v = np.min(col_data)    #獲取當前特徵的最小值
  max_v = np.max(col_data)    #獲取當前特徵的最大值
  train_data[:,k] = (col_data-min_v)/(max_v-min_v)     #正規化當前特徵，使值的範圍在0和1之間

x_train = torch.from_numpy(train_data[:,:-1])   #將正規化的數據轉換為Pytorch張量
N = x_train.shape[0]     #樣本數量
x_train = torch.cat([torch.ones(N,1),x_train],dim=1)  #在邏輯回歸模型中添加偏差項
y_train = torch.from_numpy(train_data[:,-1])     #轉換為Pytorch張量

#與上面的訓練資料做相同的處裡方式
test_data = pd.read_csv('BC_Test.csv').values
n_samples2 = test_data.shape[0]
n_attributes2 = test_data.shape[1]
for L in range(n_attributes2-1):
  col_data2 = test_data[:,L]
  min_v2 = np.min(col_data2)
  max_v2 = np.max(col_data2)
  test_data[:,L] = (col_data2-min_v2)/(max_v2-min_v2)

x_test = torch.from_numpy(test_data[:,:-1])
N2 = x_test.shape[0]
x_test = torch.cat([torch.ones(N2,1),x_test],dim=1)
y_test = torch.from_numpy(test_data[:,-1])
```

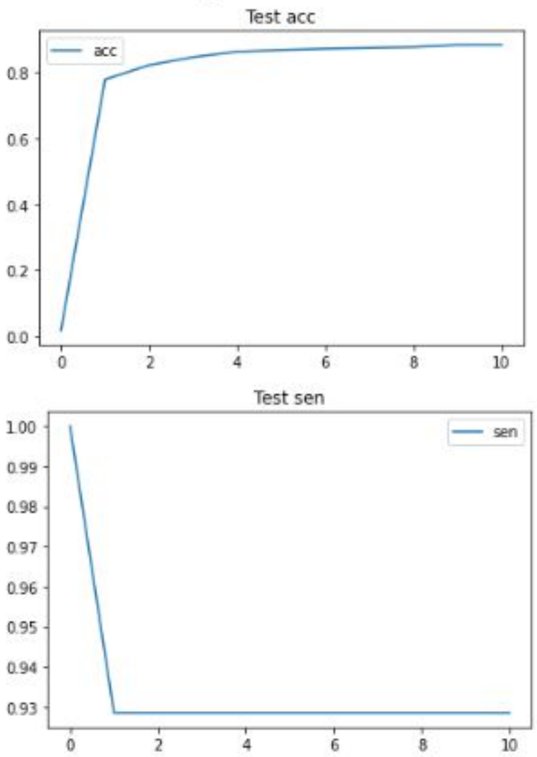
線性回歸

```python
b=0.0000001   #防止在計算損失函數時的除以0錯誤
torch.manual_seed(6)  #保證每次運行程式碼時隨機數生成是一致的
w_pred = torch.rand(n_attributes,requires_grad=True,dtype=torch.float64)    #隨機初始化權重張量
n_iterations = 100001   #訓練次數
lr = 0.1  #學習率
test_acc=[]   #存儲測試集上的準確度
test_sen=[]   #存儲測試集上的敏感度
for it in range(n_iterations):
  w_pred.grad = None    #清空梯度張量
  y_score = torch.sigmoid(torch.mv(x_train,w_pred))   #將訓練數據與權重張量相乘得到預測的類標籤
  loss = -1 * torch.mean(torch.mul(y_train,torch.log(y_score+b))+torch.mul(1-y_train,torch.log(1-y_score+b)))#計算損失
  loss.backward()  #通過反向傳播算法計算梯度
  w_pred.data = w_pred.data - lr*F.normalize(w_pred.grad,dim=0)   #使用學習率來更新權重張量
  if(it%10000==0):
    y_pred = torch.sigmoid(torch.mv(x_test,w_pred))   #將測試數據與更新後的權重張量相乘得到預測的類標籤
    n_cases = y_pred.shape[0]   #樣本數量
    total = 0
    count = 0
    for i in range(n_cases):
      if(y_pred[i]>=0.5): #如果>0.5, 則為陽性
        if(y_test[i]==1):   #為1則表示預測正確
          total += 1    #預測正確的樣本數加1
          count += 1    #預測為陽性的樣本數加1
      else:
        if(y_test[i]==0):   #為0則表示預測正確
          total += 1        #預測正確的樣本數加1
    acc = total/n_cases   #測試集上的準確度
    test_acc.append(acc)
    print(it,': acc =',acc)
    sen = count/torch.sum(y_test).item()    #測試集上的敏感度
    test_sen.append(sen)
    print(it,': sensitivity =',sen)
```

```python
#輸出圖
plt.plot(test_acc)
plt.title('Test acc')
plt.legend(['acc'])
plt.show()
plt.plot(test_sen)
plt.title('Test sen')
plt.legend(['sen'])
plt.show()
```

線性回歸結果輸出

```
0 : acc = 0.017879948914431672
0 : sensitivity = 1.0
10000 : acc = 0.7790549169859514
10000 : sensitivity = 0.9285714285714286
20000 : acc = 0.822477650063857
20000 : sensitivity = 0.9285714285714286
30000 : acc = 0.8467432950191571
30000 : sensitivity = 0.9285714285714286
40000 : acc = 0.8633461047254151
40000 : sensitivity = 0.9285714285714286
50000 : acc = 0.8684546615581098
50000 : sensitivity = 0.9285714285714286
60000 : acc = 0.8722860791826309
60000 : sensitivity = 0.9285714285714286
70000 : acc = 0.8748403575989783
70000 : sensitivity = 0.9285714285714286
80000 : acc = 0.8773946360153256
80000 : sensitivity = 0.9285714285714286
90000 : acc = 0.8837803320561941
90000 : sensitivity = 0.9285714285714286
100000 : acc = 0.8837803320561941
100000 : sensitivity = 0.9285714285714286
```



問題與討論

我們使用邏輯回歸和用多層感知機(MLP)的差別在於**訓練時間**。

邏輯回歸其訓練時間主要取決於樣本數量和特徵數量。假設輸入和輸出之間存在一個線性關係，所以在訓練過程中只需要進行一次最小平方法的運算。因此，邏輯回歸的訓練時間是相對較快的。

然而，多層感知機是一種深度學習算法，訓練時間取決於網路深度和樣本數量。多層感知機可以通過訓練得到非線性的輸入輸出關係，在訓練過程中需要進行多次反向傳播運算。因此，多層感知機的訓練時間是相對較慢的。

問題與討論

Shianwen

他每次算出來的結果差距有點大

不是1.7就是98.12 下午 2:58

已讀 2
下午 3:01
會不會是測資不太夠的問題

Shianwen

FA_DER
會不會是測資不太夠的問題

1萬3千筆 應該不是測資的問題🤣

反正多跑幾次就正常了
當沒看見 下午 3:02

已讀 2
下午 3:09
輸出的格是跟答案對不太上吧

已讀 2
下午 3:13
有加上batchsize嗎

```
總共:784
答對數:14
正確率: 1.7857142686843872 %
```

```
Epoch [80/100], Loss: 0.7083
Epoch [90/100], Loss: 0.7023
Epoch [100/100], Loss: 0.6976
總共:784
答對數:770
正確率: 98.21428680419922 %
```

問題與討論

```python
# 將訓練資料分成多個批次
batch_size = 32

num_batches = len(X) // batch_size#將X的長度除以批次大小來計算批次數
X_batches = X.split(batch_size)#將X按照batchsize分割。
Y_batches = Y.split(batch_size)#將Y按照batchsize分割
```


```python
# Train the model
for epoch in range(epoch_t):#按照指定的epoch進行
    for i in range(num_batches):
        # Forward pass
        outputs = model(X_batches[i])#處理輸入資料X_batches[i]並產生outputs
        loss = criterion(outputs, Y_batches[i])#通過比較模型的輸出與預期輸出Y_batches[i]來計算損失。
```

心得

- 使用GITHUB尋找資料
- 參考ithome文章
- 尋找英文資料
- 線性模型/多層感知機MLP的基礎觀念

參考資料

- ➢ https://youtu.be/c36lUUr864M
- ➢ https://github.com/patrickloeber/pytorchTutorial
- ➢ hibana2077/OOP-independent-study (github.com)
- ➢ https://youtu.be/kQeezFrNoOg
- ➢ 【Day 23】Google ML - Lesson 9 - 加速ML模型訓練的兩大方法(如何設定batch/檢查loss率)、batch size, iteration, epoch的概念和比較)
- ➢ Day-12 Pytorch 介紹 - iT 邦幫忙::一起幫忙解決難題，拯救 IT 人的一天 (ithome.com.tw)
- ➢ 徐位文 Wei-Wen Hsu - 物件導向程式設計 (google.com
- ➢ 《动手学深度学习》— 动手学深度学习 2.0.0 documentation (d2l.ai)

Thank you.
Questions?

Contact me