# 5.2MLP分类模型

# 使用未预处理的数据训练模型

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.manifold import TSNE

import torch

import torch.nn as nn

from torch.optim import SGD, Adam

import torch.utils.data as Data

import matplotlib.pyplot as plt

import seaborn as sns

import hiddenlayer as hl

from torchviz import make\_dot

spam = pd.read\_csv("D:\pythoncode\learn/a\deep Learning\spambase/spambase.csv")

print(spam.head())

# 0 0.64 0.64.1 0.1 0.32 0.2 ... 0.43 0.44 3.756 61 278 1

# 0 0.21 0.28 0.50 0.0 0.14 0.28 ... 0.180 0.048 5.114 101 1028 1

# 1 0.06 0.00 0.71 0.0 1.23 0.19 ... 0.184 0.010 9.821 485 2259 1

# 2 0.00 0.00 0.00 0.0 0.63 0.00 ... 0.000 0.000 3.537 40 191 1

# 3 0.00 0.00 0.00 0.0 0.63 0.00 ... 0.000 0.000 3.537 40 191 1

# 4 0.00 0.00 0.00 0.0 1.85 0.00 ... 0.000 0.000 3.000 15 54 1

#

# [5 rows x 58 columns]

df = pd.DataFrame(spam)

# 获取DataFrame的最后一列

last\_column = df.iloc[:, -1]

# 统计最后一列中值为1和值为0的数量

count\_1 = last\_column.eq(1).sum()

count\_0 = last\_column.eq(0).sum()

print(f"值为1的数量：{count\_1}")

print(f"值为0的数量：{count\_0}")

# 值为1的数量：1812

# 值为0的数量：2788

X = spam.iloc[:, 0:57].values

# y = spam.label.values

y = spam.iloc[:, -1].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.25, random\_state=123

)

scales = MinMaxScaler(feature\_range=(0, 1))

X\_train\_s = scales.fit\_transform(X\_train)

X\_test\_s = scales.transform(X\_test)

colname = spam.columns.values[:-1]

plt.figure(figsize=(20, 14))

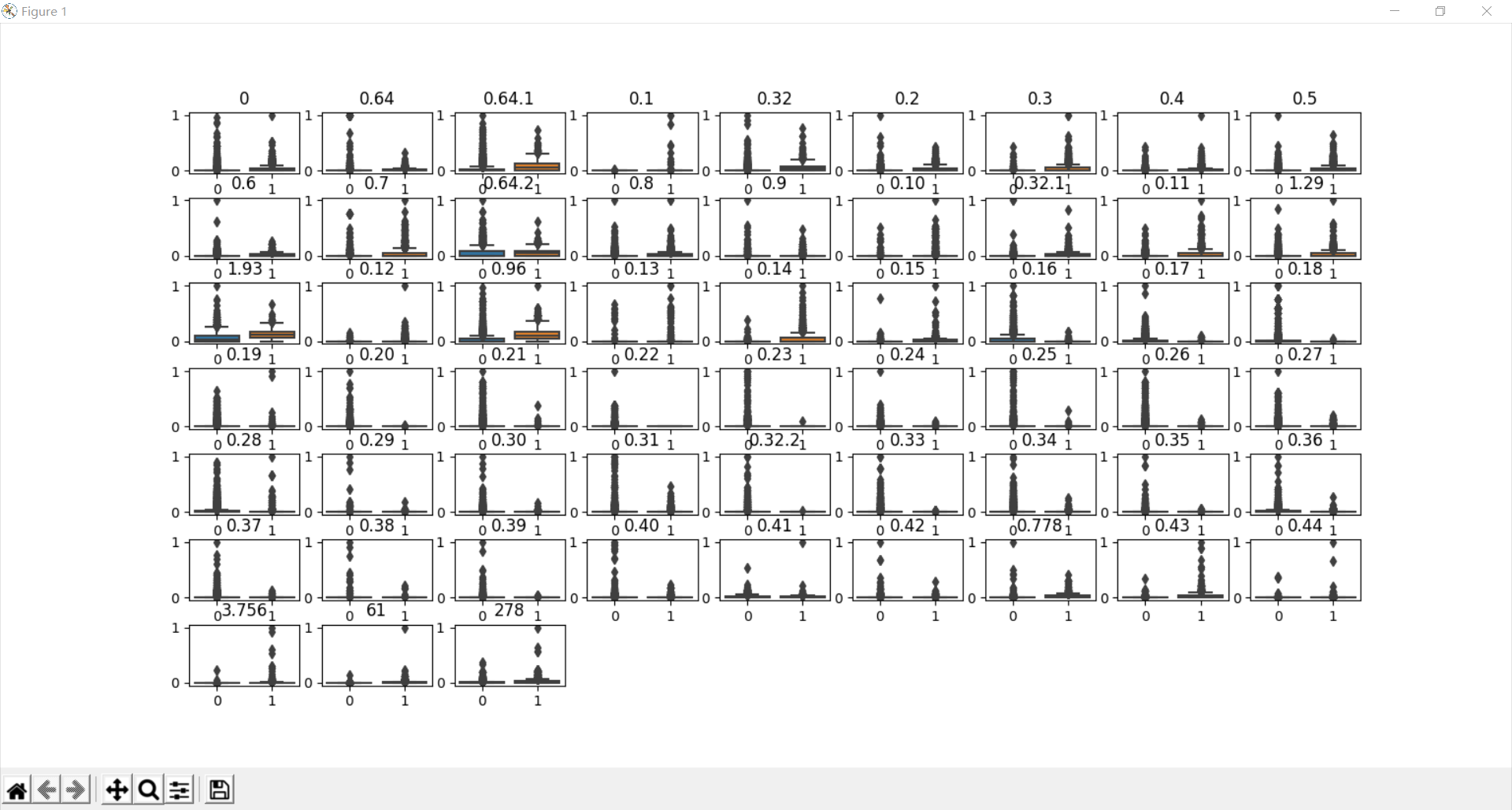
for ii in range(len(colname)):

plt.subplot(7, 9, ii + 1)

sns.boxplot(x=y\_train, y=X\_train\_s[:, ii])

plt.title(colname[ii])

plt.subplots\_adjust(hspace=0.4)

plt.show()

# 全连接网络

class MLPclassifica(nn.Module):

def \_\_init\_\_(self):

super(MLPclassifica, self).\_\_init\_\_()

# 定义第一个隐藏层

self.hidden1 = nn.Sequential(

nn.Linear(

in\_features=57, # 第一个隐藏层的输入，数据的特征数

out\_features=30, # 第一个隐藏层的输出，神经元的数量

bias=True, # 默认会有偏置

),

nn.ReLU()

)

# 定义第二个隐藏层

self.hidden2 = nn.Sequential(

nn.Linear(30, 10),

nn.ReLU()

)

# 分类层

self.classifica = nn.Sequential(

nn.Linear(10, 2),

nn.Sigmoid()

)

# 定义网络的向前传播路径

def forward(self, x):

fc1 = self.hidden1(x)

fc2 = self.hidden2(fc1)

output = self.classifica(fc2)

# 输出为两个隐藏层和输出层的输出

return fc1, fc2, output

mlpc = MLPclassifica()

print(mlpc)

# MLPclassifica(

# (hidden1): Sequential(

# (0): Linear(in\_features=57, out\_features=30, bias=True)

# (1): ReLU()

# )

# (hidden2): Sequential(

# (0): Linear(in\_features=30, out\_features=10, bias=True)

# (1): ReLU()

# )

# (classifica): Sequential(

# (0): Linear(in\_features=10, out\_features=2, bias=True)

# (1): Sigmoid()

# )

# )

x = torch.randn(1, 57).requires\_grad\_(True)

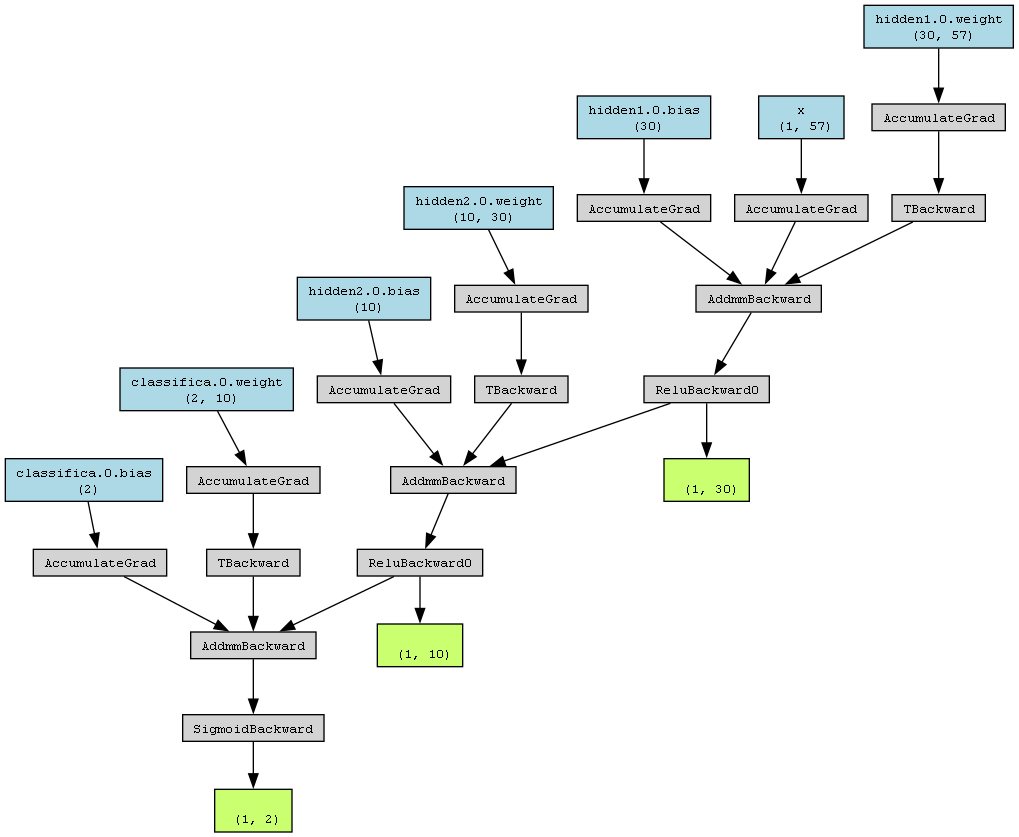
y = mlpc(x)

Mymlpcvis = make\_dot(y, params=dict(list(mlpc.named\_parameters()) + [('x', x)]))

Mymlpcvis.format = "png" # 形式转化为png,默认pdf

# 指定文件保存位置

Mymlpcvis.directory = "D:\pythoncode\learn/a\deep Learning\spambase/"

Mymlpcvis.view() # 会自动生成文件

# 使用未处理的数据训练模型

X\_train\_nots = torch.from\_numpy(X\_train.astype(np.float32)) # 带s是scale过后的，不带s是没有经过标准化处理的

y\_train\_t = torch.from\_numpy(y\_train.astype(np.int64))

X\_test\_nots = torch.from\_numpy(X\_test.astype(np.float32))

y\_test\_t = torch.from\_numpy(y\_test.astype(np.int64))

# 将训练集转化为张量后,使用TensorDataset将X和Y整理到一起

train\_data\_nots = Data.TensorDataset(X\_train\_nots, y\_train\_t)

train\_loader = Data.DataLoader(

dataset=train\_data\_nots,

batch\_size=64,

shuffle=True,

num\_workers=0, # 不使用进程

)

print(len(train\_loader))

# 54

optimizer = torch.optim.Adam(mlpc.parameters(), lr=0.01)

loss\_func = nn.CrossEntropyLoss() # 二分类损失函数

# 记录训练过程的指标

history1 = hl.History()

# 使用Canvas进行可视化

canvas1 = hl.Canvas()

print\_step = 25

# 对模型进行迭代训练,对所有的数据训练EPOCH轮

for epoch in range(15):

## 对训练数据的迭代器进行迭代计算

for step, (b\_x, b\_y) in enumerate(train\_loader):

## 计算每个batch的

\_, \_, output = mlpc(b\_x) # MLP在训练batch上的输出

train\_loss = loss\_func(output, b\_y) # 二分类交叉熵损失函数

optimizer.zero\_grad() # 每个迭代步的梯度初始化为0

train\_loss.backward() # 损失的后向传播，计算梯度

optimizer.step() # 使用梯度进行优化

niter = epoch \* len(train\_loader) + step + 1

## 计算每经过print\_step次迭代后的输出

if niter % print\_step == 0:

\_, \_, output = mlpc(X\_test\_nots)

\_, pre\_lab = torch.max(output, 1)

test\_accuracy = accuracy\_score(y\_test\_t, pre\_lab)

# 为history添加epoch，损失和精度

history1.log(niter, train\_loss=train\_loss,

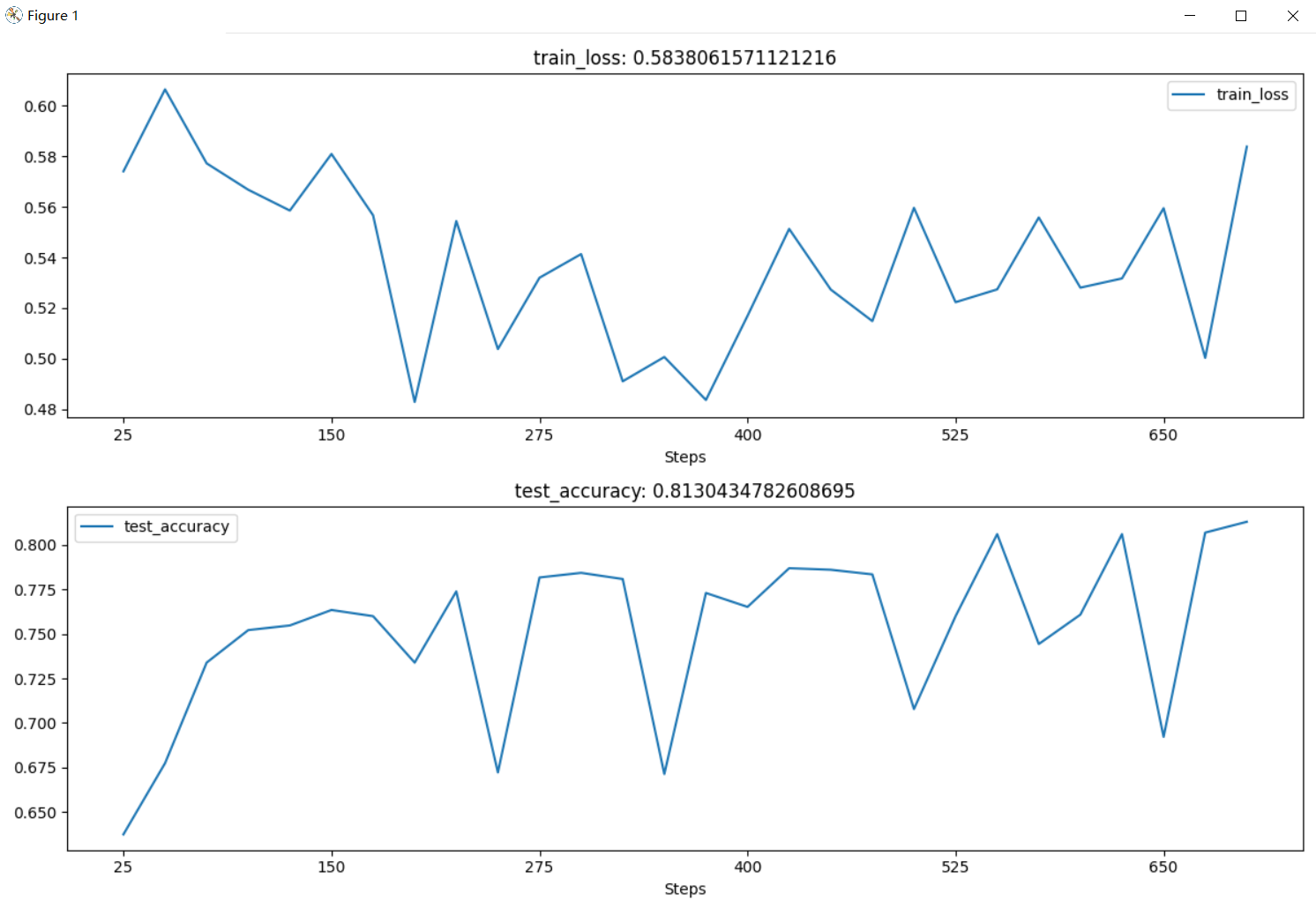
test\_accuracy=test\_accuracy)

# 使用两个图像可视化损失函数和精度

with canvas1:

canvas1.draw\_plot(history1["train\_loss"])

canvas1.draw\_plot(history1["test\_accuracy"])



# 使用预处理后的数据训练模型

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.manifold import TSNE

import torch

import torch.nn as nn

from torch.optim import SGD, Adam

import torch.utils.data as Data

import matplotlib.pyplot as plt

import seaborn as sns

import hiddenlayer as hl

from torchviz import make\_dot

spam = pd.read\_csv("D:\pythoncode\learn/a\deep Learning\spambase/spambase.csv")

X = spam.iloc[:, 0:57].values

# y = spam.label.values

y = spam.iloc[:, -1].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.25, random\_state=123

)

scales = MinMaxScaler(feature\_range=(0, 1))

X\_train\_s = scales.fit\_transform(X\_train)

X\_test\_s = scales.transform(X\_test)

X\_train\_t = torch.from\_numpy(X\_train\_s.astype(np.float32))

y\_train\_t = torch.from\_numpy(y\_train.astype(np.int64))

X\_test\_t = torch.from\_numpy(X\_test\_s.astype(np.float32))

y\_test\_t = torch.from\_numpy(y\_test.astype(np.int64))

# 将训练集转化为张量后,使用TensorDataset将X和Y整理到一起

train\_data = Data.TensorDataset(X\_train\_t, y\_train\_t)

train\_loader = Data.DataLoader(

dataset=train\_data,

batch\_size=64,

shuffle=True,

num\_workers=0, # 不使用进程

)

class MLPclassifica(nn.Module):

def \_\_init\_\_(self):

super(MLPclassifica, self).\_\_init\_\_()

# 定义第一个隐藏层

self.hidden1 = nn.Sequential(

nn.Linear(

in\_features=57, # 第一个隐藏层的输入，数据的特征数

out\_features=30, # 第一个隐藏层的输出，神经元的数量

bias=True, # 默认会有偏置

),

nn.ReLU()

)

# 定义第二个隐藏层

self.hidden2 = nn.Sequential(

nn.Linear(30, 10),

nn.ReLU()

)

# 分类层

self.classifica = nn.Sequential(

nn.Linear(10, 2),

nn.Sigmoid()

)

# 定义网络的向前传播路径

def forward(self, x):

fc1 = self.hidden1(x)

fc2 = self.hidden2(fc1)

output = self.classifica(fc2)

# 输出为两个隐藏层和输出层的输出

return fc1, fc2, output

mlpc = MLPclassifica()

optimizer = torch.optim.Adam(mlpc.parameters(), lr=0.01)

loss\_func = nn.CrossEntropyLoss() # 二分类损失函数

# 记录训练过程的指标

history1 = hl.History()

# 使用Canvas进行可视化

canvas1 = hl.Canvas()

print\_step = 25

## 对模型进行迭代训练,对所有的数据训练EPOCH轮

for epoch in range(15):

## 对训练数据的迭代器进行迭代计算

for step, (b\_x, b\_y) in enumerate(train\_loader):

## 计算每个batch的

\_, \_, output = mlpc(b\_x) # MLP在训练batch上的输出

train\_loss = loss\_func(output, b\_y) # 二分类交叉熵损失函数

optimizer.zero\_grad() # 每个迭代步的梯度初始化为0

train\_loss.backward() # 损失的后向传播，计算梯度

optimizer.step() # 使用梯度进行优化

niter = epoch \* len(train\_loader) + step + 1

## 计算每经过print\_step次迭代后的输出

if niter % print\_step == 0:

\_, \_, output = mlpc(X\_test\_t)

\_, pre\_lab = torch.max(output, 1)

test\_accuracy = accuracy\_score(y\_test\_t, pre\_lab)

# 为history添加epoch，损失和精度

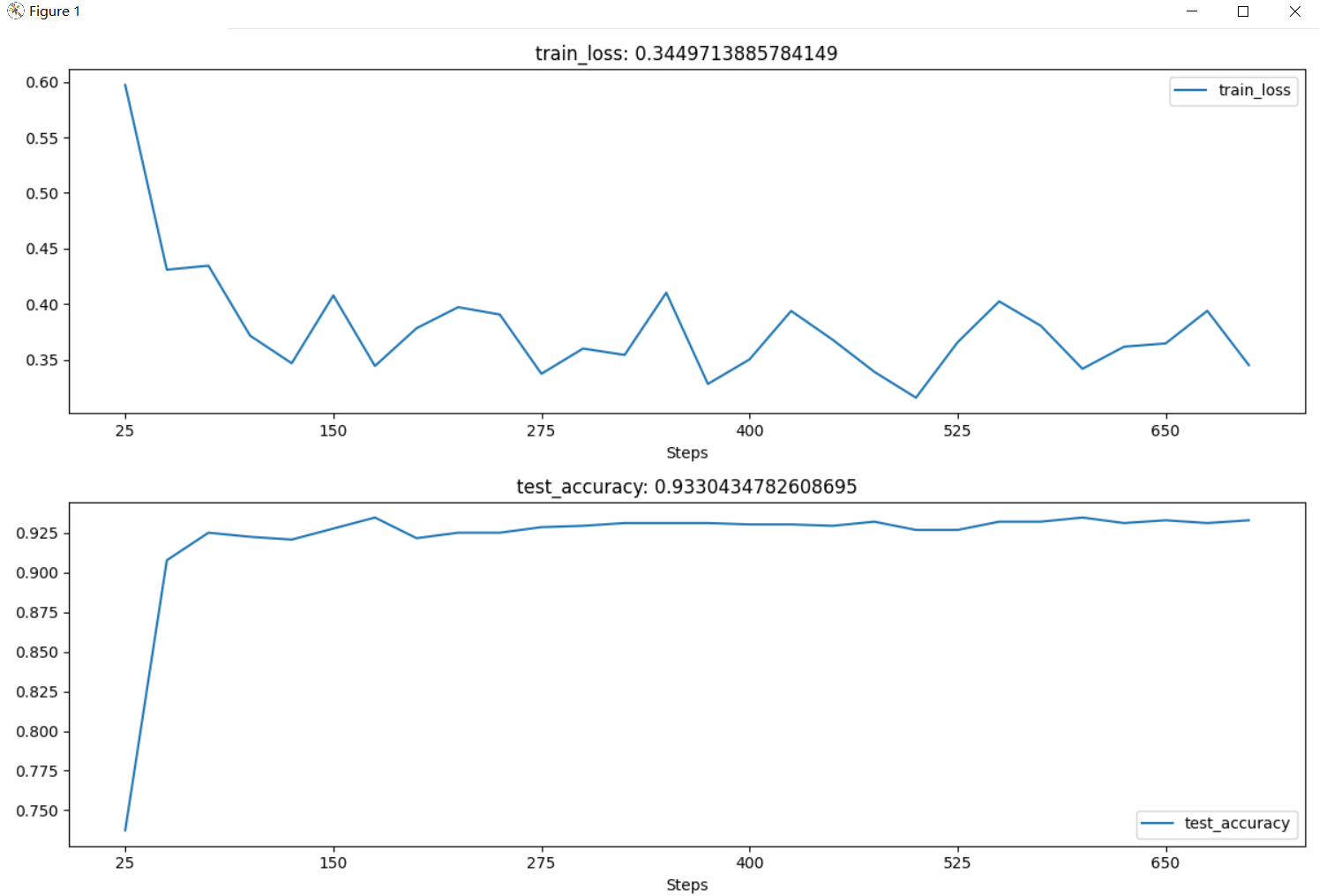
history1.log(niter, train\_loss=train\_loss,

test\_accuracy=test\_accuracy)

# 使用两个图像可视化损失函数和精度

with canvas1:

canvas1.draw\_plot(history1["train\_loss"])

 canvas1.draw\_plot(history1["test\_accuracy"])

\_, \_, output = mlpc(X\_test\_t)

\_, pre\_lab = torch.max(output, 1)

test\_accuracy = accuracy\_score(y\_test\_t, pre\_lab)

print("test\_accuracy:", test\_accuracy)

# test\_accuracy: 0.9347826086956522

# 获取中间层的输出并可视化

\_, test\_fc2, \_ = mlpc(X\_test\_t)

print("test\_fc2.shape:", test\_fc2.shape)

# test\_fc2.shape: torch.Size([1150, 10])

# 使用散点图进行可视化

# 对输出进行降维并可视化

test\_fc2\_tsne = TSNE(n\_components=2).fit\_transform(test\_fc2.data.numpy())

# 将特征进行可视化

plt.figure(figsize=(8, 6))

# 可视化前设置坐标系的取值范围

plt.xlim([min(test\_fc2\_tsne[:, 0] - 1), max(test\_fc2\_tsne[:, 0]) + 1])

plt.ylim([min(test\_fc2\_tsne[:, 1] - 1), max(test\_fc2\_tsne[:, 1]) + 1])

plt.plot(test\_fc2\_tsne[y\_test == 0, 0], test\_fc2\_tsne[y\_test == 0, 1],

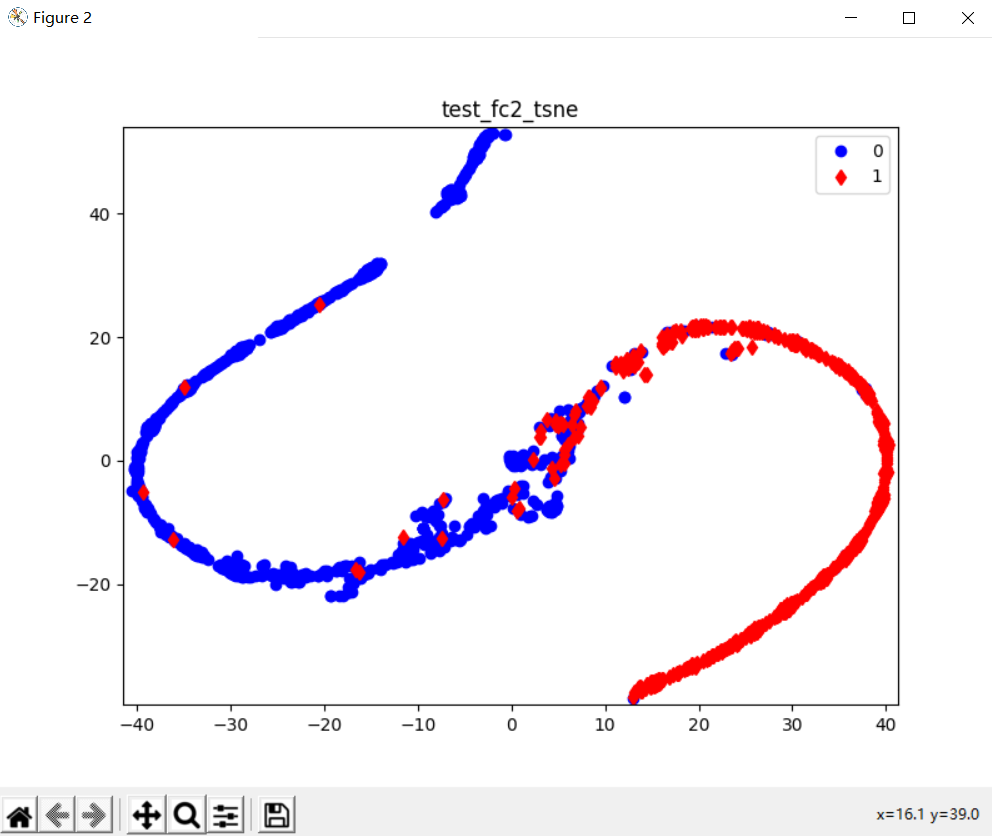
"bo", label="0")

plt.plot(test\_fc2\_tsne[y\_test == 1, 0], test\_fc2\_tsne[y\_test == 1, 1],

"rd", label="1")

plt.legend()

plt.title("test\_fc2\_tsne")

plt.show()

activation = {} # 保存不同层的输出

def get\_activation(name):

def hook(model, input, output):

activation[name] = output.detach()

return hook

mlpc.classifica.register\_forward\_hook(get\_activation("classifica"))

\_, \_, \_ = mlpc(X\_test\_t)

classifica = activation["classifica"].data.numpy()

print("classifica.shape:", classifica.shape)

# classifica.shape: (1150, 2)

plt.figure(figsize=(8, 6))

# 可视化前设置坐标系的取值范围

plt.plot(classifica[y\_test == 0, 0], classifica[y\_test == 0, 1],

"bo", label="0")

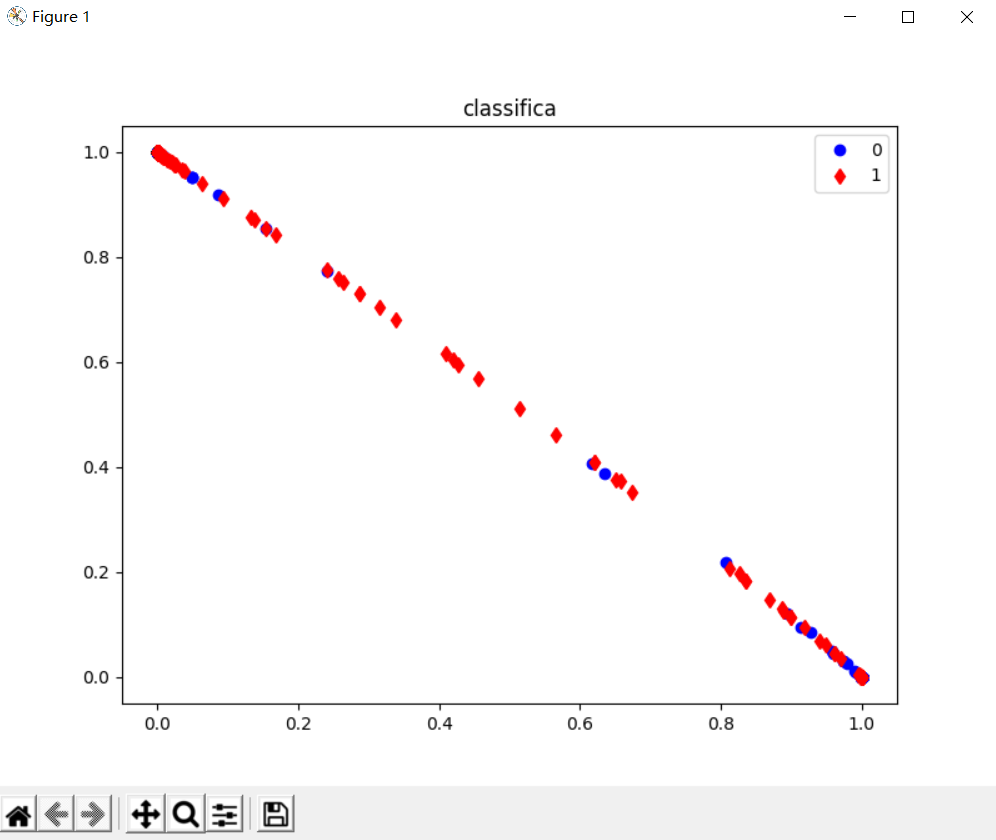
plt.plot(classifica[y\_test == 1, 0], classifica[y\_test == 1, 1],

"rd", label="1")

plt.legend()

plt.title("classifica")

plt.show()



# MLP回归模型

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

from sklearn.datasets import fetch\_california\_housing

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.optim import SGD

import torch.utils.data as Data

import matplotlib.pyplot as plt

import seaborn as sns

import hiddenlayer as hl

housdata = fetch\_california\_housing()

## 数据切分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

housdata.data, housdata.target, test\_size=0.3, random\_state=42)

## 数据标准化处理

scale = StandardScaler()

X\_train\_s = scale.fit\_transform(X\_train)

X\_test\_s = scale.transform(X\_test)

housdatadf = pd.DataFrame(data=X\_train\_s, columns=housdata.feature\_names)

housdatadf["target"] = y\_train

datacor = np.corrcoef(housdatadf.values, rowvar=0)

datacor = pd.DataFrame(data=datacor, columns=housdatadf.columns,

index=housdatadf.columns)

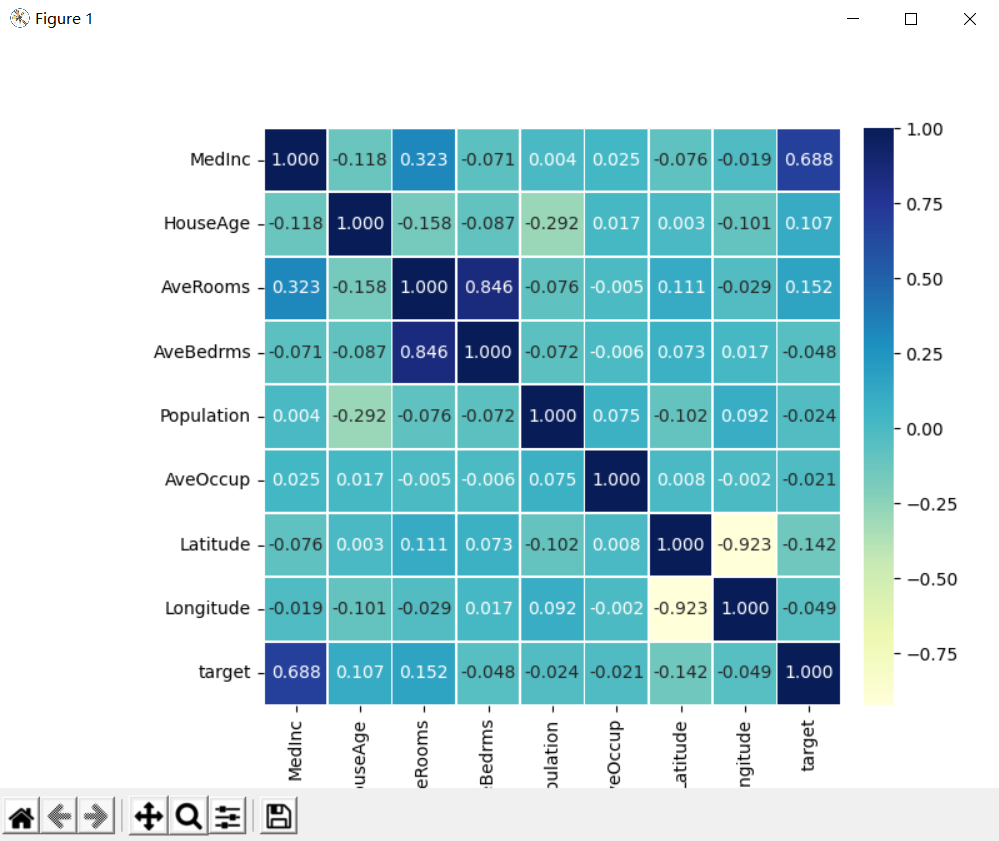
plt.figure(figsize=(8, 6))

ax = sns.heatmap(datacor, square=True, annot=True, fmt=".3f",

linewidths=.5, cmap="YlGnBu",

cbar\_kws={"fraction": 0.046, "pad": 0.03})

plt.show()



train\_xt = torch.from\_numpy(X\_train\_s.astype(np.float32))

train\_yt = torch.from\_numpy(y\_train.astype(np.float32))

test\_xt = torch.from\_numpy(X\_test\_s.astype(np.float32))

test\_yt = torch.from\_numpy(y\_test.astype(np.float32))

# 将训练数据为数据加载器

train\_data = Data.TensorDataset(train\_xt, train\_yt)

test\_data = Data.TensorDataset(test\_xt, test\_yt)

train\_loader = Data.DataLoader(dataset=train\_data, batch\_size=64,

shuffle=True, num\_workers=0)

# 搭建网络预测房价

class MLPregression(nn.Module):

def \_\_init\_\_(self):

super(MLPregression, self).\_\_init\_\_()

## 定义第一个隐藏层

self.hidden1 = nn.Linear(in\_features=8,

out\_features=100, bias=True)

## 定义第二个隐藏层

self.hidden2 = nn.Linear(100, 100)

## 定义第三个隐藏层

self.hidden3 = nn.Linear(100, 50)

## 回归预测层

self.predict = nn.Linear(50, 1)

## 定义网络的向前传播路径

def forward(self, x):

x = F.relu(self.hidden1(x))

x = F.relu(self.hidden2(x))

x = F.relu(self.hidden3(x))

output = self.predict(x)

## 输出一个一维向量

return output[:, 0]

# 输出我们的网络结构

mlpreg = MLPregression()

print(mlpreg)

# MLPregression(

# (hidden1): Linear(in\_features=8, out\_features=100, bias=True)

# (hidden2): Linear(in\_features=100, out\_features=100, bias=True)

# (hidden3): Linear(in\_features=100, out\_features=50, bias=True)

# (predict): Linear(in\_features=50, out\_features=1, bias=True)

# )

optimizer = torch.optim.SGD(mlpreg.parameters(), lr=0.01)

loss\_func = nn.MSELoss() # 均方根误差损失函数

train\_loss\_all = []

# 对模型进行迭代训练,对所有的数据训练EPOCH轮

for epoch in range(30):

train\_loss = 0

train\_num = 0

# 对训练数据的迭代器进行迭代计算

for step, (b\_x, b\_y) in enumerate(train\_loader):

output = mlpreg(b\_x) # MLP在训练batch上的输出

loss = loss\_func(output, b\_y) # 均方根误差损失函数

optimizer.zero\_grad() # 每个迭代步的梯度初始化为0

loss.backward() # 损失的后向传播，计算梯度

optimizer.step() # 使用梯度进行优化

train\_loss += loss.item() \* b\_x.size(0)

train\_num += b\_x.size(0)

train\_loss\_all.append(train\_loss / train\_num)

plt.figure(figsize=(10, 6))

plt.plot(train\_loss\_all, "ro-", label="Train loss")

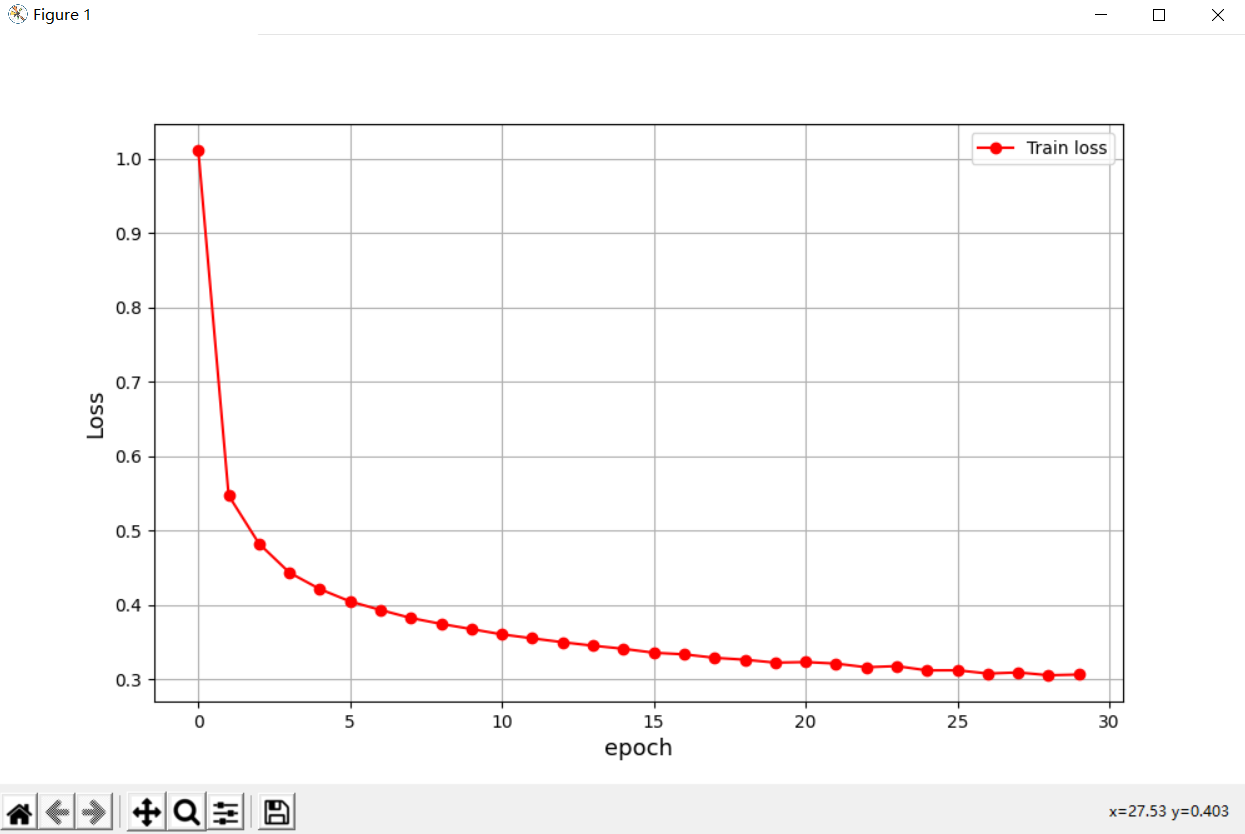
plt.legend()

plt.grid()

plt.xlabel("epoch", size=13)

plt.ylabel("Loss", size=13)

plt.show()



pre\_y = mlpreg(test\_xt)

pre\_y = pre\_y.data.numpy()

mae = mean\_absolute\_error(y\_test, pre\_y)

print("在测试集上的绝对值误差为:", mae)

# 在测试集上的绝对值误差为: 0.38886221073828003

index = np.argsort(y\_test)

plt.figure(figsize=(12, 5))

plt.plot(np.arange(len(y\_test)), y\_test[index], "r", label="Original Y")

plt.scatter(np.arange(len(pre\_y)), pre\_y[index], s=3, c="b", label="Prediction")

plt.legend(loc="upper left")

plt.grid()

plt.xlabel("Index")

plt.ylabel("Y")

plt.show()

