# pytorch 中的优化器

import torch

import torch.nn as nn

class TestNet(nn.Module):

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

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

self.hidden = nn.Sequential(nn.Linear(10, 10), nn.ReLU())

self.regression = nn.Linear(10, 1)

def forward(self, x):

x = self.hidden(x)

output = self.regression(x)

return output

testnet = TestNet()

print(testnet)

# TestNet(

# (hidden): Sequential(

# (0): Linear(in\_features=13, out\_features=10, bias=True)

# (1): ReLU()

# )

# (regression): Linear(in\_features=10, out\_features=1, bias=True)

# )

from torch.optim import Adam

optimizer = Adam(testnet.parameters(), lr=0.001)

# 书本代码示例 ， 没有结合具体场景 ， 下面结合具体场景

# for input, target in dataset:

# optimizer.zero\_grad()

# output = testnet(input)

# loss = nn.MSELoss()(output,target)

# loss.backward()

# optimizer.step()

import torch

import torch.nn as nn

from torch.utils.data import DataLoader, TensorDataset # 假设数据集已经转换为TensorDataset形式

from torch.optim import Adam

import numpy as np

from sklearn.datasets import load\_diabetes

diabetes = load\_diabetes()

X = diabetes.data # 特征数据

Y = diabetes.target # 目标数据

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

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

# 假设这是你的数据集，包含输入(input)和目标(target)的张量对

dataset = TensorDataset(train\_xt, train\_yt)

# 实例化网络、损失函数和优化器

criterion = nn.MSELoss() # 回归使用均方误差损失函数

# 假设已经有了一个数据加载器，用于批量处理数据

dataloader = DataLoader(dataset, batch\_size=32, shuffle=True)

# 训练循环

num\_epochs = 10 # 训练轮数

for epoch in range(num\_epochs):

running\_loss = 0.0

for inputs, targets in dataloader: # 使用DataLoader进行迭代

optimizer.zero\_grad() # 清零梯度

# 前向传播

outputs = testnet(inputs)

targets = targets.unsqueeze(1)

# 计算损失

loss = criterion(outputs, targets)

# 反向传播和优化

loss.backward() # 反向传播计算梯度

optimizer.step() # 更新权重

# 记录损失

running\_loss += loss.item()

# 输出每轮训练的平均损失

print(f'Epoch [{epoch + 1}/{num\_epochs}], Loss: {running\_loss / len(dataloader)}')

print("Training finished.")

# Epoch [1/10], Loss: 29159.409319196428

# Epoch [2/10], Loss: 29114.459123883928

# Epoch [3/10], Loss: 29081.398297991072

# Epoch [4/10], Loss: 29000.051897321428

# Epoch [5/10], Loss: 29023.331612723214

# Epoch [6/10], Loss: 28958.711774553572

# Epoch [7/10], Loss: 28973.793108258928

# Epoch [8/10], Loss: 28933.306222098214

# Epoch [9/10], Loss: 29034.154157366072

# Epoch [10/10], Loss: 29012.052873883928

# Training finished.

# 网络参数初始化

import torch

import torch.nn as nn

conv1 = torch.nn.Conv2d(3, 16, 3)

torch.manual\_seed(12)

torch.nn.init.normal(conv1.weight, mean=0, std=1)

import matplotlib.pyplot as plt

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

plt.hist(conv1.weight.data.numpy().reshape((-1, 1)), bins=30)

plt.show()

print(torch.nn.init.constant(conv1.bias, val=0.1))

# Parameter containing:

# tensor([0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000,

# 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000],

# requires\_grad=True)

class TestNet(nn.Module):

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

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

self.conv1 = nn.Conv2d(3, 16, 3)

# 3: 表示输入通道数。在这个例子中，意味着该卷积层期望接收的输入数据有3个通道，这通常对应于RGB彩色图像，每个通道代表图像的一种颜色信息（红色、绿色、蓝色）。

# 16: 表示输出通道数，也就是卷积操作后产生的特征图（Feature

# Maps）的数量。这意味着该卷积层会通过学习，从输入图像中提取16种不同的特征。

# 3: 指定卷积核（Kernel）的大小，这里是指一个3x3的卷积核。卷积核是在输入数据上滑动的小窗口，用于执行局部区域的加权和操作，从而提取特征。3

# x3的卷积核是比较常用的，因为它可以在保持较高分辨率的同时，捕获局部特征。

self.hidden = nn.Sequential(

nn.Linear(100, 100),

nn.ReLU(),

nn.Linear(100, 50),

nn.ReLU(),

)

self.cla = nn.Linear(50, 10)

self.cla = nn.Linear(50, 10)

# 创建了一个从50维输入映射到10维输出的全连接层，并将其赋值给类或模块的成员变量self.cla。这个全连接层在神经网络模型的尾部经常用来进行分类预测，将提取到的特征转换为类别概率或直接的类别预测。

def forward(self, x):

x = self.conv1(x)

x = x.view(x.shape[0], -1)

x = self.hidden(x)

output = self.cla(x)

return output

# Conv2D和Conv1D是深度学习中两种类型的卷积层，它们在处理不同类型的数据和应用场景上有所区别：

# Conv1D（一维卷积）

# 主要用途：一维卷积主要用于处理序列数据，如时间序列分析、自然语言处理（NLP）中的文本数据或音频信号。这类数据在结构上是一维的，即数据沿单一时间轴或序列轴排列。

# 输入结构：输入数据通常具有形状(batch\_size, channels, sequence\_length)，其中channels对应于特征的数量，sequence\_length是序列的长度。

# 卷积核：一维卷积核也是线性的，形状为(out\_channels, in\_channels, kernel\_size)，其中out\_channels是输出特征图的数量，in\_channels是输入特征的数量，kernel\_size表示卷积核覆盖的序列上的元素数量。

# 应用实例：情感分析、文本分类、语音识别等。

# Conv2D（二维卷积）

# 主要用途：二维卷积主要应用于图像处理、视频帧分析以及其他类型的二维数据。它能够捕获空间结构信息，如图像中的纹理、边缘和形状。

# 输入结构：输入数据的形状通常是(batch\_size, channels, height, width)，其中channels代表颜色通道（如RGB图像中的红、绿、蓝通道），height和width分别代表图像的高度和宽度。

# 卷积核：二维卷积核的形状为(out\_channels, in\_channels, kernel\_height, kernel\_width)，它在图像的宽度和高度上滑动，提取特征。

# 应用实例：图像分类、物体检测、图像分割、视频分析等。

# 主要区别

# 数据维度：最本质的区别在于处理数据的维度。Conv1D处理一维序列数据，而Conv2D处理二维图像或网格数据。

# 卷积核形状：相应地，卷积核的维度也不同，Conv1D的卷积核是一维的，而Conv2D的卷积核是二维的。

# 应用场景：由于数据维度的不同，它们的应用场景也有所侧重。Conv1D适用于处理序列数据相关的任务，而Conv2D则更适合图像处理和相关视觉任务。

# 特征提取：Conv1D更倾向于捕捉序列中的时间或顺序相关特征，而Conv2D则擅长于提取空间结构特征。

# 总的来说，选择Conv1D还是Conv2D取决于数据的性质和任务的需求。在某些特定情况下，通过适当的调整，Conv2D也可以处理看似一维的数据，但通常情况下，针对数据的维度选择合适的卷积类型会更高效和合适。

testnet = TestNet()

print(testnet)

# TestNet(

# (conv1): Conv2d(3, 16, kernel\_size=(3, 3), stride=(1, 1))

# (hidden): Sequential(

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

# (1): ReLU()

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

# (3): ReLU()

# )

# (cla): Linear(in\_features=50, out\_features=10, bias=True)

# )

def init\_weights(m):

if type(m) == nn.Conv2d:

torch.nn.init.normal(m.weight, mean=0, std=0.5)

if type(m) == nn.Linear:

torch.nn.init.uniform(m.weight, a=-0.1, b=0.1)

m.bias.data.fill\_(0.01)

torch.manual\_seed(13)

testnet.apply(init\_weights) # apply 方法，对所有可训练参数应用初始化权重函数

# PYTORCH中定义网络的方式

from sklearn.preprocessing import StandardScaler

import torch

import numpy as np

import torch.utils.data as Data

import pandas as pd

# from sklearn.datasets import load\_boston

import matplotlib.pyplot as plt

import torch.nn as nn

from torch.optim import SGD

# 针对回归，换成另外一个回归数据集

from sklearn.datasets import load\_diabetes

diabetes = load\_diabetes()

X = diabetes.data # 特征数据

Y = diabetes.target # 目标数据

# 查看特征数据和目标数据的形状

print("特征数据形状:", X.shape)

print("目标数据形状:", Y.shape)

# 特征数据形状: (442, 10)

# 目标数据形状: (442,)

plt.figure()

plt.hist(Y, bins=20)

plt.show()

ss = StandardScaler(with\_mean=True, with\_std=True)

diabetes\_Xs = ss.fit\_transform(X)

diabetes\_Ys = Y

train\_xt = torch.from\_numpy(diabetes\_Xs.astype(np.float32))

train\_yt = torch.from\_numpy(diabetes\_Ys.astype(np.float32))

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

train\_loader = Data.DataLoader(

dataset=train\_data,

batch\_size=128,

shuffle=True,

# num\_workers=1 不用多线程

)

class MLPmodel(nn.Module):

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

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

self.hidden1 = nn.Linear(

in\_features=10,

# in\_features=13,

out\_features=10,

bias=True

)

self.active1 = nn.ReLU()

self.hidden2 = nn.Linear(10, 10)

self.active2 = nn.ReLU()

self.regression = nn.Linear(10, 1)

def forward(self, x):

x = self.hidden1(x)

x = self.active1(x)

x = self.hidden2(x)

x = self.active2(x)

output = self.regression(x)

return output

mlp1 = MLPmodel()

print(mlp1)

# MLPmodel(

# (hidden1): Linear(in\_features=13, out\_features=10, bias=True)

# (active1): ReLU()

# (hidden2): Linear(in\_features=10, out\_features=10, bias=True)

# (active2): ReLU()

# (regression): Linear(in\_features=10, out\_features=1, bias=True)

# )

optimizer = SGD(mlp1.parameters(), lr=0.001)

loss\_func = nn.MSELoss()

train\_loss\_all = []

for epoch in range(30):

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

output = mlp1(b\_x).flatten()

train\_loss = loss\_func(output, b\_y)

optimizer.zero\_grad()

train\_loss.backward()

optimizer.step()

train\_loss\_all.append(train\_loss.item())

plt.figure()

plt.plot(train\_loss\_all, 'r-')

plt.title('train loss')

plt.show()

class MLPmodedl2(nn.Module):

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

super(MLPmodedl2, self).\_\_init\_\_()

self.hidden = nn.Sequential(

# nn.Linear(13,10)

nn.Linear(10, 10),

nn.ReLU(),

nn.Linear(10, 10),

nn.ReLU(),

)

self.regression = nn.Linear(10, 1)

def forward(self, x):

x = self.hidden(x)

output = self.regression(x)

return output

mlp2 = MLPmodedl2()

print(mlp2)

optimizer = SGD(mlp2.parameters(), lr=0.001)

loss\_func = nn.MSELoss()

train\_loss\_all = []

for epoch in range(30):

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

output = mlp2(b\_x).flatten()

train\_loss = loss\_func(output, b\_y)

optimizer.zero\_grad()

train\_loss.backward()

optimizer.step()

train\_loss\_all.append(train\_loss.item())

plt.figure()

plt.plot(train\_loss\_all, 'r-')

plt.title('train loss')

plt.show()

#模型的保存

torch.save(mlp2,'./deep Learning/mlp2.pkl')

mlp2load = torch.load('./deep Learning/mlp2.pkl')

print(mlp2load)

# MLPmodedl2(

# (hidden): Sequential(

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

# (1): ReLU()

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

# (3): ReLU()

# )

# (regression): Linear(in\_features=10, out\_features=1, bias=True)

# )

# 只保存模型的参数

torch.save(mlp2.state\_dict(),'./deep Learning/mlp2.pkl')

mlp2param = torch.load('./deep Learning/mlp2.pkl')

print(mlp2param)

# OrderedDict([('hidden.0.weight', tensor([[ -0.9232, -1.2095, -1.7874, -2.4585, -2.9473, -2.6885, 1.8891,

# -3.7561, -3.1867, -3.1027],

# [ -0.2042, -0.0691, -0.4388, -0.3634, -0.3157, -0.4386, 0.1197,

# -0.3744, -0.4468, -0.2533],

# [ -2.7624, -4.0176, -4.5567, -5.7148, -7.7913, -7.2320, 5.5492,

# -10.1651, -8.1563, -7.5583],

# [ 0.0754, -0.1642, -0.2635, -0.1018, -0.1627, -0.2434, 0.0623,

# 0.0514, -0.1754, -0.2588],

# [ -0.7921, -2.4244, -2.5805, -2.9983, -4.5993, -4.1026, 3.2293,

# -5.5808, -4.6215, -4.0749],

# [ -0.9643, -1.4268, -1.9347, -1.8638, -2.4141, -2.5897, 2.0014,

# -3.9987, -2.6508, -2.5059],

# [ -0.3963, -0.2835, -0.4565, -0.8690, -0.8343, -0.4606, 0.6018,

# -0.6620, -0.6662, -0.4508],

# [ -0.1957, -0.0131, -0.1863, -0.3652, -0.1023, -0.3196, 0.1042,

# -0.2538, -0.3476, -0.2444],

# [ 0.1132, 0.0381, -0.1355, 0.0587, -0.1994, 0.2194, 0.0137,

# -0.1489, -0.1327, -0.1354],

# [ -1.8380, -2.7347, -3.7258, -4.0203, -6.0127, -5.8283, 3.2918,

# -7.1877, -5.3882, -4.5944]])), ('hidden.0.bias', tensor([-3.4781, -0.3689, -6.9135, -0.1291, -3.7820, -0.8013, -0.1081, -0.1459,

# -0.1595, -5.0331])), ('hidden.2.weight', tensor([[-1.8987e-01, -2.1553e-01, -2.4452e-01, -1.2826e-01, -4.4600e-01,

# -3.9230e-02, -1.3994e-01, 2.4188e-01, -1.8353e-01, -3.7245e-02],

# [-1.2330e+00, 1.4935e-01, -2.1006e+00, 2.2815e-01, -1.3141e+00,

# -4.9419e-01, -2.0292e-01, -3.7244e-02, -1.8308e-02, -1.7430e+00],

# [-2.9740e+00, 2.6738e-01, -6.7490e+00, 3.9170e-01, -3.8174e+00,

# -6.6887e-01, 8.7731e-02, -1.1104e-01, -2.6860e-01, -4.4536e+00],

# [-3.8733e-01, 1.7222e-01, -6.1735e-01, 1.8737e-01, -4.5800e-01,

# -1.2687e-01, -2.8769e-01, 3.4088e-02, 1.3322e-01, -5.5480e-01],

# [-2.8074e-01, 1.0812e-01, 7.2275e-02, -1.6865e-01, -2.5029e-01,

# -2.1132e-02, 1.5517e-01, 9.2169e-02, 9.1193e-02, -4.0287e-02],

# [-1.1228e+00, -9.4796e-02, -1.7891e+00, -1.0016e-01, -8.4967e-01,

# -4.5255e-01, -2.8438e-01, 6.7924e-02, -4.4027e-02, -1.0355e+00],

# [-7.0236e+00, 2.0964e-01, -1.5519e+01, 7.8860e-03, -7.8425e+00,

# -1.5617e+00, 4.7506e-01, 3.0627e-01, 4.2218e-02, -1.0231e+01],

# [ 1.0062e-01, 1.9500e-01, 1.1216e-02, 2.4669e-01, -2.5071e-02,

# 2.0878e-01, -3.1347e-01, -8.7177e-02, 8.6670e-02, -2.5709e-01],

# [-4.4382e+00, 5.1471e-01, -9.3459e+00, -6.2396e-02, -4.8647e+00,

# -8.5413e-01, 8.7904e-02, 2.1781e-01, -1.4455e-01, -6.1295e+00],

# [-1.5022e+00, 8.5380e-02, -3.0917e+00, 2.0259e-01, -2.0209e+00,

# -9.6872e-02, -1.2980e-01, -5.7904e-02, 9.4876e-02, -2.2930e+00]])), ('hidden.2.bias', tensor([-0.2126, -0.2515, -1.5094, -0.2702, -0.2527, -0.0605, -2.9485, -0.1605,

# -1.4649, -0.6159])), ('regression.weight', tensor([[ -0.0986, -2.6625, -8.5537, 0.1091, -0.1195, -2.5662, -18.3474,

# 0.0404, -10.5618, -3.9268]])), ('regression.bias', tensor([31.6779]))])