**中国矿业大学计算机学院**

**2019 级本科生课程作业**

|  |  |
| --- | --- |
| 课程名称 | 机器学习及优化——实验一 |
| 报告时间 | 2023年4月23日 |
| 学生姓名 | 许万鹏 |
| 学 号 | 05191643 |
| 班 级 | 信息安全2019-01班 |
| 任课教师 | 李政伟 |

摘要

房价是体现经济运转好坏的重要指标，为了更好地预测房价，我们可以使用机器学习技术。在本次实验中，本人首先使用机器学习库Scikit-Learn实现了一种简洁且有效的预测方式。为了更深入地理解各机器学习算法的原理，将理论知识工程化，本人基于目前最常用的深度学习框架PyTorch，从零实现了本次任务中用到的所有机器学习算法，如SGD、Adam等，并将它们组合成了一个完整的框架——XuTorch，最后调用该框架完成了波士顿房价的预测任务。

目 录

[摘要 II](#_Toc133671080)

[1 概述 1](#_Toc133671081)

[1.1 数据集介绍 1](#_Toc133671082)

[1.2 数据探索 1](#_Toc133671083)

[2 基于SkLearn的简洁实现 3](#_Toc133671084)

[3 基于PyTorch的从零实现 4](#_Toc133671085)

[3.1 调用XuTorch的实现 4](#_Toc133671086)

[3.2 基于Pytorch实现XuTorch 10](#_Toc133671087)

[3.2.1 xutorch.dataset.prep\_dataloader 11](#_Toc133671088)

[3.2.2 xutorch.dataset.BOSTON 12](#_Toc133671089)

[3.2.3 xutorch.misc.set\_seeds 14](#_Toc133671090)

[3.2.4 xutorch.misc.get\_device 14](#_Toc133671091)

[3.2.5 xutorch.model.Linear 14](#_Toc133671092)

[3.2.6 xutorch.optim.SimpleSGD 15](#_Toc133671093)

[3.2.7 xutorch.loss.MSE 17](#_Toc133671094)

[3.2.8 xutorch.plot.\* 17](#_Toc133671095)

[3.2.9 xutorch.misc.r2\_score 19](#_Toc133671096)

# 概述

## 数据集介绍

该数据集出自Harrison, D.和Rubinfeld, D.L.的Hedonic prices and the demand for clean air，于美国卡耐基梅隆大学数据库开放获取。其共有506条样本，每条样本由13列特征和1列平均房价数据组成，如下表。

|  |  |
| --- | --- |
| 变量 | 意义 |
| CRIM | 人均犯罪率 |
| ZN | 住宅用地比例（大于25,000平方英尺的地块） |
| INDUS | 非零售业务占地比例 |
| CHAS | Charles River虚拟变量（如果地段边界为河流，则为1;否则为0） |
| NOX | 一氧化氮浓度（每1000万份） |
| RM | 住宅的平均房间数 |
| AGE | 自有住房的比例（建于1949年之前） |
| DIS | 距五个波士顿就业中心的加权距离 |
| RAD | 径向公路可达性指数 |
| TAX | 全额物业税率（每10000美元） |
| PTRATIO | 学生与教师的比例 |
| B | 其中是黑人的比例 |
| LASTAT | 低地位人口/弱势群体百分比 |
| MEDV | 自用住房的中位数价值（每1000美元） |

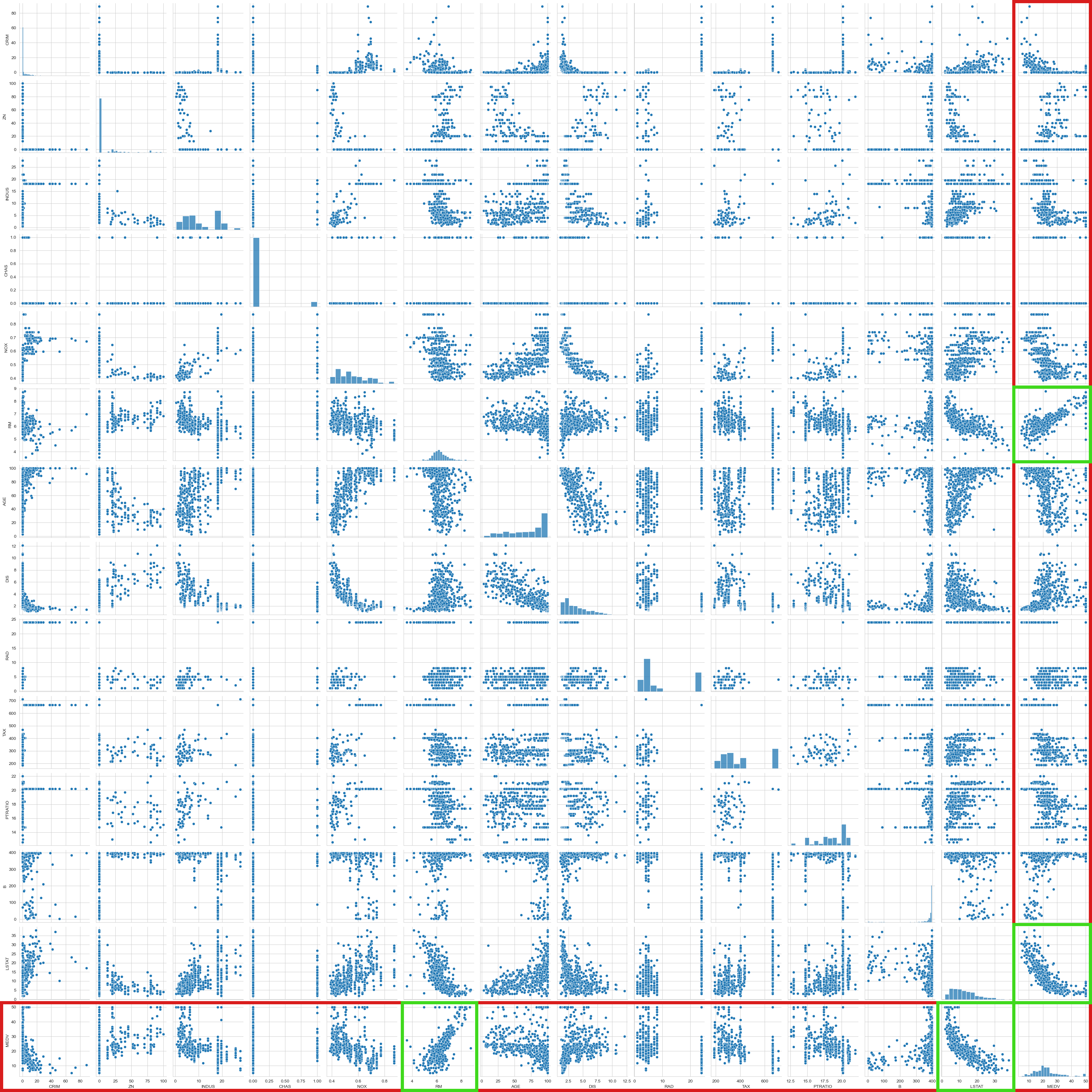
## 数据探索

随着深度学习的流行，特征工程的重要性已逐渐减弱，因为大型的深度神经网络甚至可以学习到特征的重要性。因为波士顿房价数据集本身的特征少，只有13种，所以我的倾向是全部使用，让模型自己学习重要程度。这里可以进行一些简单的数据探索。

略过数据清洗（df.isnull().sum()等类似工作），我们来分析一下数据相关性。

import pandas as pd  
import seaborn as sns  
  
raw\_df = pd.read\_csv("http://lib.stat.cmu.edu/datasets/boston", sep="\s+", skiprows=22, header=None)  
  
*# 取出每一行的全部数据（11列特征）和其下一行的前3列数据（2列特征和1列目标）*left = raw\_df.iloc[::2, :].reset\_index(drop=True)  
right = raw\_df.iloc[1::2, :3].reset\_index(drop=True)  
  
*# 拼接形成完整数据*df = pd.concat([left, right], axis=1)  
df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']  
  
sns.pairplot(df)

输出如下。



通过最后一行和一列可以看出，大部分变量都与房价MEDV有线性关系，其中RM和LSTAT有明显线性关系。

# 基于SkLearn的简洁实现

首先，导入sklearn中已实现的函数。

from sklearn.datasets import load\_boston  
from sklearn.linear\_model import SGDRegressor  
from sklearn.metrics import mean\_squared\_error  
from sklearn.preprocessing import StandardScaler  
from sklearn.model\_selection import train\_test\_split

然后，按照机器学习的一般流程：准备数据、数据分析、特征工程、模型选择及训练、模型评估，逐个调用函数。

*# 导入波士顿房价数据集*dataset = load\_boston()  
  
*# 随机划分训练集和测试集*x\_train, x\_test, y\_train, y\_test = train\_test\_split(dataset.data, dataset.target, random\_state=42)  
  
*# 特征工程：标准化数据，让数据符合正态分布，加速模型训练*transfer = StandardScaler()  
x\_train = transfer.fit\_transform(x\_train)  
x\_test = transfer.fit\_transform(x\_test)  
  
*# 创建模型及优化器的实例，指定学习率为常数，初始学习率为0.01，最大迭代次数为10000*estimator = SGDRegressor(learning\_rate="constant", eta0=0.01, max\_iter=10000)  
  
*# 训练*estimator.fit(x\_train, y\_train)  
  
*# 输出模型参数w和b*print("模型权重w:", estimator.coef\_, sep='\n')  
print("模型偏置b:", estimator.intercept\_, sep='\n')  
  
*# 使用测试集进行预测*y\_predict = estimator.predict(x\_test)  
  
*# 计算均方误差MSE，评估模型预测性能*error = mean\_squared\_error(y\_test, y\_predict)  
print("均方误差MSE:", error, sep='\n')  
  
*# 计算R2分数，评估模型拟合性能*r2\_score = estimator.score(x\_test, y\_test)  
print("R2分数:", r2\_score, sep='\n')

得到输出如下。

模型权重w:

[-0.78152826 0.91750124 0.28393219 0.88823214 -1.88078487 2.99846562

-0.3911255 -2.94062588 2.69974879 -1.09607759 -1.28587891 1.52502515

-3.88396781]

模型偏置b:

[22.73504009]

均方误差MSE:

23.92961160952027

R2分数:

0.6582809127892335

可以看出，基于sklearn的代码可以很简洁地（不超过30行）得到较好的结果（R2分数0.658）。

# 基于PyTorch的从零实现

## 调用XuTorch的实现

导入PyTroch和XuTroch库

import torch  
import xutorch

设置参数

args = {  
 'seed': 42,  
 'lr': 0.00005,  
 'num\_epochs': 10000,  
 'batch\_size': 16,  
 'early\_stop': 500,  
 'save\_path': 'weights/hw1\_model.pth'  
}

创建数据加载器

train\_loader, train\_dataset = xutorch.dataset.prep\_dataloader(  
 dataset\_name='BOSTON',  
 download=True,  
 train=True,  
 batch\_size=args['batch\_size'],  
 test\_ratio=0.25,  
 transform=True  
)  
val\_loader, val\_dataset = xutorch.dataset.prep\_dataloader(  
 dataset\_name='BOSTON',  
 download=True,  
 train=False,  
 batch\_size=args['batch\_size'],  
 test\_ratio=0.25,  
 transform=True  
)

输出如下。

Finished reading the train set of BOSTON Dataset (381 samples found, each dim = 13)

Finished reading the val set of BOSTON Dataset (125 samples found, each dim = 13)

创建模型、优化器、损失函数

*# 设置随机种子*xutorch.misc.set\_seeds(42)  
  
*# 获取计算设备*device = xutorch.misc.get\_device()  
  
*# 创建一个线性回归模型并放到计算设备上*model = xutorch.model.Linear(input\_dim=train\_loader.dataset.dim).to(device)  
  
*# 创建一个SGD优化器*optimizer = xutorch.optim.SimpleSGD(model.parameters(), lr=args['lr'], momentum=0.9)  
  
*# 创建一个MSE损失函数*criterion = xutorch.loss.MSE()

训练模型

loss\_record = {  
 'train': [], *# 记录在训练集上的loss* 'val': [] *# 记录在验证集上的loss*}  
  
min\_val\_loss = 0 *# 验证集上的最小loss*min\_val\_loss\_epoch = 0 *# 验证集上的最小loss对应的epoch*early\_stop\_cnt = 0 *# 早停epochs数，超过阈值时早停*for epoch in range(args['num\_epochs']):  
 train\_loss = 0  
  
 model.train()  
 for batch\_idx, (inputs, targets) in enumerate(train\_loader):  
 inputs, targets = inputs.to(device), targets.to(device)  
  
 *# 前向传播* outputs = model(inputs)  
  
 *# 计算loss* loss = criterion(outputs, targets)  
  
 *# 梯度清零，反向传播，参数更新* optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
  
 train\_loss += loss.item()  
  
 *# 记录在训练集上的平均loss* loss\_record['train'].append(train\_loss / len(train\_dataset))  
  
 val\_loss = 0  
  
 model.eval()  
 for batch\_idx, (inputs, targets) in enumerate(val\_loader):  
 inputs, targets = inputs.to(device), targets.to(device)  
 with torch.no\_grad():  
 outputs = model(inputs)  
 loss = criterion(outputs, targets)  
 val\_loss += loss.item()  
 loss\_record['val'].append(val\_loss / len(val\_dataset))  
  
 *# 打擂法记录最小loss及其对应的epoch和weights* if epoch == 0 or val\_loss < min\_val\_loss:  
 min\_val\_loss = val\_loss  
 min\_val\_loss\_epoch = epoch  
 print(f'Saving model (epoch = {epoch + 1 : 4d}, loss = {min\_val\_loss : .4f})')  
 torch.save(model.state\_dict(), args['save\_path'])  
 *# 模型性能进步，早停轮数清零* early\_stop\_cnt = 0  
 else:  
 *# 模型性能未进步，计入早停轮数* early\_stop\_cnt += 1  
  
 *# 早停轮数超过阈值* if early\_stop\_cnt > args['early\_stop']:  
 print('EARLY STOP')  
 break  
  
 *# 每10轮打印训练集和验证集上的平均loss* if epoch % 10 == 9:  
 print('[{:03d}/{:03d}] Train Loss: {:3.6f} | Val loss: {:3.6f}'.format(  
 epoch + 1, args['num\_epochs'], train\_loss / len(train\_dataset), train\_loss / len(train\_dataset), val\_loss / len(val\_dataset)))

输出如下。

Saving model (epoch = 1, loss = 5032.8027)

Saving model (epoch = 2, loss = 4768.5233)

Saving model (epoch = 3, loss = 4512.2601)

Saving model (epoch = 4, loss = 4292.4985)

Saving model (epoch = 5, loss = 4085.9283)

Saving model (epoch = 6, loss = 3885.4892)

Saving model (epoch = 7, loss = 3699.9818)

Saving model (epoch = 8, loss = 3535.5717)

Saving model (epoch = 9, loss = 3379.0996)

Saving model (epoch = 10, loss = 3222.6381)

[010/10000] Train Loss: 22.740570 | Val loss: 22.740570

Saving model (epoch = 11, loss = 3085.7941)

Saving model (epoch = 12, loss = 2950.0515)

Saving model (epoch = 13, loss = 2825.0240)

Saving model (epoch = 14, loss = 2705.6272)

Saving model (epoch = 15, loss = 2590.1957)

Saving model (epoch = 16, loss = 2482.3602)

Saving model (epoch = 17, loss = 2381.4466)

Saving model (epoch = 18, loss = 2282.9214)

Saving model (epoch = 19, loss = 2188.8889)

Saving model (epoch = 20, loss = 2100.2793)

[020/10000] Train Loss: 14.490291 | Val loss: 14.490291

...

[1010/10000] Train Loss: 1.440343 | Val loss: 1.440343

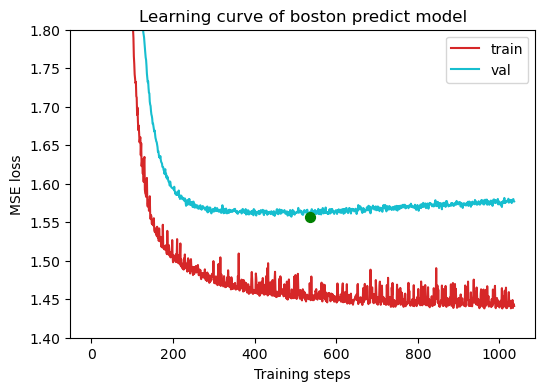
[1020/10000] Train Loss: 1.439115 | Val loss: 1.439115

[1030/10000] Train Loss: 1.441514 | Val loss: 1.441514

\*\*EARLY STOP\*\*

绘制学习曲线

xutorch.plot.plot\_learning\_curve(loss\_record, loss\_name='MSE', title='boston predict model', bottom=1.4, top=1.8, min\_loss\_x=min\_val\_loss\_epoch, min\_loss\_y=min\_val\_loss/len(val\_dataset))

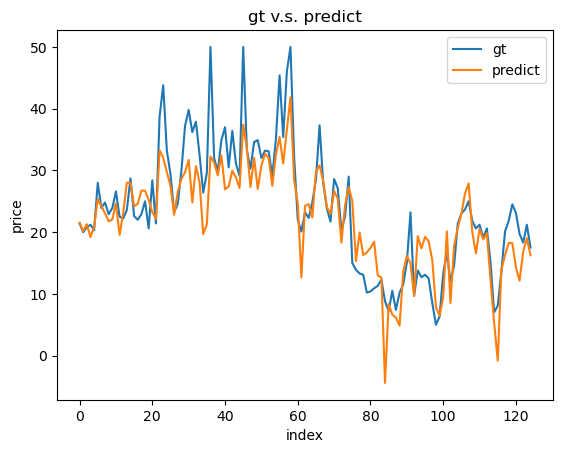


计算预测值，和真实值对比以评估模型

model.eval()  
targets, preds = [], []  
for X, y in val\_loader:  
 X, y = X.to(device), y.to(device)  
 with torch.no\_grad():  
 pred = model(X)  
 preds.append(pred.detach().cpu())  
 targets.append(y.detach().cpu())

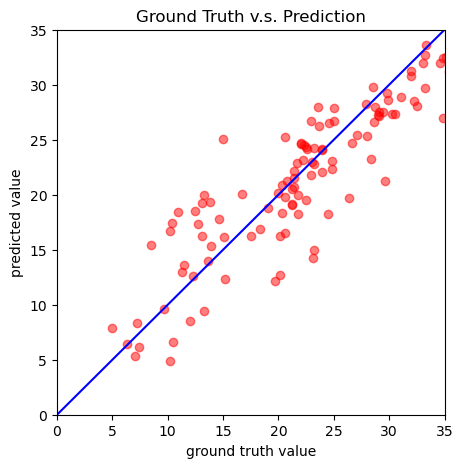
绘制拟合曲线

xutorch.plot.plot\_gt\_vs\_pred(targets, preds)



绘制预测-真实值散点图

xutorch.plot.plot\_gt\_and\_pred(targets, preds)



计算R2分数，可以看出R2分数略高于同随机种子下的基于sklearn的实现。事实上，将线性回归模型换做MLP或深度神经网络可以得到更高的分数。

r2 = xutorch.misc.r2\_score(targets, preds)  
print("R2 score:", r2)

输出如下。

R2 score: 0.7478697299957275

分析权重系数，可以发现RAD和RM特征的系数较大（在不同种子下二者的排名会有变化），LSTAT特征的系数较小。这说明在波士顿，买房者十分注重房子的房间数和交通便利度，这两项越好房价越贵；同时十分在意社区的居住成员，其中的弱势群体越高房价越低。

params = model.state\_dict()  
  
for name, param in params.items():  
 print(name, param)  
  
feature\_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']  
  
feature\_names[torch.max(model.fc.weight, 1)[1].item()], feature\_names[torch.min(model.fc.weight, 1)[1].item()]

输出如下。

fc.weight tensor([[-0.6342, 1.0948, 0.6785, 0.5673, -2.5955, 2.6878, 0.1956, -3.0039, 2.7394, -1.4655, -2.2621, 1.4881, -3.6845]], device='cuda:0')

fc.bias tensor([22.1328], device='cuda:0')

('RAD', 'LSTAT')

## 基于Pytorch实现XuTorch

这里介绍**本次任务所用到的**XuTorch中的机器学习算法的实现过程。

XuTorch

│ \_\_init\_\_.py

│

├─dataset

│ │ **boston.py**

│ │ breast\_cancer.py

│ │ **prep\_dataloder.py**

│ │ sentiment.py

│ │ \_\_init\_\_.py

│ │

│ └─data

│ **boston.txt**

│ huangguo\_mountain.csv

│ wdbc.data

│

├─loss

│ cross\_entropy.py

│ hinge.py

│ **mean\_squared\_error.py**

│ \_\_init\_\_.py

│

├─misc

│ **get\_device.py**

│ **r2\_score.py**

│ **set\_seeds.py**

│ \_\_init\_\_.py

│

├─model

│ embedding\_mlp.py

│ **linear.py**

│ mlp.py

│ svm.py

│ \_\_init\_\_.py

│

├─optim

│ adam.py

│ **sgd.py**

│ \_\_init\_\_.py

│

└─plot

**plot.py**

\_\_init\_\_.py

### xutorch.dataset.prep\_dataloader

这里定义了一个数据加载函数prep\_dataloader，其目的是根据数据集名称生成相应数据集对象并将其放入数据加载器中。

from xutorch.dataset.boston import BOSTON  
from xutorch.dataset.breast\_cancer import BreastCancer  
from xutorch.dataset.sentiment import Sentiment  
from torch.utils.data import DataLoader  
  
  
def prep\_dataloader(dataset\_name, download, train, batch\_size, num\_workers=0, test\_ratio=0.25, \*\*kwargs):  
 dataset = None  
 if dataset\_name == 'BOSTON':  
 dataset = BOSTON(download=download, train=train, test\_ratio=test\_ratio, \*\*kwargs)  
 elif dataset\_name == 'BreastCancer':  
 dataset = BreastCancer(download=download, train=train, test\_ratio=test\_ratio, \*\*kwargs)  
 elif dataset\_name == 'Sentiment':  
 dataset = Sentiment(download=download, train=train, test\_ratio=test\_ratio, \*\*kwargs)  
  
 dataloader = DataLoader(dataset, batch\_size, shuffle=train, drop\_last=False, num\_workers=num\_workers, pin\_memory=True)  
 return dataloader, dataset

### xutorch.dataset.BOSTON

该代码实现了一个 BOSTON 数据集的 PyTorch 数据集类，提供了 BOSTON 数据集的读取和预处理方法。其中包含以下属性和方法：

data\_url：BOSTON 数据集的 URL。

current\_dir：当前文件所在的目录。

\_\_init\_\_：初始化函数，负责解析 BOSTON 数据集并将其处理成 PyTorch 数据集类可以使用的形式。

download：是否需要下载数据集，默认为 True，即需要下载。

path：数据集存储路径，默认为 current\_dir 下的 data/boston.txt。

train：是否使用训练集，默认为 True。

transform：是否进行数据集的预处理（归一化），默认为 True。

test\_ratio：测试集的比例，默认为 0.25。

\_\_len\_\_：获取数据集的大小。

\_\_getitem\_\_：获取指定索引的数据样本。

data：BOSTON 数据集的特征数据。

targets：BOSTON 数据集的目标数据。

dim：BOSTON 数据集的特征维度。

在 \_\_init\_\_ 方法中，首先会读取数据集并将其分割为特征数据和目标数据两个部分。其中，特征数据包含 13 个维度的特征，而目标数据仅包含一个维度的标签。然后根据 train 参数将数据集分为训练集和测试集，并且将数据集进行了归一化处理。最后输出数据集的大小信息。

import os  
import torch  
import pandas as pd  
from torch.utils.data import Dataset  
  
  
class BOSTON(Dataset):  
 data\_url = "http://lib.stat.cmu.edu/datasets/boston"  
 current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
  
 def \_\_init\_\_(self, download=True, path=os.path.join(current\_dir, 'data', 'boston.txt'), train=True, transform=True, test\_ratio=0.25):  
 if download:  
 path = self.data\_url  
 column\_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']  
 raw\_df = pd.read\_csv(path, sep="\s+", skiprows=22, header=None)  
 *# 取出每一行的全部数据（11列特征）和其下一行的前3列数据（2列特征和1列目标）* left = raw\_df.iloc[::2, :].reset\_index(drop=True)  
 right = raw\_df.iloc[1::2, :3].reset\_index(drop=True)  
  
 df = pd.concat([left, right], axis=1)  
 df.columns = column\_names  
  
 data = df.drop('MEDV', axis=1)  
 targets = df['MEDV']  
  
 *# 不建议对全集归一化  
 # mean = np.mean(data, axis=0)  
 # std = np.std(data, axis=0)* pivot = int(100 \* (1 - test\_ratio))  
 if train:  
 indices = [i for i in range(len(data)) if i % 100 < pivot]  
 else:  
 indices = [i for i in range(len(data)) if i % 100 >= pivot]  
  
 self.data = torch.FloatTensor(data.iloc[indices].values)  
 self.targets = torch.FloatTensor(targets.iloc[indices].values)  
  
 if transform:  
 *# 对train或dev集合分别进行归一化* self.data = (self.data - self.data.mean(dim=0, keepdim=True)) / self.data.std(dim=0, keepdim=True)  
  
 self.dim = self.data.shape[1]  
  
 print('Finished reading the {} set of BOSTON Dataset ({} samples found, each dim = {})'.format(  
 'train' if train else 'val', len(self.data), self.dim))  
  
 def \_\_len\_\_(self):  
 return len(self.data)  
  
 def \_\_getitem\_\_(self, index):  
 return self.data[index], self.targets[index]

### xutorch.misc.set\_seeds

固定随机种子以复现相同结果。

import os  
import torch  
import random  
import numpy as np  
  
  
def set\_seeds(seed):  
 torch.manual\_seed(seed)  
 os.environ['PYTHONHASHSEED'] = str(seed)  
 random.seed(seed)  
 np.random.seed(seed)  
 if torch.cuda.is\_available():  
 torch.cuda.manual\_seed(seed)  
 torch.cuda.manual\_seed\_all(seed)  
 *# if torch.backends.cudnn.is\_available():  
 # torch.backends.cudnn.benchmark = False  
 # torch.backends.cudnn.deterministic = True*

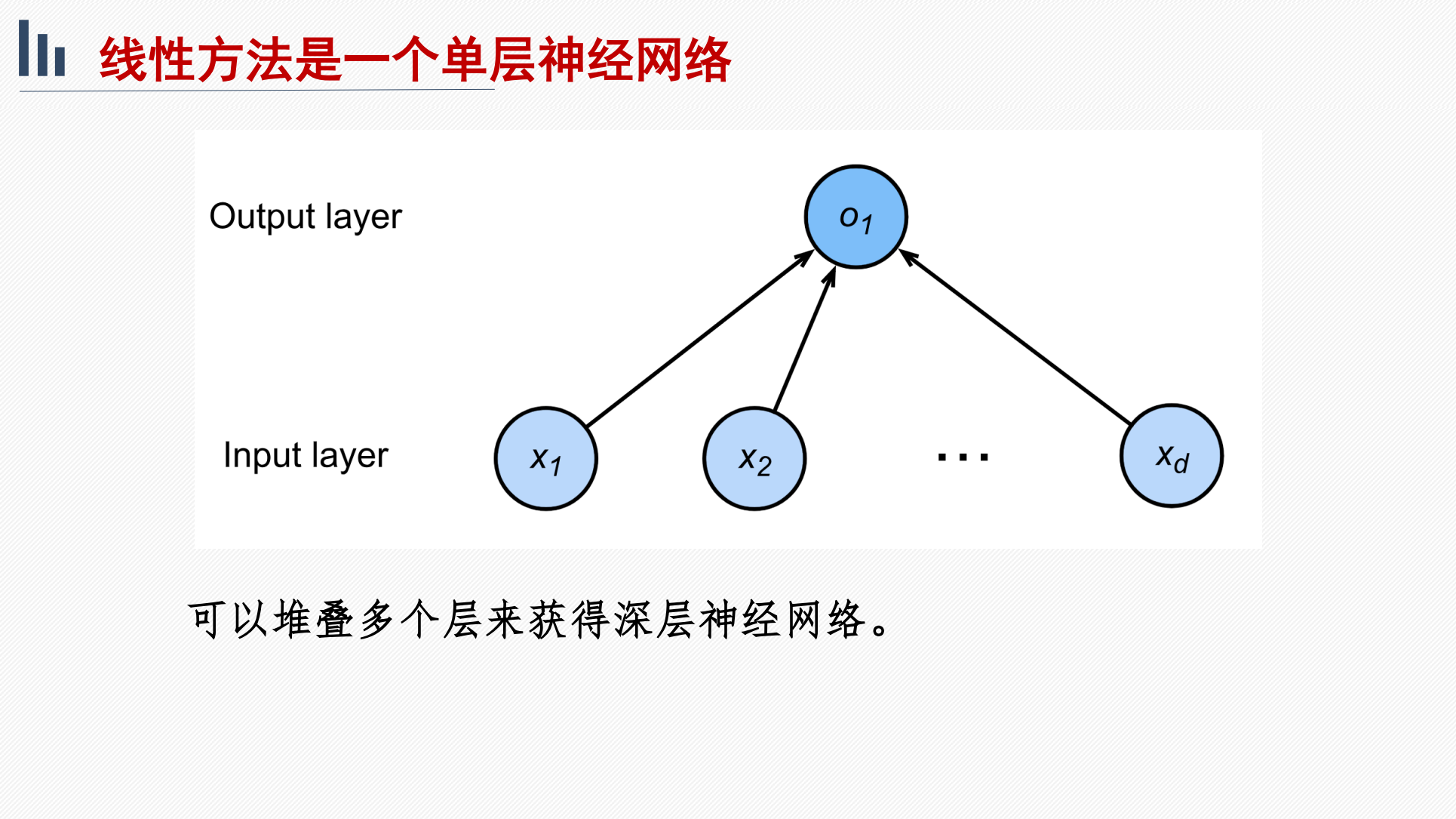
### xutorch.misc.get\_device

检查计算设备：是否有可用的GPU，如果有则使用GPU，否则使用CPU。

import torch  
  
  
def get\_device():  
 return torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

### xutorch.model.Linear

这里实现了一个简单的线性网络，参考自课程PPT中的Chap3 P20 线性方法。



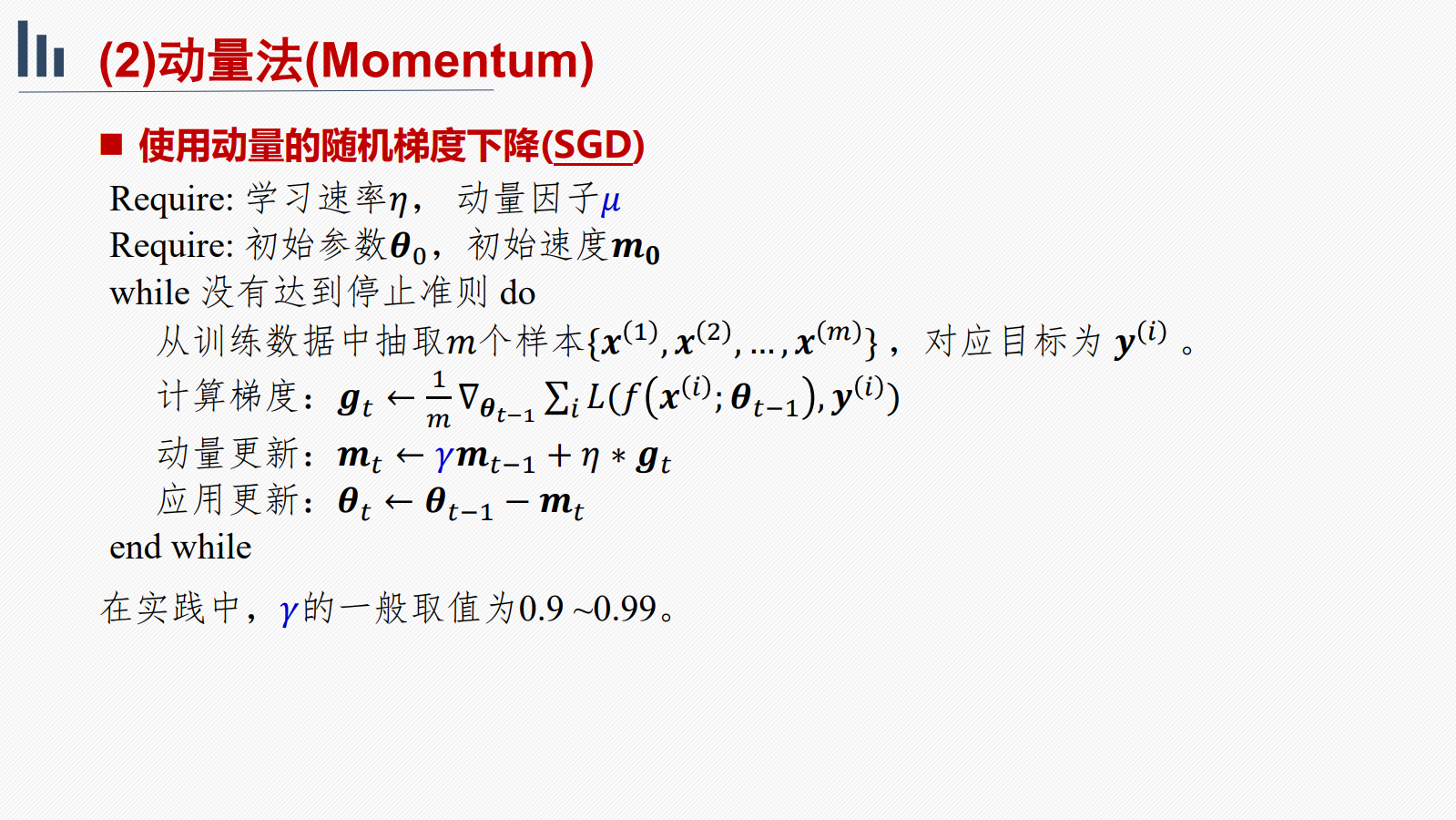
import torch.nn as nn  
  
  
class Linear(nn.Module):  
 def \_\_init\_\_(self, input\_dim):  
 super(Linear, self).\_\_init\_\_()  
  
 *# 线性回归模型* self.fc = nn.Linear(input\_dim, 1)  
  
 def forward(self, x):  
 return self.fc(x).squeeze(1)

### xutorch.optim.SimpleSGD

这里实现了简单的随机梯度下降（SGD）算法。为了省略一些不重要的操作（如zero\_grad和另存参数），我令该类继承自PyTorch的Optimizer类。

构造函数\_\_init\_\_中，接收三个参数：params表示要更新的参数组成的列表，lr是学习率（默认为0.01），momentum是动量（默认为0）。

类中的step方法是该优化器的核心方法，用于更新参数，**参考自课程PPT中的Chap3 P62 动量法**。



在这个方法中，首先遍历参数组group，然后遍历每个参数p，如果p没有梯度，则跳过。如果p有梯度，则获取该参数的状态state，以及学习率lr和动量momentum。如果状态中没有动量缓存，则创建一个全零的tensor作为缓存。然后将该参数的梯度与动量缓存进行累积，根据学习率和动量的大小对梯度进行加权和，更新参数p的数值。注意，如果不在step内部清空梯度（p.data=None），则一定要在代码外部显式地书写optimizer.zero\_grad()

需要注意的是，对于原地方法buf.mul\_和p.data.add\_中的alpha参数，它们的含义分别是乘数和加数的系数，用于对乘积或加和进行缩放。

import torch  
from torch.optim import Optimizer  
  
  
class SimpleSGD(Optimizer):  
 def \_\_init\_\_(self, params, lr=0.01, momentum=0):  
 defaults = dict(lr=lr, momentum=momentum)  
 super(SimpleSGD, self).\_\_init\_\_(params, defaults)  
  
 def step(self):  
 for group in self.param\_groups:  
 for p in group['params']:  
 if p.grad is None:  
 continue  
 grad = p.grad.data  
 state = self.state[p]  
  
 lr = group['lr']  
 momentum = group['momentum']  
 if 'momentum\_buffer' not in state:  
 buf = state['momentum\_buffer'] = torch.zeros\_like(p.data)  
 else:  
 buf = state['momentum\_buffer']  
 *# buf = momentum \* buf + grad* buf.mul\_(momentum).add\_(grad)  
  
 *# p.data += -self.lr \* buf* p.data.add\_(buf, alpha=-lr) *# new*

### xutorch.loss.MSE

该类用于计算均方误差（MSE）损失函数。

Forward方法实现了接受两个tensor作为输入参数inputs和targets，计算两者差的平方的平均值，即均方误差损失函数的值，并返回该值。

该损失函数通常用于回归问题，其中inputs表示预测值，targets表示真实值。MSE损失函数越小，表示模型预测值与真实值之间的差距越小，模型的性能越好。

import torch  
  
  
class MSE(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super(MSE, self).\_\_init\_\_()  
  
 def forward(self, outputs, targets):  
 return torch.mean((outputs - targets) \*\* 2)

### xutorch.plot.\*

这里包含了三个用于绘图的函数，其中：

plot\_gt\_and\_pred 用于绘制预测值与真实值的散点图（预测-真实值散点图）；

plot\_gt\_vs\_pred 用于绘制真实值与预测值的曲线图（拟合曲线）；

plot\_learning\_curve 用于绘制训练曲线。

import torch  
import matplotlib.pyplot as plt  
from matplotlib.pyplot import figure  
  
  
def plot\_gt\_and\_pred(targets=None, preds=None, bottom=0.0, top=35.0):  
 import os  
 os.environ["KMP\_DUPLICATE\_LIB\_OK"] = "TRUE"  
  
 preds = torch.cat(preds, dim=0).numpy()  
 targets = torch.cat(targets, dim=0).numpy()  
  
 figure(figsize=(5, 5))  
 plt.scatter(targets, preds, c='r', alpha=0.5)  
 plt.plot([bottom, top], [bottom, top], c='b')  
 plt.xlim(bottom, top)  
 plt.ylim(bottom, top)  
 plt.xlabel('ground truth value')  
 plt.ylabel('predicted value')  
 plt.title('Ground Truth v.s. Prediction')  
 plt.show()  
  
  
def plot\_gt\_vs\_pred(y\_true, y\_pred):  
 y\_pred = torch.cat(y\_pred, dim=0).numpy()  
 y\_true = torch.cat(y\_true, dim=0).numpy()  
 plt.plot(y\_true, label="gt")  
 plt.plot(y\_pred, label="predict")  
 plt.xlabel("index")  
 plt.ylabel("price")  
 plt.legend(loc="best")  
 plt.title("gt v.s. predict")  
 plt.show()  
  
  
def plot\_learning\_curve(loss\_record, loss\_name='', title='', bottom=0.0, top=100.0, min\_loss\_x=None, min\_loss\_y=None):  
 x\_1 = range(len(loss\_record['train']))  
 x\_2 = x\_1[::len(loss\_record['train']) // len(loss\_record['val'])]  
 figure(figsize=(6, 4))  
 plt.plot(x\_1, loss\_record['train'], c='tab:red', label='train')  
 plt.plot(x\_2, loss\_record['val'], c='tab:cyan', label='val')  
 if min\_loss\_x and min\_loss\_y:  
 plt.scatter(min\_loss\_x, min\_loss\_y, c='g', alpha=1, marker='o', s=50, zorder=10)  
 plt.ylim(bottom, top)  
 plt.xlabel('Training steps')  
 plt.ylabel('{} loss'.format(loss\_name))  
 plt.title('Learning curve of {}'.format(title))  
 plt.legend()  
 plt.show()

### xutorch.misc.r2\_score

这里实现了一个R²评分函数。R²评分是回归模型的一种常见的评价指标，用于衡量模型的拟合程度。

函数的输入是y\_true和y\_pred，分别表示真实值和预测值。函数计算y\_true的平均值，根据其计算总平方和（ss\_total）和残差平方和（ss\_residual）。最后，根据



将R²分数计算为1减去残差平方和与总平方和之比，并将其返回。

import torch  
  
  
def r2\_score(y\_true, y\_pred):  
 y\_true = torch.cat(y\_true, dim=0)  
 y\_pred = torch.cat(y\_pred, dim=0)  
 y\_mean = torch.mean(y\_true)  
 ss\_total = torch.sum((y\_true - y\_mean) \*\* 2)  
 ss\_residual = torch.sum((y\_true - y\_pred) \*\* 2)  
 r2 = 1 - (ss\_residual / ss\_total)  
 return r2.item()