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

1 概述

1.1 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	$1000(B_k-0.63)^2$ 其中 B_k 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）

1.2 数据探索

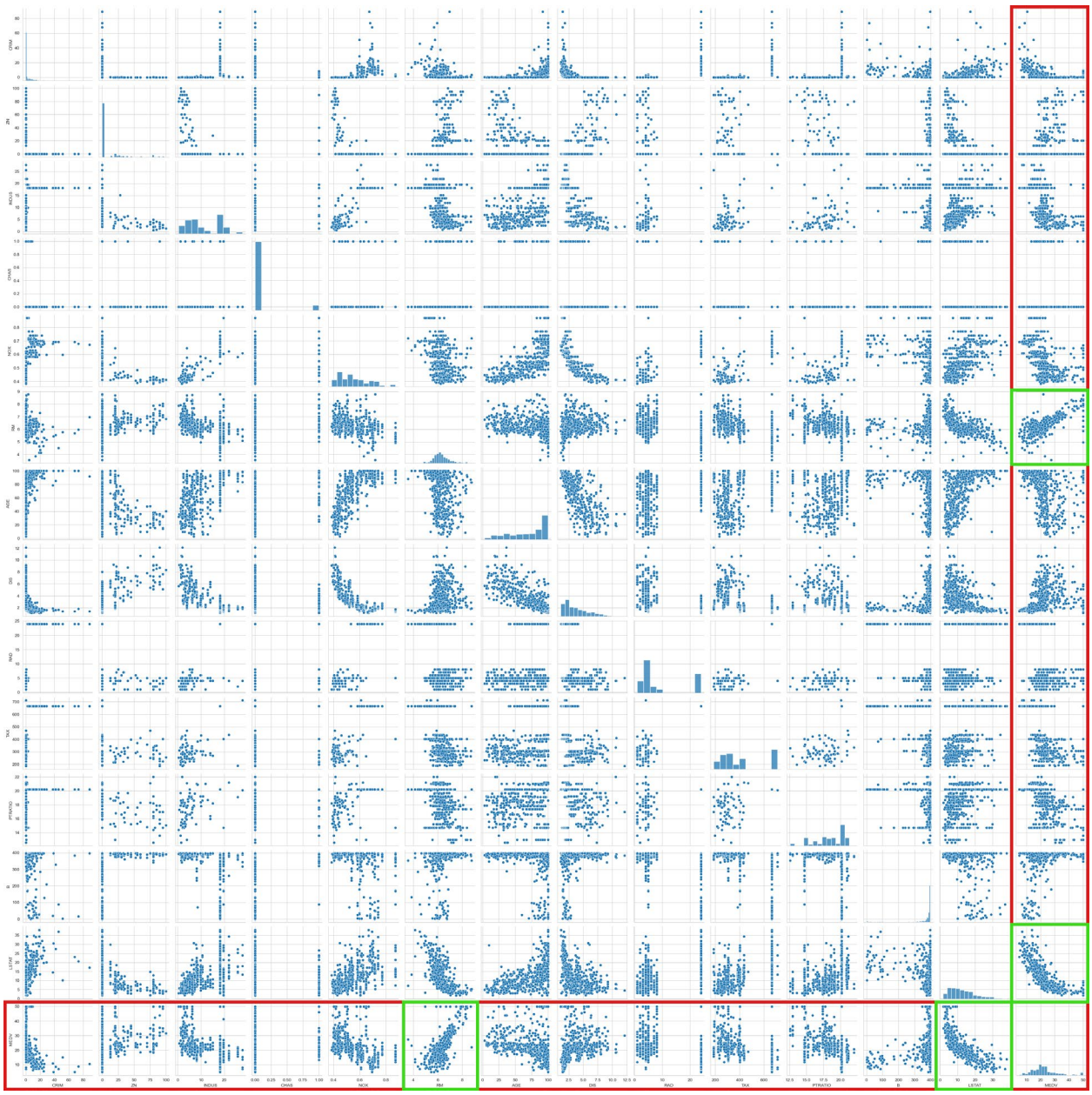
随着深度学习的流行，特征工程的重要性已逐渐减弱，因为大型的深度神经网络甚至可以学习到特征的重要性。因为波士顿房价数据集本身的特征少，只有 13 种，所以我的倾向是全部使用，让模型自己学习重要程度。这里可以进行一些简单的数据探索。

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。

2 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）。

3 基于 PyTorch 的从零实现

3.1 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
```

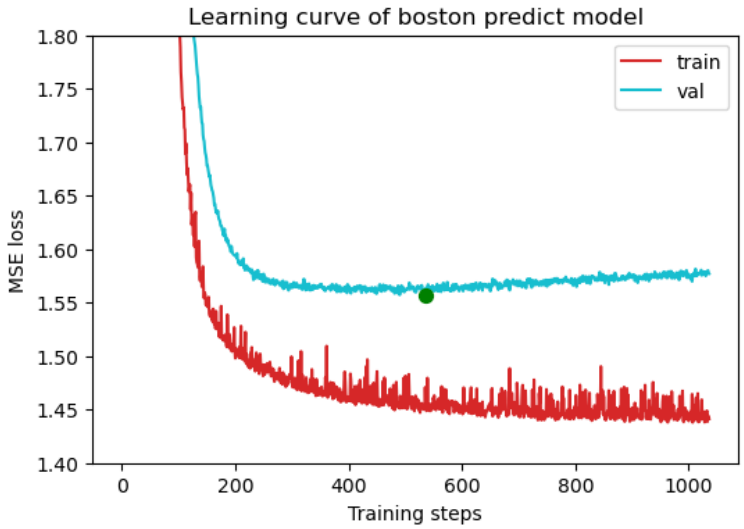
```
    if early_stop_cnt > args['early_stop']:
        print('EARLY STOP')
        break

    # 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))
```

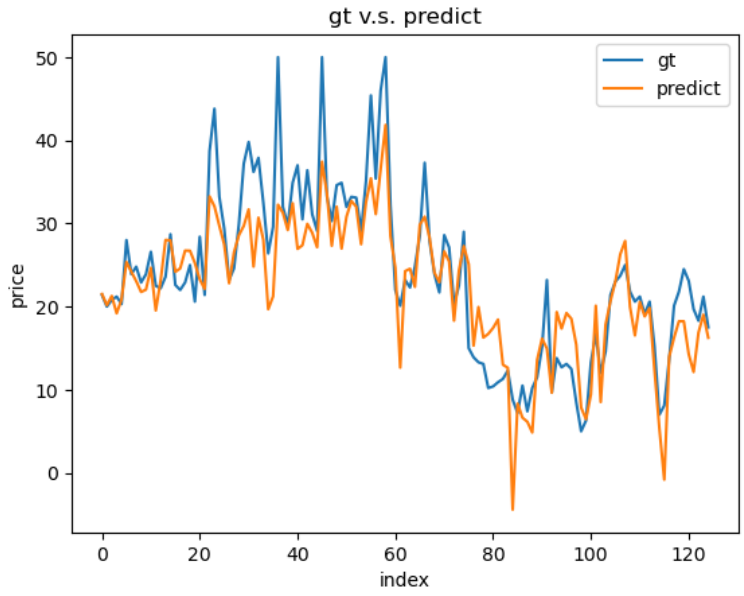


计算预测值，和真实值对比以评估模型

```
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())
```

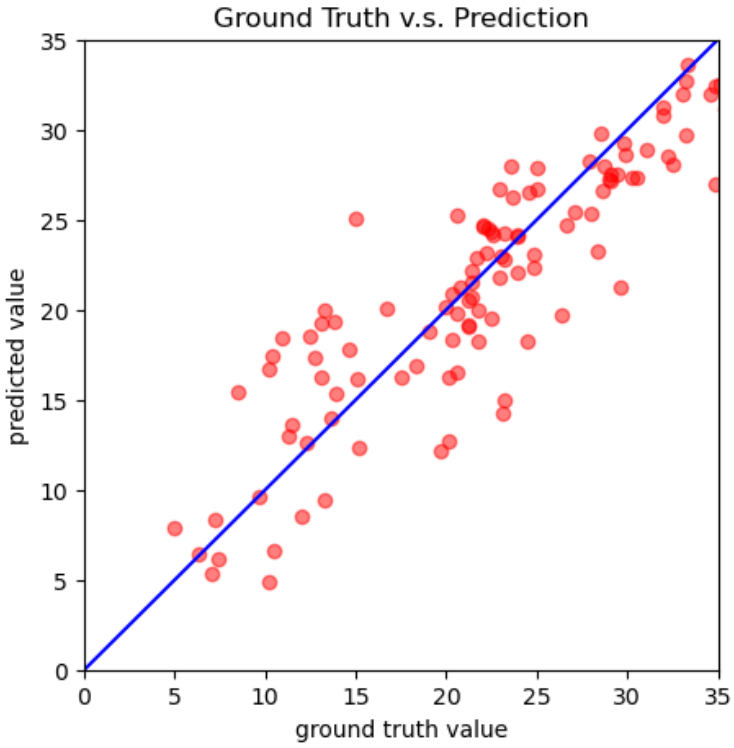
绘制拟合曲线

```
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')
```

3.2 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
```

3.2.1 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
```

3.2.2 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]
```

3.2.3 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
```

3.2.4 xutorch.misc.get_device

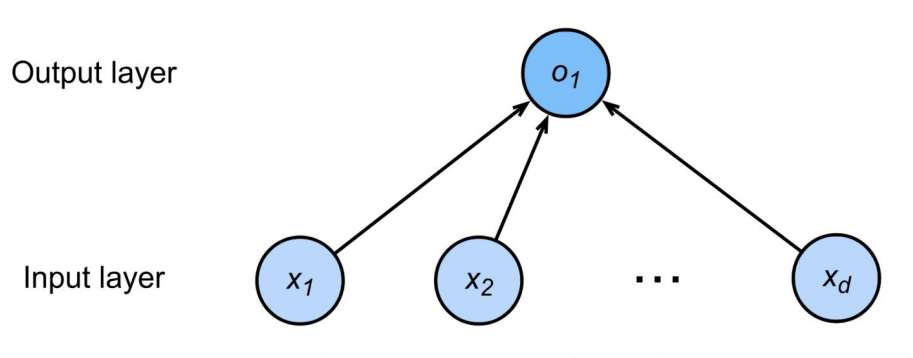
检查计算设备：是否有可用的 GPU，如果有则使用 GPU，否则使用 CPU。

```
import torch


def get_device():
    return torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

3.2.5 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。

线性方法是一个单层神经网络



可以堆叠多个层来获得深层神经网络。

```
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)
```

3.2.6 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**。

(2)动量法(Momentum)

■ 使用动量的随机梯度下降(SGD)

Require: 学习速率η， 动量因子μ
Require: 初始参数$\boldsymbol{\theta}_0$，初始速度$\boldsymbol{m_0}$
while 没有达到停止准则 do
　从训练数据中抽取m个样本$\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \dots, \boldsymbol{x}^{(m)}\}$，对应目标为 $\boldsymbol{y}^{(i)}$。
　计算梯度：$\boldsymbol{g}_t \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}_{t-1}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_{t-1}), \boldsymbol{y}^{(i)})$
　动量更新：$\boldsymbol{m}_t \leftarrow \gamma \boldsymbol{m}_{t-1} + \eta * \boldsymbol{g}_t$
　应用更新：$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \boldsymbol{m}_t$
end while

在实践中，γ的一般取值为0.9 ~0.99。

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
```

3.2.7 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)
```

3.2.8 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()
```

3.2.9 xutorch.misc.r2_score

这里实现了一个 R^2 评分函数。R^2 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

$$R^2 = 1 - \left(ss\_residual \ / \ ss\_total\right)$$

将 R^2 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()
```