训练，已知一定数量的（x, y），学习出Y~X的关系

预测：面对新出现的X预测Y

机器学习算法构成3要素

假设空间

优化目标

寻解算法

Minθ(Eval(H(θ, X), Y)

θ参数

X输入

Y输出

Eval已知样本

神经网络三个关键点：加权和（线性变换），非线性变换，多层连接

Y=f3(f2(f1(w1x1+w2x2+w3x3+b)+...)+...)

matplotlab

# 波士顿房价预测模型

模型：多元线性回归

**回归**问题：输出为连续值（实数），如身高

分类问题：输出为离散值，如性别

一层神经网络

激活函数

线性回归中activation\_fn为

* 数据处理：读取数据和预处理操作
* 模型设计：网络结构（假设）
* 训练配置：优化器（寻解算法），计算资源配置
* 训练过程：循环调用训练过程，前向计算+损失函数（优化目标）+后向传播
* 保存模型：将训练好的模型保存

## 读入数据

import numpy as np

import json

datafile='./housing.data'

data=np.fromfile(datafile, sep=' ')

* numpy.fromfile(file, dtype=float, count=-1, sep='', offset=0)

读一个已知数据类型的文件

sep=''，说明原文件是一个二进制文件，sep=' '说明文件中数字的分隔符至少是一个空格

本例housing.data分隔符是两个空格

将数据读入到一个数组中，data = array([6.320e-03, 1.800e+01, 2.310e+00, ..., 1.190e+01])

data是numpy.ndarray类型

feature\_names = [ 'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE','DIS',

'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV' ]

feature\_num = len(feature\_names)

data = data.reshape([data.shape[0] // feature\_num, feature\_num])

x = data[0]

print(x.shape)

print(x)

将只有一行的一维数据的形状进行变换，形成一个2维的矩阵，每行为一个数据样本（14个值），每个数据样本包含13个X（影响房价的特征）和一个Y（该类型房屋的均价）。

* /表示浮点数除法，返回浮点结果。//表示整数除法，结果取整。
* data.shape属性返回维度demensions的元组。不如就叫形状

维度，对于只有一行的data，返回(元素个数,)。和上面一维二维的意思不一样

data.shape=(7084,)，data数组中一共7084个数

7084//14=506，说明一共506行记录

data1=np.array([[1,2,3,2], [4,5,6,7]])

data1.shape

(2, 4)

np.array() list转np.ndarray

* data.reshape([506, 14])

把这一组数字分成506组，每组14个

现在data是一个506\*14的矩阵，504行\*14列

x = data[0]

print(x.shape)

print(x)

data[0]查看第0行数据

data(0).shape当然是(14, 0)

data.shape就变成了(506, 14)

## 数据集划分

将数据集划分成训练集和测试集

训练集确定模型参数

测试集用于评价模型效果

将80%的数据用作训练集，20%用作测试集

ratio = 0.8

offset = int(data.shape[0] \* ratio)

training\_data = data[:offset]

ndarray也可以进行切片操作

训练集的形状共有404个样本，每个样本含有13个特征和1个预测值。

data[:404]取data矩阵中前404行的数据

## 归一化处理

每个特征的取值缩放到0~1之间

使模型训练更高效

特征前的权重大小可代表该变量对预测结果的贡献度

预测时，样本数据同样也需要归一化，以训练样本的均值和极值计算

计算train数据集的最大值，最小值，平均值

maximums, minimums, avgs = training\_data.max(axis=0), training\_data.min(axis=0), training\_data.sum(axis=0) / training\_data.shape[0]

* max(axis=0)返回沿固定轴的最大值

默认axis=None，返回展平数组的最大值。

axis=0 计算每一列的最大值。Maxima along the first axis

axis=1 计算每一行的最大值。Maxima along the second axis

* ndarray.sum(axis=0)

计算每一列的和

对数据进行归一化处理

for i in range(feature\_num):

data[:, i] = (data[:, i] - minimums[i]) / (maximums[i] - minimums[i])

逗号分隔维度。单个:表示取此维度的所有值

data[:, i] 表示，第一个维度（行）取所有数据，第二个维度（列）取第i个

所以data[:, 0]提取出了一个ndarray，内容是所有行的第一个字段

归一化，就是所有参数介于0-1之间。最小则为0，最大则为1

training\_data[:, :-1]第一个维度（行）提取所有数据，第二个维度（列）提取前13(14-1)个。

training\_data[:, -1:]第一个维度（行）提取所有数据，第二个维度（列）提取第14个。

## 模型设计

也称为网络结构设计，相当于模型的假设空间，即实现模型“前向计算”（从输入到输出）的过程。

参数是一个列向量，训练数据x是行向量，那么两个向量的矩阵乘积**xw**是一个标量y

w = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, -0.1, -0.2, -0.3, -0.4, 0.0]

w = np.array(w).reshape([13, 1])

以任意数字赋值参数做初始化

取出第1条样本数据，观察样本的特征向量与参数向量相乘的结果。

x1=x[0]

t = np.dot(x1, w)

print(t)

np.dot()矩阵乘积

线性回归模型的完整输出是z=t+b

b = -0.2

z = t + b

初始化偏移量b，随意赋初值-0.2

从特征x和参数w计算输出值的过程称为“前向计算”。上面就是一次完整的向前计算过程

class Network(object):

def \_\_init\_\_(self, num\_of\_weights):

np.random.seed(0)

self.w = np.random.randn(num\_of\_weights, 1)

self.b = 0.

def forward(self, x):

z = np.dot(x, self.w) + self.b

return z

将上述计算预测输出的过程以“类和对象”的方式来描述

seed设置一个种子，后面每次randn产生的随机数是一样的，可以复现

random.randn(x, y)产生一个x\*y的随机数矩阵

此处num\_of\_weights是13

## 训练配置

使用均方误差作为损失函数，衡量预测房价和真实房价的差异

目标：极小化损失函数，确定参数w和b的值

Loss = (y1 - z)\*(y1 - z)

\* 做矩阵的元素级乘法

def loss(self, z, y):

error = z - y

cost = error \* error

cost = np.mean(cost)

return cost

在network中定义损失函数计算方法

z是一个404\*1的向量，y也是。error也是。

-做元素级减法

mean()求平均数

## 训练过程

### 梯度下降法

根据样本数据，找到一组参数(w, b)的值，使得Loss取最小值

* 损失函数在极值点处导数为0，求解下列方程组得到参数w和b
* 发现求导的函数正向计算容易，反向求解较难

探索身边的坡度，梯度下降法

由此可见，均方误差表现的“圆滑”的坡度有两个好处：

* 曲线的最低点是可导的。
* 越接近最低点，曲线的坡度逐渐放缓，有助于通过当前的梯度来判断接近最低点的程度（是否逐渐减少步长，以免错过最低点）。

而绝对值误差是不具备这两个特性的，这也是损失函数的设计不仅仅要考虑“合理性”，还要追求“易解性”的原因。

net = Network(13)

losses = []

w5 = np.arange(-160.0, 160.0, 5.0)

w9 = np.arange(-160.0, 160.0, 5.0)

losses = np.zeros([len(w5), len(w9)])

for i in range(len(w5)):

for j in range(len(w9)):

net.w[5] = w5[i]

net.w[9] = w9[j]

z = net.forward(x)

loss = net.loss(z, y)

losses[i, j] = loss

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

fig = plt.figure()

ax = Axes3D(fig)

w5, w9 = np.meshgrid(w5, w9)

ax.plot\_surface(w5, w9, losses, rstride=1, cstride=1, cmap='rainbow')

plt.show()

* np.arange()返回一个有终点和起点的固定步长的排列

np.arange(-5, 5, 2)

array([-5, -3, -1, 1, 3])

* len(ndarray)当然返回数组长度
* np.zeros([320, 320])返回一个给定形状的用0填充的矩阵
* 先初始化Net，然后分别设置w[5]和w[9]的值，这样就有了一个其他参数都相同，只有w[5]和w[9]不同的参数列表。用forward函数计算z值，再loss函数计算loss，结果记入losses矩阵中
* losses是一个320\*320的矩阵，分别对应w[5]和w[9]取不同值时的值
* x和y都是矩阵

### 计算梯度

损失函数,加二分之一好算一点

其中zi是网络对第i个样本的预测值

为什么要j次方？？？？？？？？？？？？？？？？

看了一万次以后我觉得是xij写成了xi的j次方。所以下面用xi\_j代替xi的j次方

**梯度**定义：Loss对每个参数的偏导数

只有一行数据的情况下，i=1

所以

x1 = x[0]

y1 = y[0]

z1 = net.forward(x1)

print('x1 {}, shape {}'.format(x1, x1.shape))

print('y1 {}, shape {}'.format(y1, y1.shape))

print('z1 {}, shape {}'.format(z1, z1.shape))

计算单行数据的z值

gradient\_w0 = (z1 - y1) \* x1[0]

print('gradient\_w0 {}'.format(gradient\_w0))

计算只有单行数据时w0的梯度

因为x1[0]=0，所以gradient\_w0也为0

可以用一个循环计算出w0-w12的值。更进一步，可以用两个循环计算出，当只有第i行时，每个i对应的w0-w12的值

损失函数L计算时需要每一行的数据。对w0单独求偏导时，w1-w12，x，y都可以视为常数。最后的导数值是这些只有第i行时w值的平均数

用两个循环计算只有第i行时，w0-w12的梯度。（假设一共100行）

net=Network(13)

gradient\_w=np.zeros([100, 13])

for rowCount in range(100):

currentX=x[rowCount]

currentY=y[rowCount]

z=net.forward(currentX)

for i in range(13):

gradient\_w[rowCount, i]=(z-currentY)\*currentX[i]

print(gradient\_w)

而使用广播机制可以不用循环，直接矩阵乘法

### 计算梯度2

基于Numpy广播机制更快速的实现梯度计算。

当有404行数据时，L对于w1的偏导数是所有(z-y)x1的平均数。L对于b的偏导数是所有(z-y)的平均数

net=Network(13)

z=net.forward(x)

gradient\_w=(z-y)\*x

gradient\_w=np.mean(gradient\_w, axis=0)

x是404\*13的向量，y和z都是404\*1

(z-y)\*x进行元素级别的矩阵乘法，将z-y每一行的元素和x每一行所有元素分别相乘，得到一个404\*13的矩阵。这个矩阵中，每个元素表示仅有当前行的数据存在时的梯度

最后进行np.mean()将每列求平均数，总梯度是对每个样本对梯度贡献的平均值。

aaa=np.array([[1], [3], [4]])

bbb=np.array([[1, 3], [12, 2], [12, 765]])

aaa\*bbb

array([[ 1, 3],

[ 36, 6],

[ 48, 3060]])

冥冥之中暗示我应该买RTX3060

此时

print(gradient\_w.shape)

(13,)

因为使用np.mean函数时消除了第0维。需要

gradient\_w = gradient\_w[:, np.newaxis]

### 确定损失函数更小的点

eta = 0.1

net.w[5] = net.w[5] - eta \* gradient\_w5

net.w[9] = net.w[9] - eta \* gradient\_w9

z = net.forward(x)

loss = net.loss(z, y)

gradient\_w, gradient\_b = net.gradient(x, y)

gradient\_w5 = gradient\_w[5][0]

gradient\_w9 = gradient\_w[9][0]

* 定义步长eta=0.1
* net.w[5]=...更新参数w[5]。在模型设计中Network类初始化w属性为一个13\*1的向量

值得注意的是即使w[5]也不是一个标量，这里的+是矩阵的元素级运算

更新参数后相当于沿着梯度反方向移动到[w5, w9]平面的下一个点

* 重新计算z和loss，进而计算w和b的梯度
* 因为只改变w5和w9的值，所以单独提取出w5和w5的梯度
* 更新参数，继续下一轮循环
* 因为使用的全部数据训练，所以可以看到每轮循环的loss值在不断变小

前向计算->计算损失->计算梯度->更新参数

### 训练扩展到全部参数

self.w = self.w - eta \* gradient\_w

self.b = self.b - eta \* gradient\_b

由于矩阵可以进行加减运算，所以直接减掉步长\*梯度即可

### 随机梯度下降法

Stochastic Gradient Descent

实际问题中数据集往往非常大，如果每次都使用全部数据进行计算，效率非常低

一个合理的解决方案是每次从总的数据集中随机抽取出小部分数据来代表整体，基于这部分数据计算梯度和损失来更新参数

核心概念

* mini-batch：每次迭代时抽取出来的一批数据被称为一个mini-batch。
* batch\_size：一个mini-batch所包含的样本数目称为batch\_size。
* epoch：当程序迭代的时候，按mini-batch逐渐抽取出样本，当把整个数据集都遍历到了的时候，则完成了一轮训练，也叫一个epoch。启动训练时，可以将训练的轮数num\_epochs和batch\_size作为参数传入。
* 拆分数据批次

train\_data1 = train\_data[0:10]

x = train\_data1[:, :-1]

y = train\_data1[:, -1:]

loss = net.train(x, y, iterations=1, eta=0.01)

train\_data2 = train\_data[10:20]

x = train\_data2[:, :-1]

y = train\_data2[:, -1:]

loss = net.train(x, y, iterations=1, eta=0.01)

这样一次抽取10个样本作为mini-batch，循环41次更新w和b的值

但是这样会造成模型对最后出现的数据印象更加深刻。越接近模型训练结束，最后几个批次数据对模型参数的影响越大。

为了避免模型记忆影响训练效果，需要进行样本乱序操作。

* 样本乱序

# 获取数据

train\_data, test\_data = load\_data()

# 打乱样本顺序

np.random.shuffle(train\_data)

# 将train\_data分成多个mini\_batch

batch\_size = 10

n = len(train\_data)

mini\_batches = [train\_data[k:k+batch\_size] for k in range(0, n, batch\_size)]

# 创建网络

net = Network(13)

# 依次使用每个mini\_batch的数据

for mini\_batch in mini\_batches:

x = mini\_batch[:, :-1]

y = mini\_batch[:, -1:]

loss = net.train(x, y, iterations=1)

## 使用PaddlePaddle重写

import paddle

import paddle.fluid as fluid

import paddle.fluid.dygraph as dygraph

from paddle.fluid.dygraph import Linear

import numpy as np

import os

import random

paddle/fluid：飞桨的主库，目前大部分的实用函数均在paddle.fluid包内。

dygraph：动态图的类库。

Linear：神经网络的全连接层函数，即包含所有输入权重相加和激活函数的基本神经元结构。在房价预测任务中，使用只有一层的神经网络（全连接层）来实现线性回归模型。

需要numpy为1.19.9版本，最新的版本会报错

飞桨支持两种深度学习建模编写方式，更方便调试的动态图模式和性能更好并便于部署的静态图模式。

* 静态图模式（声明式编程范式，类比C++）：先编译后执行的方式。用户需预先定义完整的网络结构，再对网络结构进行编译优化后，才能执行获得计算结果。
* 动态图模式（命令式编程范式，类比Python）：解析式的执行方式。用户无需预先定义完整的网络结构，每写一行网络代码，即可同时获得计算结果。

数据处理load\_data()方法不依赖框架实现

**模型定义**的实质是定义线性回归的网络结构，飞桨建议通过创建Python类的方式完成模型网络的定义，即定义init函数和forward函数。类似于创建Network类

forward函数是框架指定实现前向计算逻辑的函数，程序在调用模型实例时会自动执行forward方法。在forward函数中使用的网络层需要在init函数中声明。

class Regressor(fluid.dygraph.Layer):

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

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

self.fc = Linear(input\_dim=13, output\_dim=1, act=None)

def forward(self, inputs):

x = self.fc(inputs)

return x

定义init函数：在类的初始化函数中声明每一层网络的实现函数。在房价预测模型中，只需要定义一层全连接层

forward函数构建神经网络结构，实现前向计算过程，并返回预测结果

Regressor继承fluid.dygraph.Layer类

self.fc定义一层全连接层，输出维度是1，激活函数为None，即不使用激活函数

fc forward calculate？

### 训练配置

* 指定运行训练的机器资源
* 声明模型实例
* 加载训练和测试数据
* 设置优化算法和学习率

with fluid.dygraph.guard():

model = Regressor()

model.train()

training\_data, test\_data = load\_data()

opt = fluid.optimizer.SGD(learning\_rate=0.01, parameter\_list=model.parameters())

获取Regressor实例

model.train()开启模型训练模式

加载数据

定义优化算法随机梯度下降-SGD，学习率设置为0.01

### 训练过程

数据准备：将一个批次的数据转变成np.array和内置格式。

前向计算：将一个批次的样本数据灌入网络中，计算输出结果。

计算损失函数：以前向计算结果和真实房价作为输入，通过损失函数square\_error\_cost计算出损失函数值（Loss）。飞桨所有的API接口都有完整的说明和使用案例，在后续的资深教程中我们会详细介绍API的查阅方法。

反向传播：执行梯度反向传播backward函数，即从后到前逐层计算每一层的梯度，并根据设置的优化算法更新参数opt.minimize。

with dygraph.guard(fluid.CPUPlace()):

EPOCH\_NUM = 10

BATCH\_SIZE = 10

for epoch\_id in range(EPOCH\_NUM):

np.random.shuffle(training\_data)

mini\_batches = [training\_data[k:k+BATCH\_SIZE] for k in range(0, len(training\_data), BATCH\_SIZE)]

for iter\_id, mini\_batch in enumerate(mini\_batches):

x = np.array(mini\_batch[:, :-1]).astype('float32')

y = np.array(mini\_batch[:, -1:]).astype('float32')

house\_features = dygraph.to\_variable(x)

prices = dygraph.to\_variable(y)

predicts = model(house\_features)

loss = fluid.layers.square\_error\_cost(predicts, label=prices)

avg\_loss = fluid.layers.mean(loss)

avg\_loss.backward()

opt.minimize(avg\_loss)

model.clear\_gradients()

fluid.save\_dygraph(model.state\_dict(), 'LR\_model')

* dygraph.to\_variable()将numpy数据转为飞桨动态图variable形式
* fluid.layers.square\_error\_cost加上mean方法计算损失
* fluid.optimizer.SGD.minimize()最小化loss,更新参数
* clear\_gradients清除梯度
* save\_dygraph()保存模型到LR\_model.pdparams文件中
* 外层循环时自定义的，每次外层循环都要训练全部数据
* [training\_data[k:k+BATCH\_SIZE] for k in range(0, len(training\_data), BATCH\_SIZE)]

生成一个list，list有len(training\_data)/BATCH\_SIZE个矩阵，每个矩阵就是内层循环中用到的数据。每个矩阵有BATCH\_SIZE行，对应BATCH\_SIZE行数据

### 测试模型

下面我们选择一条数据样本，测试下模型的预测效果。测试过程和在应用场景中使用模型的过程一致，主要可分成如下三个步骤：

配置模型预测的机器资源。本案例默认使用本机，因此无需写代码指定。

将训练好的模型参数加载到模型实例中。由两个语句完成，第一句是从文件中读取模型参数；第二句是将参数内容加载到模型。加载完毕后，需要将模型的状态调整为eval()（校验）。上文中提到，训练状态的模型需要同时支持前向计算和反向传导梯度，模型的实现较为臃肿，而校验和预测状态的模型只需要支持前向计算，模型的实现更加简单，性能更好。

将待预测的样本特征输入到模型中，打印输出的预测结果。

通过load\_one\_example函数实现从数据集中抽一条样本作为测试样本，具体实现代码如下所示。

In [7]

def load\_one\_example(data\_dir):

f = open(data\_dir, 'r')

datas = f.readlines()

# 选择倒数第10条数据用于测试

tmp = datas[-10]

tmp = tmp.strip().split()

one\_data = [float(v) for v in tmp]

# 对数据进行归一化处理

for i in range(len(one\_data)-1):

one\_data[i] = (one\_data[i] - avg\_values[i]) / (max\_values[i] - min\_values[i])

data = np.reshape(np.array(one\_data[:-1]), [1, -1]).astype(np.float32)

label = one\_data[-1]

return data, label

In [8]

with dygraph.guard():

# 参数为保存模型参数的文件地址

model\_dict, \_ = fluid.load\_dygraph('LR\_model')

model.load\_dict(model\_dict)

model.eval()

# 参数为数据集的文件地址

test\_data, label = load\_one\_example('./work/housing.data')

# 将数据转为动态图的variable格式

test\_data = dygraph.to\_variable(test\_data)

results = model(test\_data)

# 对结果做反归一化处理

results = results \* (max\_values[-1] - min\_values[-1]) + avg\_values[-1]

print("Inference result is {}, the corresponding label is {}".format(results.numpy(), label))

# Numpy

# Deep Learning