神经网络实现波士顿房价预测

神经网络算法基本步骤

- 数据处理：从本地读取数据，并进行数据预处理，并对数据进行一定的变形，保证数据可处理性
- 模型设计：网络结构设计，设计模型设计的处理关系的集合
- 训练配置：设定训练求解的算法，并指定计算资源
- 训练过程：循环调用训练过程，每轮都包括前向计算，损失函数（优化的目标）和后向传播三个步骤
- 测试数据：将训练好的模型留存，在模型预测是调用

数据处理

数据处理主要包括：数据导入，数据变形，训练测试数据集划分，数据归一化。将数据处理这四个步骤封装成一个load_data函数

数据导入+变形

数据存放在同级目录中的"housing.data"文件中，使用numpy函数的fromfile方法将文件中的数据录入到data数组变量中，并将读入的数据重塑成记录数*因素数的二维数组形式

```
datafile = '波士顿房价预测(神经网络)\\housing.data'
data = np.fromfile(datafile, sep=' ')
# 十三个因素的名称和房价，表明每一列表示的是什么
feature_names = [
    'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX',
    'PTRATIO', 'B', 'LSTAT', 'MEDV'
]
feature_nums = len(feature_names)
# 将一维的数组变为506*14的二维数组
data = data.reshape([data.shape[0] // feature_nums, feature_nums])
```

数据划分

总的数据集划分成4：1 前80%的数据作为训练数据集，后20%作为测试数据集

```
# 将数据集划分为训练集和测试集
# 数据集中的前80%作为训练集，后20%作为测试集
ratio = 0.8
offset = int(data.shape[0] * ratio)
training_data = data[:offset]  # 训练集
test_data = data[offset:]  # 测试集
```

数据归一化

对每个特征值进行归一化处理，将所有的数据缩放到0-1之间。这样会使得模型训练更加高效，又因为所有的特征值都在一个范围里面，所以之后特征值的权值大小可以代表该特征值对房价结果的贡献度

```
maximums = training_data.max(axis=0)
minimums = training_data.min(axis=0)
# 数据归一化 将所有的数据归到（0，1）中
for i in range(feature_nums):
    data[:, i] = (data[:, i] - minimums[i]) / (maximums[i] - minimums[i])
return training_data, test_data
```

数据处理函数封装

将上述操作封装在load_data()函数中，可以方便在主函数中调用，其中返回的是的训练数据集数据和测试数据集

模型设计

简单的来说，神经网络是一个输入，计算，再输出的过程。在计算时，需要有输入的权值，然后再用激活函数来转化某个输出的结果

预测值 = Xi * X对应的权值 + 偏转量。该院算即为前向计算

```
# 运用numpy的dot函数将权值和X进行矩阵相乘
t = np.dot(x,w)
# 添加偏转量
z = t + b
```

将上述网络封装成一个类，类成员中含有对应维度的权值和偏转量，另外用forward函数表示前向计算，权值是用随机数产生。

```
class Network(object):
    def __init__(self, num_of_weights):
        # 随机产生13个w值
        # 为保证每次结果都不一样，设置固定的随机种子
        np.random.seed(0)
        self.w = np.random.randn(num_of_weights, 1)
        self.b = 0.

    # 前向计算：得到的随机数作为特征值计算得到输出值的过程称为“前向计算”
    def forward(self, x):
        z = np.dot(x, self.w) + self.b
        return z
```

房价问题因为没有涉及多项输出的几率问题，所以直接用线形函数 y = x作为激活函数。

训练配置

基础模型设计完成后，需要在这个模型中寻找最优解，即通过损失函数来衡量模型的好坏，

通过模型计算得到的预测值为z，实际的数据为y。用均方差作为评价模型好坏的指标：

$$Loss = (y - z)^2$$

由于计算损失需要把所有的样本考虑进去，所以需要对样本的方差进行求和，再取平均。

$$Loss = sum((y-z)^2)/N$$

损失函数添加在Network类中，作为网络的一个方法

```
# 计算得到 预测值和实际值的损失值
def loss(self, z, y):
    error = z - y
    cost = error * error
    cost = np.mean(cost)
    return cost
```

训练过程

以上部分为构建神经网络，并计算预测值和真实值的误差，在训练过程中会不断地更新权值和偏转量，直到Loss函数尽可能小，即寻找组w和b使得Loss取到极小值。

此处运用梯度下降和反向传播法计算得到权值和偏转量

梯度下降法

要计算极小值，预测值应当慢慢的完数据曲线地谷下降，也就是沿着梯度反向方向改变，将权值和偏转量根据一定的更新率向各自变量地梯度下降方向递减，知道损失值几乎不再下降。如此可以找到当前范围内地局部极小值。

使用梯度下降法，首先需要计算梯度

计算梯度

为了计算方便在Loss函数中引入1/2，在之后的求导中可以去除系数，会更加方便一点。

$$Loss = sum((yi-zi)^2)/2N$$

其中第i个样本值为

$$Zi = sum(wi*xi)+b$$

求梯度，需要对Loss对每个权值和偏转量求偏导

$$gradient = (\frac{\partial L}{\partial w_0}, \frac{\partial L}{\partial w_1}, ..., \frac{\partial L}{\partial w_{12}}, \frac{\partial L}{\partial b})$$

$$\partial L/(\partial w_j) = \partial L/\partial z \partial z/(\partial w_j)$$

根据上面的链式法则可以计算L对w和b的偏导数

$$\partial L/(\partial w_j) = \partial L/\partial z \partial z/(\partial w_j) = sum(z(i)-y(i))*x(ij)$$

$$\partial L/(\partial w_j) = \partial L/\partial z \partial z/(\partial w_j) = sum(z(i)-y(i))*x(ij)$$

```
# 每一个w的偏导，x1为训练数据
z1 = net.forward(x1)
gradient_wx = (z1 - y1) * x1[x]
```

根据矩阵的特点，可以使用numpy一次性计算所有的权值的梯度，并把它封装成类方法

```
def gradient(self, x, y):
    # 前向计算得到新的z
    z = self.forward(x)
    # 使用numpy矩阵计算每一个w的梯度，取每个w权值的平均值，并将其变为列矩阵
    gradient_w = (z - y) * x
    gradient_w = np.mean(gradient_w, axis=0)
    gradient_w = gradient_w[:, np.newaxis]
    # 使用numpy计算y的梯度
    gradient_b = (z - y)
    gradient_b = np.mean(gradient_b, axis=0)
    return gradient_w, gradient_b
```

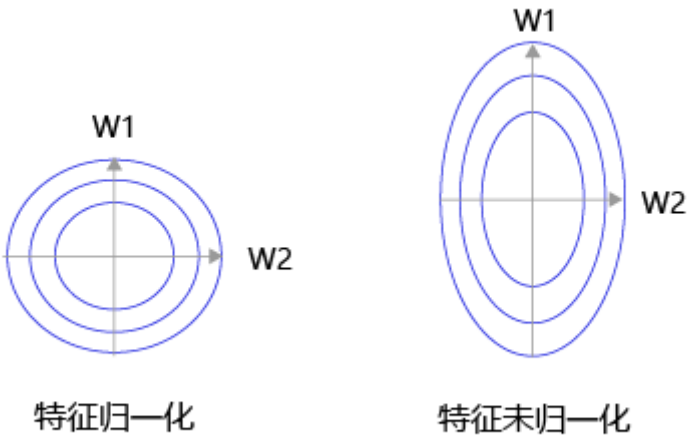
更新特征参数

更新权值：现在的权值根据一定的更新率向梯度相反的方向更新，这个需要在每次训练循环中执行

- 相减:参数需要向梯度的反方向移动
- eta:控制每次参数值沿着梯度反方向变动的大小,即每次移动的步长,又称为学习率

```
# 沿着梯度的反方向移动步长
def update(self, gradient_w, gradient_b, eta=0.01):
    self.w = self.w - gradient_w * eta
    self.b = self.b - gradient_b * eta
```

此时特征值归一化就有了意义，如果没有归一化，那么不同的特征值的参数的长度不一样尺度较大的需要较长的步长，但是尺度较小的需要较小的步长，导致不能使用相同的学习率



封装Train过程

将梯度下降法的训练过程封装成train

```
def train(self, x, y, iterations=100, eta=0.01):
        points = []
        losses = []
        for i in range(iterations):
            z = self.forward(x)
            L = self.loss(z, y)
            gradient_w, gradient_b = self.gradient(x, y)
            self.update(gradient_w, gradient_b, eta)
            losses.append(L)
            if i % 50 == 0:
                print('iter {}, loss {}'.format(i, L))
        return points, losses
```

随机梯度下降法

以上训练过程是针对全体训练数据的，在波士顿房价预测中只有400个样本，全部遍历在执行时间中还是可以接受的，但是实际情况下的数据集是有非常大的数据集的，如果每次训练时都遍历所有训练集，效率会非常低，所以采用了随机梯度下降法。

随机梯度下降法的主要思想：由于更新参数只沿着梯度下降方向减少一点点，因此方向并不需要那么精确；在每次迭代的时候只要从所有数据集中随机抽取一小部分作为整体，基于这部分数据来计算权值和偏转量的梯度值，按照这个梯度进行参数更新

- 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。

对原代码进行修改

将每个随机抽取的mini-batch数据输入到模型中用于参数训练。训练过程的核心是两层循环：

1. 第一层循环，代表样本集合要被训练遍历几次，称为“epoch”，代码如下：

```
for epoch_id in range(num_epoches):
```

1. 第二层循环，代表每次遍历时，样本集合被拆分成的多个批次，需要全部执行训练，称为“iter (iteration)”，代码如下：

```
for iter_id, batch in emumerate(mini_batches):
```

在两层循环的内部是经典的四步训练流程：前向计算->计算损失->计算梯度->更新参数

```
# eta为步长，或者可以被称为学习率，num_epoches为被训练的次数，batch_size为小组的个数
def train(self, training_data, num_epoches=100, batch_size=5, eta=0.01):
    n = len(training_data)
    losses = []
    for epoch_id in range(num_epoches):
        # 每次循环都需要把数据打乱一次
        np.random.shuffle(training_data)
        mini_batch = [training_data[k:k + batch_size] for k in range(0, n,
batch_size)]
        for iter_id, batch in enumerate(mini_batch):
            x = batch[:, :-1]
            y = batch[:, -1:]
```

```
            a = self.forward(x)
            loss = self.loss(a, y)
            gradient_w, gradient_b = self.gradient(x, y)
            self.update(gradient_w, gradient_b, eta)
            losses.append(loss)
            print('Epoch {:3d} / iter {:3d}, loss = {:.4f}'.format(
                epoch_id, iter_id, loss))
    return losses
```

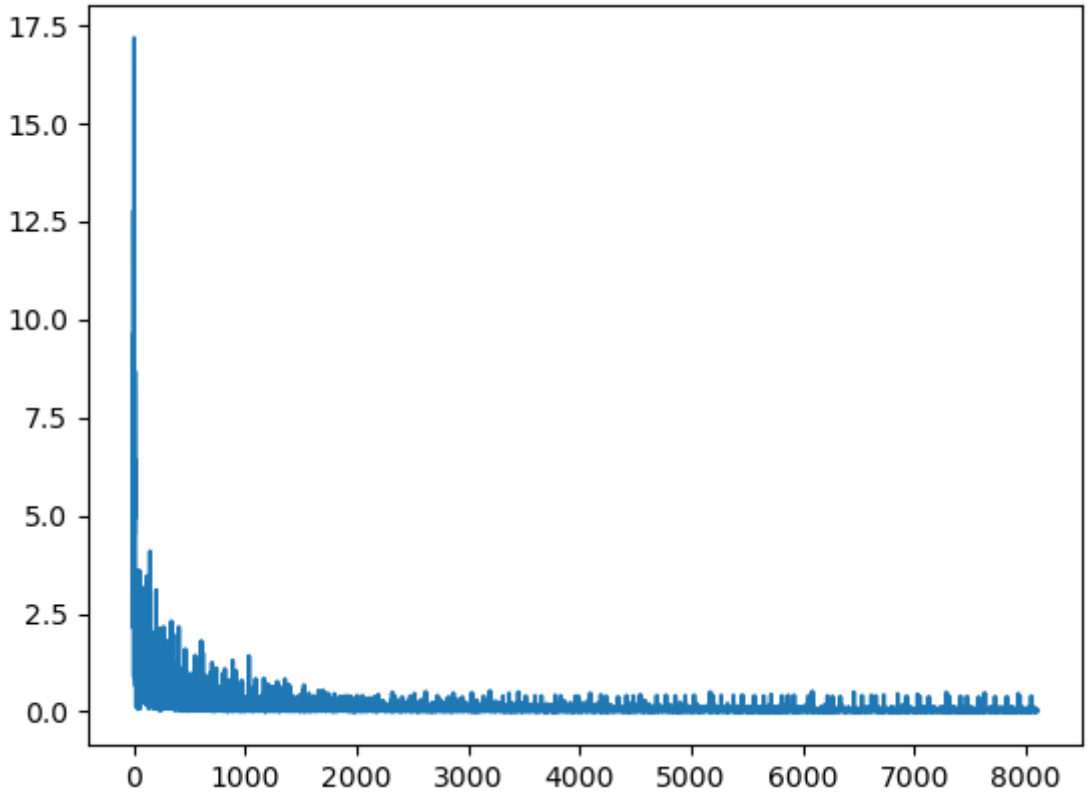
模型测试

测试主函数代码

```
# 获取数据 x为前13个因素 y为最后的房价
training_data, test_data = LoadData()
x = training_data[:, :-1]
y = training_data[:, -1:]

# 初始化网络
net = Network(13)
# 开始训练
losses = net.train(training_data, num_epoches=100, batch_size=10, eta=0.01)
plot_x = np.arange(len(losses))
plot_y = np.array(losses)
# 绘画残值的迭代减小过程
plt.plot(plot_x, plot_y)
plt.show()
trained_w = net.w
test_x = test_data[:, :-1]
test_y = test_data[:, -1:]
result_y = np.dot(test_x, trained_w) + net.b
# 绘制测试集的实际数据和预测值的拟合情况
plt.plot(np.arange(len(test_y)), test_y, 'g--')
plt.plot(np.arange(len(test_y)), result_y)
plt.show()
```

损失函数的波动减小



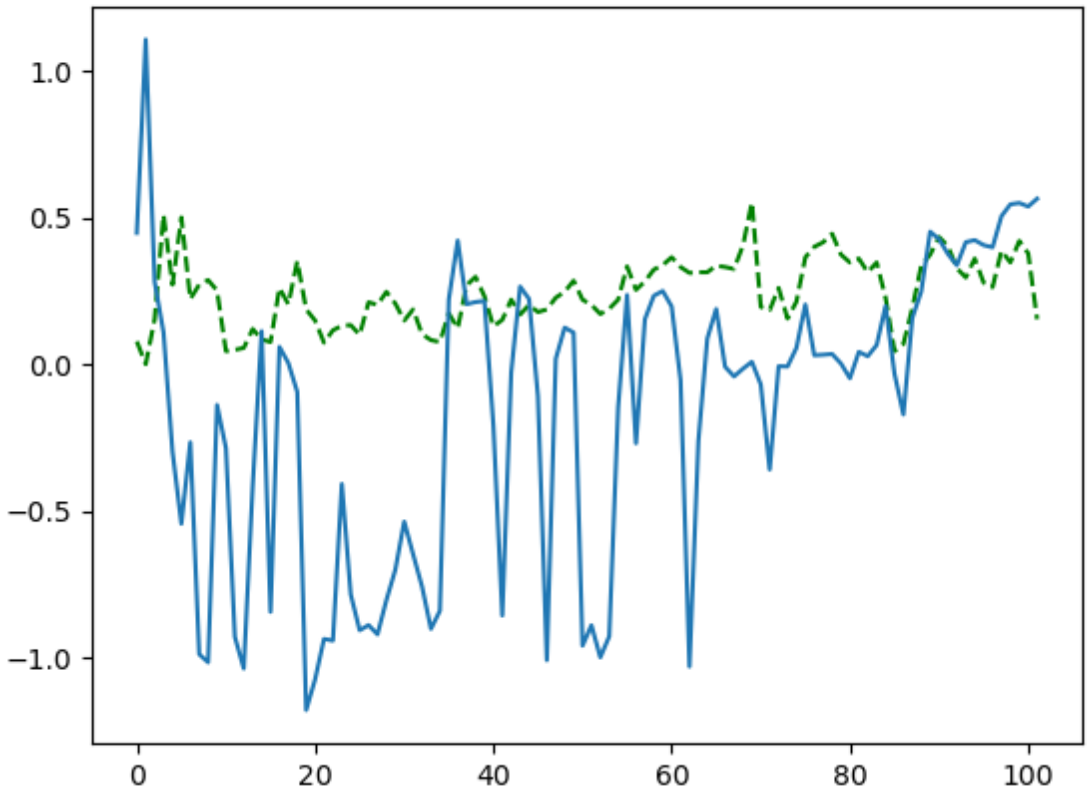
由于使用的时随机梯度下降法，每个批次都是随机的，在又些点会爆发一些数据损失

测试数据的拟合

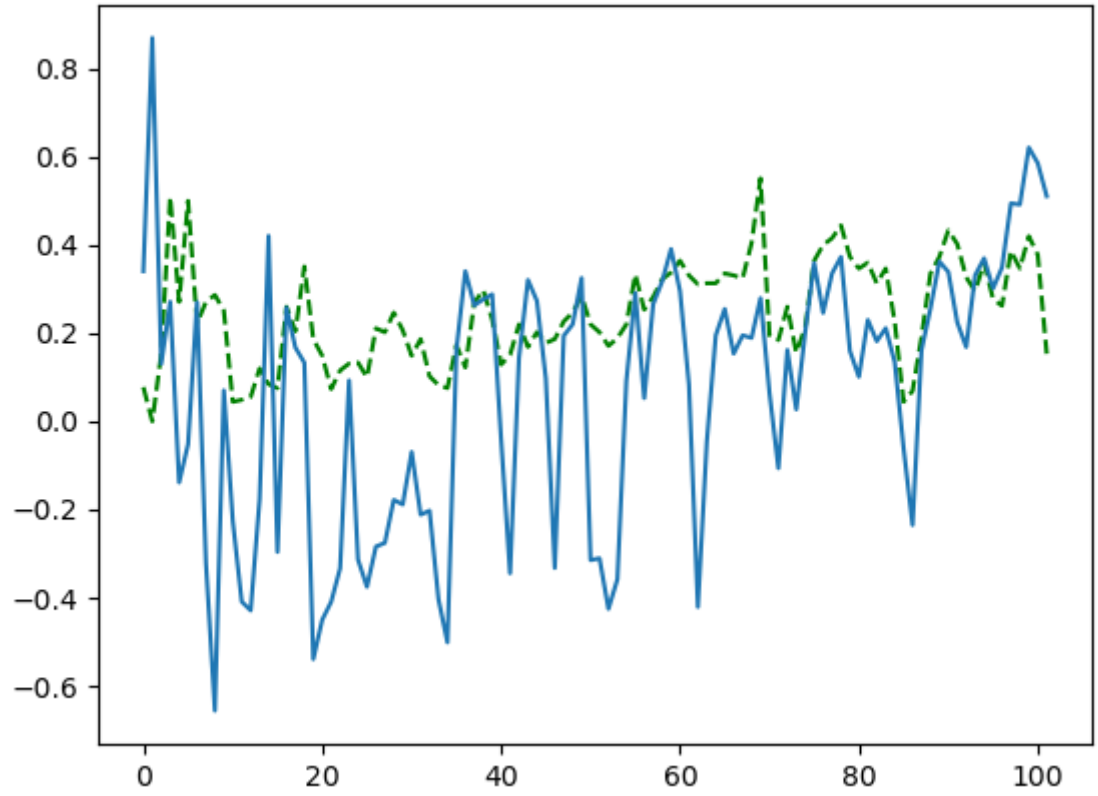
经过对参数的调整，发现在随机梯度下降法中，减小每个小批次的个数，会提高预测结果的准确度。

但是如果减少每个小批次的个数，效率会明显比批次个数比较大的效率低。

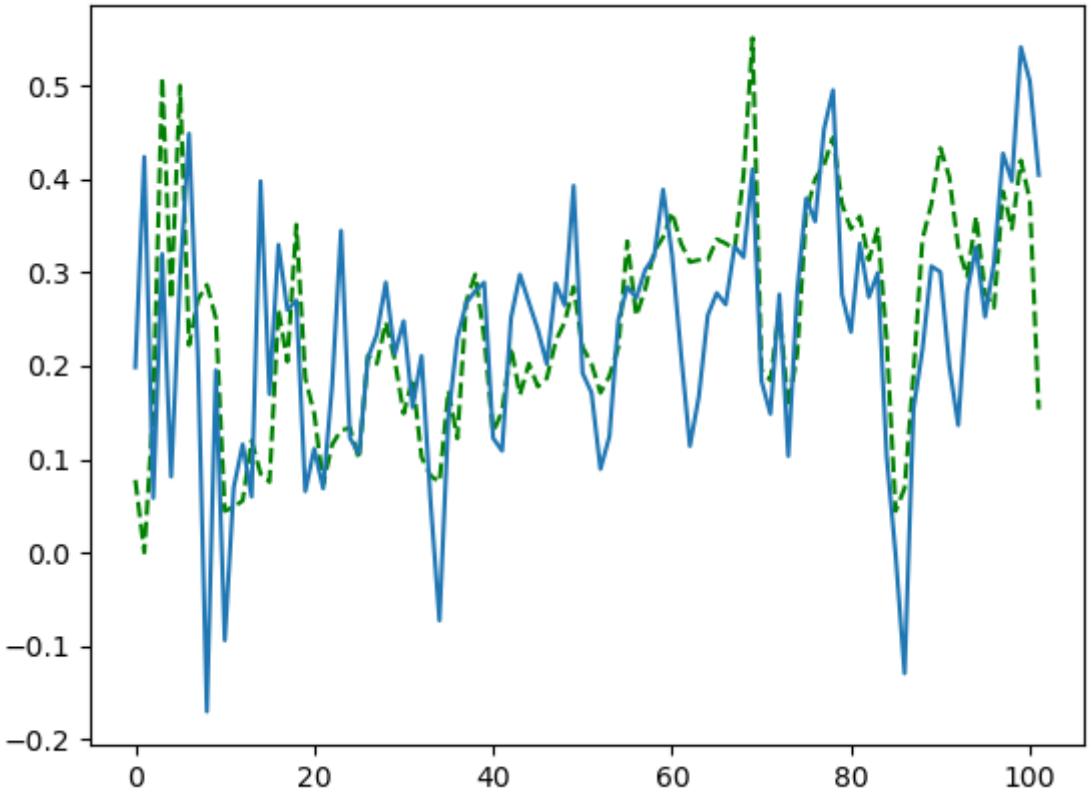
当batch_size = 10时：



当batch_size = 3 时：



当batch_size = 1 时：其实和没有使用随机梯度下降法的道理一样，虽然拟合的比较好，但是比上面的低



参考资料

博客:

- https://blog.csdn.net/im6520/article/details/108065804
- https://blog.csdn.net/kun_csdn/article/details/88853907