深度学习

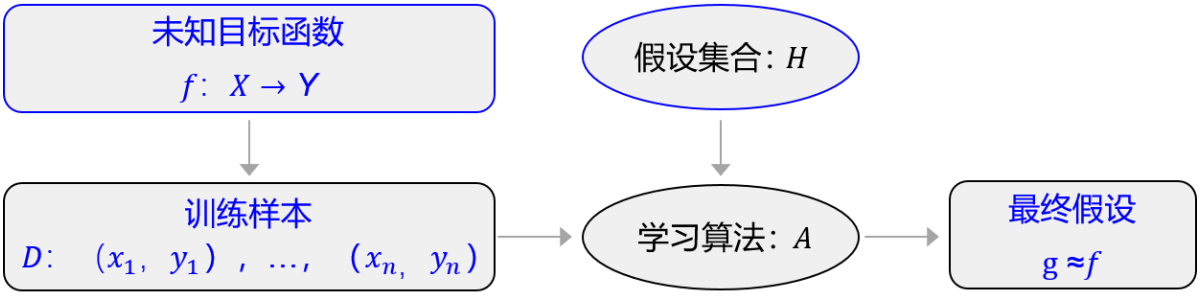
基础知识

模型有效的基本条件是能够拟合已知的样本

衡量模型预测值和真实值差距的**评价函数**也被称为**损失函数**（Loss）

机器通过尝试答对（最小化损失）大量的习题（已知样本）来学习知识（模型参数ww），并期望用学习到的知识所代表的模型H(w,x)*H*(*w*,*x*)，回答不知道答案的考试题（未知样本）。最小化损失是模型的优化目标，实现损失最小化的方法称为**优化算法**，也称为寻解算法（**找到使得损失函数最小的参数解**）。参数w和输入x组成公式的基本结构称为**假设**。在牛顿第二定律的案例中，基于对数据的观测，我们提出了线性假设，即作用力和加速度是线性关系，用线性方程表示。由此可见，**模型假设、评价函数（损失/优化目标）和优化算法是构成模型的三个关键要素。**

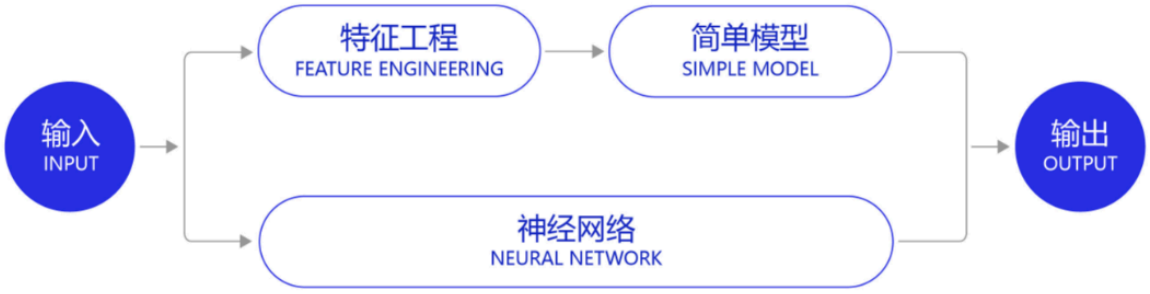
机器执行学习任务的框架体现了其**学习的本质是"参数估计"**



许多看起来完全不一样的问题都可以使用同样的框架进行学习，它们的学习目标都是拟合一个"大公式f"

深度学习

DL vs ML：两者在理论结构上是一致的，即：模型假设、评价函数和优化算法，其根本差别在于假设的复杂度。

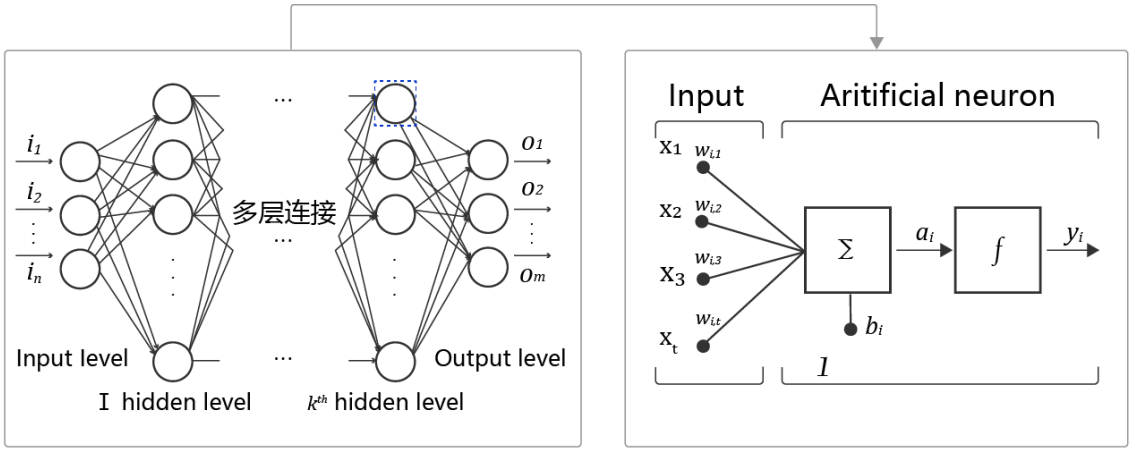


研究者们借鉴了人脑神经元的结构，设计出**神经网络**的模型

神经网络

人工神经网络包括多个神经网络层，如：**卷积层、全连接层、LSTM等**，**每一层又包括很多神经元**，**超过三层的非线性神经网络**都可以被称为**深度神经网络**。通俗的讲，深度学习的模型可以视为是输入到输出的映射函数

足够深的神经网络理论上可以拟合任何复杂的函数。因此神经网络非常适合学习样本数据的内在规律和表示层次



- **神经元**：神经网络中每个节点称为神经元，由两部分组成：
  - 加权和：将所有输入加权求和。
  - 非线性变换（激活函数）：加权和的结果经过一个非线性函数变换，让神经元计算具备非线性的能力。
- **多层连接**：大量这样的节点按照不同的层次排布，形成多层的结构连接起来，即称为神经网络。
- **前向计算**：从输入计算输出的过程，顺序从网络前至后。
- **计算图**：以图形化的方式展现神经网络的计算逻辑又称为计算图，也可以将神经网络的计算图以公式的方式表达：

$$Y = f_3(f_2(f_1(w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b) + \ldots) \ldots) \ldots)$$

Boston房价预测

模型选择：假设房价和各影响因素之间能够用线性关系来描述

$$y = \sum_{j=1}^{M} x_j w_j + b$$

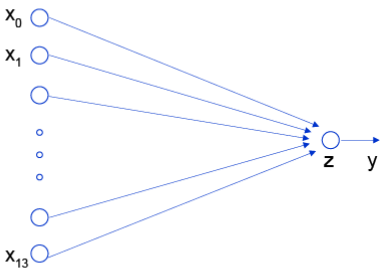
模型的求解即是通过数据拟合出每个wj和b。其中，wj和b分别表示该线性模型的权重和偏置。

损失函数：线性回归模型使用均方误差作为（Mean Squared Error，MSE）损失函数（Loss）

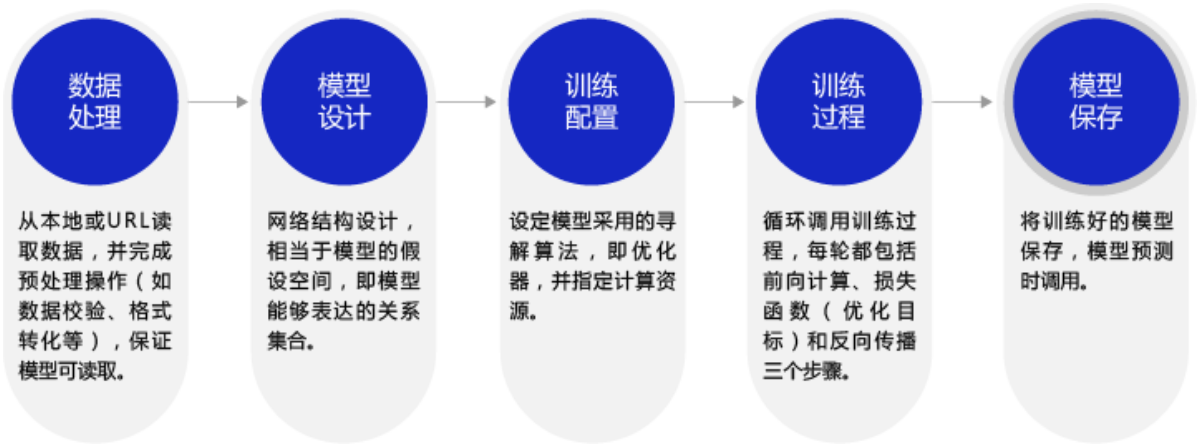
$$MSE = \frac{1}{N} \sum_{i=1}^{N} (\hat{Y}_i - Y_i)^2$$

线性回归模型的神经网络结构

神经网络的标准结构中每个神经元由加权和与非线性变换构成，然后将多个神经元分层的摆放并连接形成神经网络。线性回归模型可以认为是神经网络模型的一种极简特例，是一个只有加权和、没有非线性变换的神经元（无需形成网络）



构建神经网络/深度学习模型的基本步骤



数据处理

数据处理包含五个部分：**数据导入**、**数据形状变换**、**数据集划分**、**数据归一化处理**和**封装`load data`函数**。数据预处理后，才能被模型调用。

数据读取

通过如下代码读入数据，数据存放在本地目录下housing.data文件中。

```
# 导入需要用到的package
import numpy as np
import json
# 读入训练数据
datafile = './work/housing.data'
data = np.fromfile(datafile, sep=' ')
data
```

数据形状变换

由于读入的原始数据是1维的，所有数据都连在一起。因此需要我们将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）。

```
# 这里对原始数据做reshape，变成N x 14的形式
feature_names = [ 'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE','DIS', 'RAD',
'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV' ]
feature_num = len(feature_names)
# 将原始数据进行Reshape，变成[N, 14]这样的形状
data = data.reshape([data.shape[0] // feature_num, feature_num])
```

数据集划分

将数据集划分成训练集和测试集，其中训练集用于确定模型的参数，测试集用于评判模型的效果。

在本案例中，我们将80%的数据用作训练集，20%用作测试集，实现代码如下。

```
ratio = 0.8
offset = int(data.shape[0] * ratio)
training_data = data[:offset]
```

数据归一化处理

对每个特征进行归一化处理，使得每个特征的取值缩放到0~1之间。这样做有两个好处：

- 一：模型训练更高效；
- 二：特征前的权重大小可以代表该变量对预测结果的贡献度（因为每个特征值本身的范围相同）。

```
# 计算train数据集的最大值，最小值
maximums, minimums = training_data.max(axis=0), \
                     training_data.min(axis=0),
# 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])
```

封装成load data函数

```
def load_data():
    # 从文件导入数据
    datafile = './work/housing.data'
    data = np.fromfile(datafile, sep=' ')

    # 每条数据包括14项，其中前面13项是影响因素，第14项是相应的房屋价格中位数
    feature_names = [ 'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', \
                      'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV' ]
    feature_num = len(feature_names)

    # 将原始数据进行Reshape，变成[N, 14]这样的形状
    data = data.reshape([data.shape[0] // feature_num, feature_num])

    # 将原数据集拆分成训练集和测试集
    # 这里使用80%的数据做训练，20%的数据做测试
    # 测试集和训练集必须是没有交集的
    ratio = 0.8
    offset = int(data.shape[0] * ratio)
    training_data = data[:offset]

    # 计算训练集的最大值，最小值
    maximums, minimums = training_data.max(axis=0), \
                         training_data.min(axis=0)

    # 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])

    # 训练集和测试集的划分比例
```

```
    training_data = data[:offset]
    test_data = data[offset:]
    return training_data, test_data
```

获取数据

```
# 获取数据
training_data, test_data = load_data()
x = training_data[:, :-1]
y = training_data[:, -1:]
```

模型设计

模型设计是深度学习模型关键要素之一，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。

通过写一个`forward`函数（代表"前向计算"）完成上述从特征和参数到输出预测值的计算过程，代码实现如下。

```
class Network(object):
    def __init__(self, num_of_weights):
        # 随机产生w的初始值
        # 为了保持程序每次运行结果的一致性，
        # 此处设置固定的随机数种子
        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
```

训练配置

模型设计完成后，需要通过训练配置寻找模型的最优值，即通过损失函数来衡量模型的好坏。训练配置也是深度学习模型关键要素之一。

我们需要有某种指标来衡量预测值z跟真实值y之间的差距。对于回归问题，最常采用的衡量方法是使用均方误差作为评价模型好坏的指标，公式为

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

上式中的Loss通常也被称作**损失函数**，它是衡量模型好坏的指标。

在回归问题中常用均方误差作为损失函数，而在分类问题中常用采用交叉熵（Cross-Entropy）作为损失函数

因为计算损失函数时需要把每个样本的损失函数值都考虑到，所以我们需要对单个样本的损失函数进行求和，并除以样本总数N。公式为

$$Loss = \frac{1}{N}\sum_{i=1}^{N}(y_i - z_i)^2$$

在Network类下面添加损失函数的代码实现如下

```
class Network(object):
```

```
    def __init__(self, num_of_weights):
        # 随机产生w的初始值
        # 为了保持程序每次运行结果的一致性，此处设置固定的随机数种子
        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

    def loss(self, z, y):
        error = z - y
        cost = error * error
        cost = np.mean(cost)
        return cost
```

训练过程

这个过程也称为模型训练过程。训练过程是深度学习模型的关键要素之一，其目标是让定义的损失函数尽可能的小，也就是说找到一个参数解w和b，使得损失函数取得极小值。

梯度下降法

在现实中存在大量的函数正向求解容易，但反向求解较难，被称为单向函数

神经网络模型的损失函数就是这样的单向函数，反向求解并不容易

求解Loss函数最小值可以这样实现：从当前的参数取值，一步步的按照下坡的方向下降，**直到走到最低点**。这种方法笔者称它为“盲人下坡法”。哦不，有个更正式的说法“**梯度下降法**（Gradient Descent，GD）”。

训练的关键是找到一组(*w*,*b*)，使得损失函数*L*取极小值。

梯度计算

对 *L* 进行简化

$$L = \frac{1}{2}(y_i - z_i)^2$$

$$z_1 = x_1^0 \cdot w_0 + x_1^1 \cdot w_1 + ... + x_1^{12} \cdot w_{12} + b$$

可以计算出*L*对*w*和*b*的偏导数：

$$\frac{\partial L}{\partial w_0} = (x_1^0 \cdot w_0 + x_1^1 \cdot w_1 + ... + x_1^{12} \cdot w_1 2 + b - y_1) \cdot x_1^0 = (z_1 - y_1) \cdot x_1^0$$

$$\frac{\partial L}{\partial b} = (x_1^0 \cdot w_0 + x_1^1 \cdot w_1 + ... + x_1^{12} \cdot w_{12} + b - y_1) \cdot 1 = (z_1 - y_1)$$

使用`Numpy`进行梯度计算

基于`Numpy`广播机制（对向量和矩阵计算如同对 1个单一变量计算一样），可以更快速的实现梯度计算。使用Numpy的矩阵操作来简化运算

那么对于有N个样本的情形，我们可以直接使用如下方式计算出 ***所有*** **样本对梯度的贡献**

使用Numpy库的广播功能：

- 一方面可以扩展参数的维度，代替for循环来计算1个样本对从*w*0到*w*12的所有参数的梯度。
- 另一方面可以扩展样本的维度，代替for循环来计算样本0到样本403对参数的梯度。

```
z = net.forward(x)
gradient_w = (z - y) * x
```

测验结果

```
print('gradient_w shape {}'.format(gradient_w.shape))
print(gradient_w)
**********************************************************
gradient_w shape (404, 13)
[[0.00000000e+00 2.34805180e+01 9.58029163e+00 ... 3.74689117e+01 1.30447322e+02
1.16985043e+01]
 [2.54738434e-02 0.00000000e+00 2.83333765e+01 ... 5.97311025e+01 1.07975454e+02
2.20777626e+01]
 [3.07963708e-02 0.00000000e+00 3.42860463e+01 ... 7.22802431e+01 1.29029688e+02
8.29246719e+00]
 ...
 [3.97706874e+01 0.00000000e+00 1.74130673e+02 ... 2.01043762e+02 2.48659390e+02
1.27554582e+02]
 [2.69696515e+01 0.00000000e+00 1.75225687e+02 ... 2.02308019e+02 2.34270491e+02
1.28287658e+02]
 [6.08972123e+01 0.00000000e+00 1.53017134e+02 ... 1.76666981e+02 2.18509161e+02
1.08772220e+02]]
```

上面gradient_w的每一行代表了一个样本对梯度的贡献。根据梯度的计算公式，总梯度是对每个样本对梯度贡献的平均值。

$$\frac{\partial L}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(z_i - y_i)\frac{\partial z_i}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(z_i - y_i)x_i^j$$

可以使用Numpy的均值函数来完成此过程

```
# axis = 0 表示把每一行做相加然后再除以总的行数
gradient_w = np.mean(gradient_w, axis=0)
```

使用Numpy的矩阵操作方便地完成了gradient的计算，但引入了一个问题，`gradient_w`的形状是(13,)，而*w*的维度是(13, 1)。导致该问题的原因是使用`np.mean`函数时消除了第0维。为了加减乘除等计算方便，`gradient_w`和*w*必须保持一致的形状。因此我们将`gradient_w`的维度也设置为(13,1)，代码如下

```
gradient_w = gradient_w[:, np.newaxis]
```

将计算*w*和*b*的梯度的过程，写成Network类的`gradient`函数，实现方法如下所示

```
class Network(object):
    ......
    def gradient(self, x, y):
        z = self.forward(x)
        gradient_w = (z-y)*x
        gradient_w = np.mean(gradient_w, axis=0)
        gradient_w = gradient_w[:, np.newaxis]
        gradient_b = (z - y)
        gradient_b = np.mean(gradient_b)

        return gradient_w, gradient_b
```

确定损失函数更小的点

下面我们开始研究更新梯度的方法。首先沿着梯度的反方向移动一小步，找到下一个点P1，观察损失函数的变化

这里考虑w5，w9其他参数固定

```
# 在[w5, w9]平面上，沿着梯度的反方向移动到下一个点P1
# 定义移动步长 eta
eta = 0.1
# 更新参数w5和w9
net.w[5] = net.w[5] - eta * gradient_w5
net.w[9] = net.w[9] - eta * gradient_w9
# 重新计算z和loss
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]
```

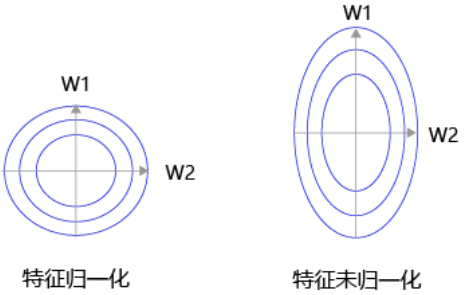
运行上面的代码，可以发现沿着梯度反方向走一小步，下一个点的损失函数的确减少了。

在上述代码中，每次更新参数使用的语句：

```
net.w[5] = net.w[5] - eta * gradient_w5
```

- 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**。

之前我们要做输入特征的归一化，保持尺度一致，这是为了让统一的步长更加合适，未归一化的特征，会导致不同特征维度的理想步长不同



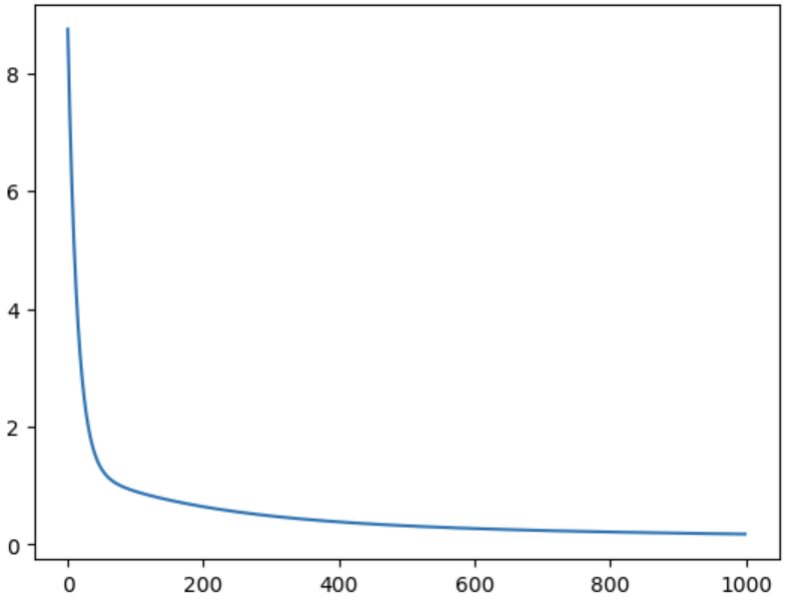
训练扩展到全部参数

实现逻辑：“前向计算输出、根据输出和真实值计算Loss、基于Loss和输入计算梯度、根据梯度更新参数值”四个部分反复执行，直到到损失函数最小。具体代码如下所示。

```
class Network(object):
    def __init__(self, num_of_weights):
        # 随机产生w的初始值
        # 为了保持程序每次运行结果的一致性，此处设置固定的随机数种子
        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

    def loss(self, z, y):
        error = z - y
        num_samples = error.shape[0]
        cost = error * error
        cost = np.sum(cost) / num_samples
        return cost

    def gradient(self, x, y):
        z = self.forward(x)
        gradient_w = (z-y)*x
        gradient_w = np.mean(gradient_w, axis=0)
        gradient_w = gradient_w[:, np.newaxis]
        gradient_b = (z - y)
        gradient_b = np.mean(gradient_b)
        return gradient_w, gradient_b

    def update(self, gradient_w, gradient_b, eta = 0.01):
        self.w = self.w - eta * gradient_w
        self.b = self.b - eta * gradient_b

    def train(self, x, y, iterations=100, eta=0.01):
        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+1) % 10 == 0:
                print('iter {}, loss {}'.format(i, L))
        return losses

# 获取数据
train_data, test_data = load_data()
x = train_data[:, :-1]
y = train_data[:, -1:]
# 创建网络
net = Network(13)
```

```
num_iterations=1000
# 启动训练
losses = net.train(x,y, iterations=num_iterations, eta=0.01)
```

```
# 画出损失函数的变化趋势
plot_x = np.arange(num_iterations)
plot_y = np.array(losses)
plt.plot(plot_x, plot_y)
plt.show()
```



随机梯度下降法（Stochastic Gradient Descent）

在实际问题中，数据集往往非常大，如果每次都使用全量数据进行计算，效率非常低。由于参数每次只沿着梯度反方向更新一点点，因此方向并不需要那么精确。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

这种方法被称作**随机梯度下降法**（Stochastic Gradient Descent，SGD），核心概念如下：

- minibatch：**每次迭代时抽取**出来的一批数据被称为一个minibatch。
- batch size：**每个minibatch所包含的样本数目**称为batch size。
- Epoch：当程序迭代的时候，按minibatch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（轮次）**。启动训练时，可以将训练的轮数`num_epochs`和`batch_size`作为参数传入。

train_data中一共包含404条数据，如果batch_size=10，即取前0-9号样本作为第一个minibatch，命名train_data1。

使用train_data1的数据（0-9号样本）计算梯度并更新网络参数。

再取出10-19号样本作为第二个minibatch，计算梯度并更新网络参数。

按此方法**不断的取出新的minibatch**，并**逐渐更新网络参数。**

通过大量实验发现，模型对最后出现的数据印象更加深刻。训练数据导入后，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。**随机打乱样本顺序**，需要用到`np.random.shuffle`函数

训练过程代码修改：训练过程的核心是两层循环

- 第一层循环，代表样本集合要被训练遍历几次，称为“epoch”

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

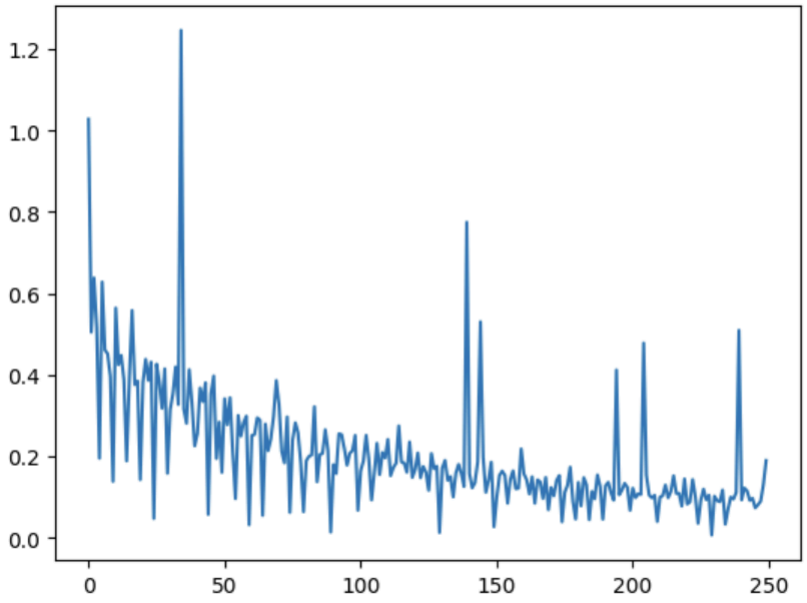
- 第二层循环，代表每次遍历时，样本集合被拆分成的多个批次，需要全部执行训练，称为“iter (iteration)”

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

将这部分实现SGD算法的代码集成到Network类中的`train`函数中，最终的完整代码如下。

```
import numpy as np

class Network(object):
    def __init__(self, num_of_weights):
        # 随机产生w的初始值
        # 为了保持程序每次运行结果的一致性，此处设置固定的随机数种子
        #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

    def loss(self, z, y):
        error = z - y
        num_samples = error.shape[0]
        cost = error * error
        cost = np.sum(cost) / num_samples
        return cost

    def gradient(self, x, y):
        z = self.forward(x)
        N = x.shape[0]
        gradient_w = 1. / N * np.sum((z-y) * x, axis=0)
        gradient_w = gradient_w[:, np.newaxis]
        gradient_b = 1. / N * np.sum(z-y)
        return gradient_w, gradient_b

    def update(self, gradient_w, gradient_b, eta = 0.01):
        self.w = self.w - eta * gradient_w
        self.b = self.b - eta * gradient_b


    def train(self, training_data, num_epochs, batch_size=10, eta=0.01):
        n = len(training_data)
        losses = []
        for epoch_id in range(num_epochs):
            # 在每轮迭代开始之前，将训练数据的顺序随机打乱
            # 然后再按每次取batch_size条数据的方式取出
            np.random.shuffle(training_data)
            # 将训练数据进行拆分，每个mini_batch包含batch_size条的数据
```

```
            mini_batches = [training_data[k:k+batch_size] for k in range(0, n,
batch_size)]
            for iter_id, mini_batch in enumerate(mini_batches):
                #print(self.w.shape)
                #print(self.b)
                x = mini_batch[:, :-1]
                y = mini_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

# 获取数据
train_data, test_data = load_data()

# 创建网络
net = Network(13)
# 启动训练
losses = net.train(train_data, num_epochs=50, batch_size=100, eta=0.1)

# 画出损失函数的变化趋势
plot_x = np.arange(len(losses))
plot_y = np.array(losses)
plt.plot(plot_x, plot_y)
plt.show()
```



观察上述损失函数的变化，随机梯度下降加快了训练过程，但由于每次仅基于少量样本更新参数和计算损失，所以损失下降曲线会出现震荡。

深度学习框架

- 节省编写大量底层代码的精力：深度学习框架屏蔽了底层实现，用户只需关注模型的逻辑结构，同时简化了计算逻辑，降低了深度学习入门门槛。

- 省去了部署和适配环境的烦恼：深度学习框架具备灵活的移植性，可将代码部署到CPU、GPU和AIPU等芯片上，选择具有分布式能力的深度学习框架会使模型训练更高效。

深度学习框架设计思想

深度学习框架的本质是自动实现建模过程中相对通用的模块，建模者只实现模型中个性化的部分

无论是计算机视觉任务还是自然语言处理任务，使用的深度学习模型代码结构是类似的，只是在每个环节指定的计算单元不同。因此，多数情况下，计算单元只是相对有限的一些选择，如常见的损失函数不超过10种、常用的网络配置也就十几种、常用优化算法不超过五种等等，这些特性使得**基于框架建模更像一个编写“模型配置”的过程。**

使用飞桨重写波士顿房价预测任务

加载飞桨框架的相关类库

```
#加载飞桨、NumPy和相关类库
import paddle
from paddle.nn import Linear
import paddle.nn.functional as F
import numpy as np
import os
import random
```

- paddle：飞桨的主库，paddle 根目录下保留了常用API的别名，当前包括：paddle.tensor、paddle.device目录下的所有API。
- paddle.nn：组网相关的API，包括 Linear、卷积 Conv2D、循环神经网络LSTM、损失函数CrossEntropyLoss、激活函数ReLU等。
- Linear：神经网络的全连接层函数，包含所有输入权重相加的基本神经元结构。在房价预测任务中，使用只有一层的神经网络（全连接层）实现线性回归模型。
- paddle.nn.functional：与paddle.nn一样，包含组网相关的API，如：Linear、激活函数ReLU等，二者包含的同名模块功能相同，运行性能也基本一致。差别在于paddle.nn目录下的模块均是类，每个类自带模块参数；paddle.nn.functional目录下的模块均是函数，需要手动传入函数计算所需要的参数。在实际使用时，卷积、全连接层等本身具有可学习的参数，建议使用paddle.nn实现；而激活函数、池化等操作没有可学习参数，可以考虑使用paddle.nn.functional。

数据处理

数据处理的代码不依赖飞桨框架实现，与使用Python构建房价预测任务的代码相同

```
def load_data():
    # 从文件导入数据
    datafile = './work/housing.data'
    data = np.fromfile(datafile, sep=' ', dtype=np.float32)

    # 每条数据包括14项，其中前面13项是影响因素，第14项是相应的房屋价格中位数
    feature_names = [ 'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', \
                      'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV' ]
    feature_num = len(feature_names)

    # 将原始数据进行Reshape，变成[N, 14]这样的形状
    data = data.reshape([data.shape[0] // feature_num, feature_num])
```

```
    # 将原始数据集拆分成训练集和测试集
    # 使用80%的数据做训练，20%的数据做测试，测试集和训练集不能存在交集
    ratio = 0.8
    offset = int(data.shape[0] * ratio)
    training_data = data[:offset]

    # 计算训练集的最大值和最小值
    maximums, minimums = training_data.max(axis=0), training_data.min(axis=0)

    # 记录数据的归一化参数，在预测时对数据进行归一化
    global max_values
    global min_values

    max_values = maximums
    min_values = minimums

    # 对数据进行归一化处理
    for i in range(feature_num):
        data[:, i] = (data[:, i] - min_values[i]) / (maximums[i] - minimums[i])

    # 划分训练集和测试集
    training_data = data[:offset]
    test_data = data[offset:]
    return training_data, test_data
```

模型设计

通过创建Python类的方式构建模型，该类需要继承`paddle.nn.Layer`父类，并且在类中定义`init`函数和`forward`函数。`forward`是飞桨前向计算逻辑的函数，在调用模型实例时会自动执行，其使用的网络层需要在`init`中声明。

- `init`函数：在类的初始化函数中声明每一层网络的实现函数。在房价预测任务中，只需要定义一层全连接层，模型结构和**第1.3节**保持一致。
- `forward`函数：在构建神经网络时实现前向计算过程，并返回预测结果，在本任务中返回的是房价预测结果。

```
class Regressor(paddle.nn.Layer):

    # self代表类的实例自身
    def __init__(self):
        # 初始化父类中的一些参数
        super(Regressor, self).__init__()

        # 定义一层全连接层，输入维度是13，输出维度是1
        self.fc = Linear(in_features=13, out_features=1)

    # 网络的前向计算
    def forward(self, inputs):
        x = self.fc(inputs)
        return x
```

训练配置

- 声明定义好的回归模型实例为Regressor，并将模型的状态设置为`train`。
- 使用`load_data`函数加载训练数据和测试数据。
- 设置优化算法和学习率，优化算法采用随机梯度下降，学习率设置为0.01。

```
# 声明定义好的线性回归模型
model = Regressor()
# 开启模型训练模式，模型的状态设置为train
model.train()
# 使用load_data加载训练集数据和测试集数据
training_data, test_data = load_data()
# 定义优化算法，采用随机梯度下降SGD
# 学习率设置为0.01，parameters指参数
opt = paddle.optimizer.SGD(learning_rate=0.005, parameters=model.parameters())
```

说明：

模型实例有两种状态：**训练状态**`.train()`和**预测状态**`.eval()`。训练时要执行前向计算和反向传播两个过程，而预测时只需要执行前向计算。为模型指定运行状态，有如下两点原因：

1）部分高级的算子在两个状态执行的逻辑不同，如`Dropout`和`BatchNorm`（在“计算机视觉”章节中详细介绍）。

2）从性能和存储空间的考虑，预测状态时更节省内存(无需记录反向梯度)，性能更好。

训练过程

模型的训练过程采用**二层循环嵌套方式**：

- **内层循环**：按批大小（batch_size，即一次模型训练使用的样本数量），对数据集进行一次遍历，完成一轮模型训练。代码实现为：

```
for iter_id, mini_batch in enumerate(mini_batches):
```

- **外层循环**：定义遍历数据集的次数，即模型训练的轮次（epoch），代码实现为：

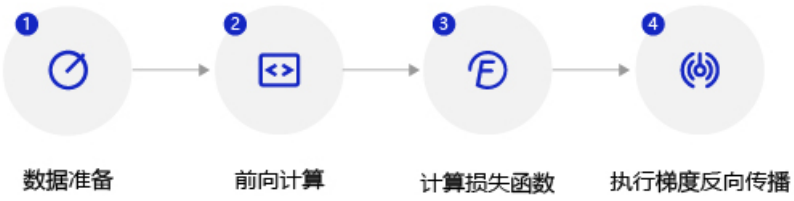
```
for epoch_id in range(EPOCH_NUM):
```

batch_size的取值大小会影响模型训练的效果。

- batch_size过大，模型训练速度越快，但会增大内存消耗，且训练效果并不会明显提升（每次参数只向梯度反方向移动一小步，因此方向没必要特别精确）；

- batch_size过小，模型可以更快地收敛，但训练过程中的梯度方向可能存在较大偏差。

因此，batch_size的大小需要结合具体的任务配置，由于房价预测模型的训练数据集较小，batch_size设置为10。



- 数据准备：将数据先转换成`np.array`格式，再转换成张量（Tensor）。
- 前向计算：按batch_size大小，将数据灌入模型中，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square_error_cost API计算损失函数。
- 反向传播：执行梯度反向传播`backward`，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.step`。

```
epoch_num = 20   # 设置模型训练轮次
batch_size = 10  # 设置批大小，即一次模型训练使用的样本数量

# 定义模型训练轮次epoch（外层循环）
for epoch_id in range(epoch_num):
    # 在每轮迭代开始之前，对训练集数据进行样本乱序
    np.random.shuffle(training_data)
    # 对训练集数据进行拆分，batch_size设置为10
    mini_batches = [training_data[k:k+batch_size] for k in range(0,
len(training_data), batch_size)]
    # 定义模型训练（内层循环）
    for iter_id, mini_batch in enumerate(mini_batches):
        x = np.array(mini_batch[:, :-1]) # 将当前批的房价影响因素的数据转换为np.array格
式
        y = np.array(mini_batch[:, -1:]) # 将当前批的标签数据（真实房价）转换为np.array
格式
        # 将np.array格式的数据转为张量tensor格式
        house_features = paddle.to_tensor(x, dtype='float32')
        prices = paddle.to_tensor(y, dtype='float32')

        # 前向计算
        predicts = model(house_features)

        # 计算损失，损失函数采用平方误差square_error_cost
        loss = F.square_error_cost(predicts, label=prices)
        avg_loss = paddle.mean(loss)
        if iter_id%20==0:
            print("epoch: {}, iter: {}, loss is: {}".format(epoch_id, iter_id,
avg_loss.numpy()))

        # 反向传播，计算每层参数的梯度值
        avg_loss.backward()
        # 更新参数，根据设置好的学习率迭代一步
        opt.step()
        # 清空梯度变量，进行下一轮计算
```

```
        opt.clear_grad()
```

模型保存和推理

模型保存

使用paddle.save API将模型当前的参数 model.state_dict() 保存到文件中，用于模型模型评估或模型推理。

```
# 保存模型参数，文件名为LR_model.pdparams
paddle.save(model.state_dict(), 'LR_model.pdparams')
```

模型推理

任意选择一条样本数据，测试模型的推理效果。推理过程和在应用场景中使用模型的过程一致，主要可分成如下三步：

1）配置模型推理的机器资源。本案例默认使用本机，因此无需代码指定。

2）将训练好的模型参数加载到模型实例中。先从文件中读取模型参数，再将参数加载到模型。加载后，将模型的状态调整为`eval()`。上文提到，训练状态的模型需要同时支持前向计算和反向梯度传播；而评估和推理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。

3）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。

```
def load_one_example():
    # 从测试集中随机选择一条作为推理数据
    idx = np.random.randint(0, test_data.shape[0])
    idx = -10
    one_data, label = test_data[idx, :-1], test_data[idx, -1]
    # 将数据格式修改为[1,13]
    one_data =  one_data.reshape([1,-1])

    return one_data, label

# 将模型参数保存到指定路径中
model_dict = paddle.load('LR_model.pdparams')
model.load_dict(model_dict)
# 将模型状态修改为.eval
model.eval()

one_data, label = load_one_example()
# 将数据格式转换为张量
one_data = paddle.to_tensor(one_data,dtype="float32")
predict = model(one_data)

# 对推理结果进行后处理
predict = predict * (max_values[-1] - min_values[-1]) + min_values[-1]
# 对label数据进行后处理
label = label * (max_values[-1] - min_values[-1]) + min_values[-1]

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

```
Inference result is [[19.59287]], the corresponding label is 19.700000762939453
```

NumPy介绍

NumPy（Numerical Python的简称）是高性能科学计算和数据分析的基础包。

NumPy具有如下功能：

- ndarray数组：一个具有矢量算术运算和复杂广播能力的多维数组，具有快速且节省空间的特点。
- 对整组数据进行快速运算的标准数学函数（无需编写循环）。
- 线性代数、随机数生成以及傅里叶变换功能。
- 读写磁盘数据、操作内存映射文件。

本质上，NumPy期望用户在执行“向量”操作时，像使用“标量”一样轻松。

创建ndarray数组

- `array`：创建嵌套序列（比如由一组等长列表组成的列表），并转换为一个多维数组。

```
# 导入numpy
import numpy as np

# 从list创建array
a = [1,2,3,4,5,6]  # 创建简单的列表
b = np.array(a)    # 将列表转换为数组
b
```

```
array([1, 2, 3, 4, 5, 6])
```

- `arange`：创建元素从0到10依次递增2的数组。

```
# 通过np.arange创建
# 通过指定start, stop (不包括stop), interval来产生一个1维的ndarray
a = np.arange(0, 10, 2)
a
```

```
array([0, 2, 4, 6, 8])
```

- `zeros`：创建指定长度或者形状的全0数组。

```
# 创建全0的ndarray
a = np.zeros([3,3])
a
```

```
array([[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
```

- `ones`：创建指定长度或者形状的全1数组。

```
# 创建全1的ndarray
a = np.ones([3,3])
a
```

```
array([[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]])
```

查看ndarray数组的属性

ndarray的属性包括`shape`、`dtype`、`size`和`ndim`等，通过如下代码可以查看ndarray数组的属性。

- `shape`：数组的形状 ndarray.shape，1维数组（N,）（*N*,），2维数组（M,N）（*M*,*N*），3维数组（M,N,K）3维数组（*M*,*N*,*K*）。
- `dtype`：数组的数据类型。
- `size`：数组中包含的元素个数`ndarray.size`，其大小等于各个维度的长度的乘积。
- `ndim`：数组的维度大小`ndarray.ndim`，其大小等于`ndarray.shape`所包含元素的个数。

```
import numpy as np
b = np.random.rand(10, 10)

b.shape     #(10, 10)
b.size      #100
b.ndim      #2
b.dtype     #dtype('float64')
```

改变ndarray数组的数据类型和形状

```
# 转化数据类型
b = a.astype(np.int64)
print('b, dtype: {}, shape: {}'.format(b.dtype, b.shape))
#b, dtype: int64, shape: (3, 3)

# 改变形状
c = a.reshape([1, 9])
print('c, dtype: {}, shape: {}'.format(c.dtype, c.shape))
#c, dtype: float64, shape: (1, 9)
```

ndarray数组的基本运算

标量和ndarray数组之间的运算

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])

# 标量除以数组，用标量除以数组的每一个元素
1. / arr
#array([[1. , 0.5 , 0.33333333], [0.25 , 0.2 , 0.16666667]])

# 标量乘以数组，用标量乘以数组的每一个元素
2.0 * arr
#array([[ 2., 4., 6.], [ 8., 10., 12.]])

# 标量加上数组，用标量加上数组的每一个元素
2.0 + arr
#array([[3., 4., 5.], [6., 7., 8.]])

# 标量减去数组，用标量减去数组的每一个元素
```

```
2.0 - arr
#array([[ 1., 0., -1.], [-2., -3., -4.]])
```

两个ndarray数组之间的运算

```
arr1 = np.array([[1., 2., 3.], [4., 5., 6.]])
arr2 = np.array([[11., 12., 13.], [21., 22., 23.]])

# 数组 减去 数组， 用对应位置的元素相减
arr1 - arr2
#array([[-10., -10., -10.], [-17., -17., -17.]])

# 数组 加上 数组， 用对应位置的元素相加
arr1 + arr2
#array([[12., 14., 16.], [25., 27., 29.]])

# 数组 乘以 数组，用对应位置的元素相乘
arr1 * arr2
#array([[ 11., 24., 39.], [ 84., 110., 138.]])

# 数组 除以 数组，用对应位置的元素相除
arr1 / arr2
#array([[0.09090909, 0.16666667, 0.23076923], [0.19047619, 0.22727273,
0.26086957]])

# 数组开根号，将每个位置的元素都开根号
arr ** 0.5
#array([[1. , 1.41421356, 1.73205081], [2. , 2.23606798, 2.44948974]])
```

ndarray数组的索引和切片

ndarray数组的索引和切片的使用方式与Python中的list类似。

- 通过[-n , n-1]的下标进行索引
- 通过内置的`slice`函数，设置其`start`,`stop`和`step`参数进行切片，从原数组中切割出一个新数组。

1维ndarray数组的索引和切片

从表面上看，一维数组跟Python列表的功能类似，它们重要区别在于：数组切片产生的新数组，**还是指向原来的内存区域，数据不会被复制**，视图上的任何修改都会**直接反映到源数组上。**将一个标量值赋值给一个切片时，该值会自动传播到整个选区。

```
# 1维数组索引和切片
a = np.arange(30)

a[10]
a
#10

b = a[4:7]
b
#array([4, 5, 6])
```

```
a[4:7] = 10
a
#array([ 0,  1,  2,  3, 10, 10, 10,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

赋值（切片）与copy的区别

```
# 数组切片产生的新数组，还是指向原来的内存区域，数据不会被复制。
# 视图上的任何修改都会直接反映到源数组上。
a = np.arange(30)
arr_slice = a[4:7]      #a[4]开始不包括a[7]
arr_slice[0] = 100      #指向a[4]
a, arr_slice
#(array([ 0,  1,  2,  3, 100,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29]), array([100,  5,  6]))
```

```
# 通过copy给新数组创建不同的内存空间
a = np.arange(30)
arr_slice = a[4:7]
arr_slice = np.copy(arr_slice)
arr_slice[0] = 100
a, arr_slice
#(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29]), array([100,  5,  6]))
```

多维ndarray数组的索引和切片

多维ndarray数组的索引和切片具有如下特点：

- 在多维数组中，各索引位置上的元素不再是标量而是多维数组。
- 以逗号隔开的索引列表来选取单个元素。
- 在多维数组中，如果省略了后面的索引，则返回对象会是一个维度低一点的ndarray。

```
# 创建一个多维数组
a = np.arange(30)
arr3d = a.reshape(5, 3, 2)
arr3d
#array([[[ 0,  1], [ 2,  3], [ 4,  5]], [[ 6,  7], [ 8,  9], [10, 11]], [[12, 13],
[14, 15], [16, 17]], [[18, 19], [20, 21], [22, 23]], [[24, 25], [26, 27], [28,
29]]])
```

方便观察（先均分5组即构建0维，再均分3组即构建1维，再均分2组即构建2维）

```
array([ [[ 0,  1],   [ 2,  3],    [ 4,  5]],
        [[ 6,  7],   [ 8,  9],    [10, 11]],
        [[12, 13],   [14, 15],    [16, 17]],
        [[18, 19],   [20, 21],    [22, 23]],
        [[24, 25],   [26, 27],    [28, 29]]   ])
```

```
# 只有一个索引指标时，会在第0维上索引，后面的维度保持不变
arr3d[0]
#array([[0, 1], [2, 3], [4, 5]])

# 两个索引指标
arr3d[0][1]
#array([2, 3])
```

```
# reshape成一个二维数组
a = a.reshape([6, 4])
a
#array([[ 0, 1, 2, 3], [ 4, 5, 6, 7], [ 8, 9, 10, 11], [12, 13, 14, 15], [16, 17,
18, 19], [20, 21, 22, 23]])

# 使用for语句生成list
[k for k in range(0, 6, 2)]
[0, 2, 4]

# 结合上面列出的for语句的用法
# 使用for语句对数组进行切片
# 下面的代码会生成多个切片构成的list
# k in range(0, 6, 2) 决定了k的取值可以是0, 2, 4
# 产生的list的包含三个切片
# 第一个元素是a[0 : 0+2],
# 第二个元素是a[2 : 2+2],
# 第三个元素是a[4 : 4+2]
slices = [a[k:k+2] for k in range(0, 6, 2)]
slices
#[array([[0, 1, 2, 3], [4, 5, 6, 7]]), array([[ 8, 9, 10, 11], [12, 13, 14,
15]]), array([[16, 17, 18, 19], [20, 21, 22, 23]])]

slices[0]
#array([[0, 1, 2, 3], [4, 5, 6, 7]])
```

ndarray数组的统计方法

可以通过数组上的一组数学函数对整个数组或某个轴向的数据进行统计计算。主要包括如下统计方法：

- `mean`：计算算术平均数，零长度数组的mean为NaN。
- `std`和`var`：计算标准差和方差，自由度可调（默认为n）。
- `sum`：对数组中全部或某轴向的元素求和，零长度数组的sum为0。
- `max`和`min`：计算最大值和最小值。
- `argmin`和`argmax`：分别为最大和最小元素的索引。
- `cumsum`：计算所有元素的累加。
- `cumprod`：计算所有元素的累积。

`sum`、`mean`以及标准差`std`等聚合计算既可以当做数组的实例方法调用，也可以当做NumPy函数使用。

```
arr = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

```
# 计算均值，使用arr.mean() 或 np.mean(arr)，二者是等价的
arr.mean(), np.mean(arr)
# 5.0, 5.0
```

```
# 求和
arr.sum(), np.sum(arr)
# 45, 45
```

```
# 求最大值
arr.max(), np.max(arr)
# 9, 9
```

```
# 求最小值
arr.min(), np.min(arr)
# 1, 1
```

```
# 指定计算的维度
# 沿着第0维求和，看作[[1,2,3], [4,5,6], [7,8,9]]一个整体都看
#也就是将[1, 4, 7]求和等于12，[2, 5, 8]求和等于15，[3, 6, 9]求和等于18
arr.sum(axis = 0)
#array([12, 15, 18])
```

```
# 沿着第1维求平均，看作[1,2,3], [4,5,6], [7,8,9]三个分别单独看
#也就是将[1, 2, 3]取平均等于2，[4, 5, 6]取平均等于5，[7, 8, 9]取平均等于8
arr.mean(axis = 1)
#array([2., 5., 8.])
```

```
# 沿着第0维求最大值，也就是将[1, 4, 7]求最大值等于7，[2, 5, 8]求最大值等于8，[3, 6, 9]求最大值等于9
arr.max(axis = 0)
#array([7, 8, 9])
```

```
# 沿着第1维求最小值，也就是将[1, 2, 3]求最小值等于1，[4, 5, 6]求最小值等于4，[7, 8, 9]求最小值等于7
arr.min(axis = 1)
#array([1, 4, 7])
```

```
# 计算标准差
arr.std()
#2.581988897471611
```

```
# 计算方差
arr.var()
#6.666666666666667
```

```
# 找出最大元素的索引
arr.argmax(), arr.argmax(axis=0), arr.argmax(axis=1)
#8, array([2, 2, 2]), array([2, 2, 2])
```

```
# 找出最小元素的索引
arr.argmin(), arr.argmin(axis=0), arr.argmin(axis=1)
#0, array([0, 0, 0]), array([0, 0, 0])
```

创建随机ndarray数组

创建随机ndarray数组主要包含设置随机种子、均匀分布和正态分布三部分内容

设置随机数种子

```
np.random.seed(10)
a = np.random.rand(3, 3)
a
#array([[0.77132064, 0.02075195, 0.63364823], [0.74880388, 0.49850701,
0.22479665], [0.19806286, 0.76053071, 0.16911084]])
```

均匀分布

```
# 生成均匀分布随机数，随机数取值范围在[0, 1)之间
a = np.random.rand(3, 3)
a
#array([[0.08833981, 0.68535982, 0.95339335], [0.00394827, 0.51219226,
0.81262096], [0.61252607, 0.72175532, 0.29187607]])
```

```
# 生成均匀分布随机数，指定随机数取值范围和数组形状
a = np.random.uniform(low = -1.0, high = 1.0, size=(2,2))
a
#array([[ 0.83554825, 0.42915157], [ 0.08508874, -0.7156599 ]])
```

正态分布

```
# 生成标准正态分布随机数
a = np.random.randn(3, 3)
a
#array([[ 1.484537 , -1.07980489, -1.97772828], [-1.7433723 , 0.26607016,
2.38496733], [ 1.12369125, 1.67262221, 0.09914922]])
```

```
# 生成正态分布随机数，指定均值loc和方差scale
a = np.random.normal(loc = 1.0, scale = 1.0, size = (3,3))
a
#array([[2.39799638, 0.72875201, 1.61320418], [0.73268281, 0.45069099, 1.1327083
], [0.52385799, 2.30847308, 1.19501328]])
```

随机打乱ndarray数组顺序

随机打乱1维ndarray数组顺序

```
# 生成一维数组
a = np.arange(0, 30)
print('before random shuffle: ', a)
# 打乱一维数组顺序
np.random.shuffle(a)
print('after random shuffle: ', a)
```

```
before random shuffle:  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27 28 29]
after random shuffle:  [14  3 20 21  9 26 25  1 22 23  0 11 17 10 27 16  6 19 13
15  8  2 28 18 29  7  4  5 12 24]
```

随机打乱2维ndarray数组顺序

```
# 生成一维数组
a = np.arange(0, 30)
# 将一维数组转化成2维数组
a = a.reshape(10, 3)
print('before random shuffle: \n{}'.format(a))
# 打乱一维数组顺序
np.random.shuffle(a)
print('after random shuffle: \n{}'.format(a))
```

```
before random shuffle:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]
 [24 25 26]
 [27 28 29]]
after random shuffle:
[[ 9 10 11]
 [24 25 26]
 [18 19 20]
 [12 13 14]
 [21 22 23]
 [27 28 29]
 [ 0  1  2]
 [15 16 17]
 [ 6  7  8]
 [ 3  4  5]]
```

只有行的顺序被打乱了，列顺序不变

随机选取元素

```
# 随机选取部分元素
a = np.arange(30)
b = np.random.choice(a, size=5)
b
#array([23, 3, 29, 16, 20])
```

线性代数

线性代数（如矩阵乘法、矩阵分解、行列式以及其他方阵数学等）是任何数组库的重要组成部分，NumPy中实现了线性代数中常用的各种操作，并形成了numpy.linalg线性代数相关的模块。本节主要介绍如下函数：

- `diag`：以一维数组的形式返回方阵的对角线（或非对角线）元素，或将一维数组转换为方阵（非对角线元素为0）。
- `dot`：矩阵乘法。
- `trace`：计算对角线元素的和。
- `det`：计算矩阵行列式。
- `eig`：计算方阵的特征值和特征向量。
- `inv`：计算方阵的逆。

```
# 矩阵相乘
a = np.arange(12)
b = a.reshape([3, 4])
c = a.reshape([4, 3])
# 矩阵b的第二维大小，必须等于矩阵c的第一维大小
d = b.dot(c) # 等价于 np.dot(b, c)
print('a: \n{}'.format(a))
print('b: \n{}'.format(b))
print('c: \n{}'.format(c))
print('d: \n{}'.format(d))
```

```
a:
[ 0  1  2  3  4  5  6  7  8  9 10 11]
b:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
c:
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
d:
[[ 42  48  54]
 [114 136 158]
 [186 224 262]]
```

```
# np.linalg.diag 以一维数组的形式返回方阵的对角线（或非对角线）元素，
# 或将一维数组转换为方阵（非对角线元素为0）
e = np.diag(d)
print('e: \n{}'.format(e))
```

```
e:
[ 42 136 262]
```

```
# trace，计算对角线元素的和
g = np.trace(d)
g
```

```
440
```

```
# det，计算行列式
h = np.linalg.det(d)
h
```

```
1.3642420526593978e-11
```

```
# eig，计算特征值和特征向量
i = np.linalg.eig(d)
i
```

```
(array([ 4.36702561e+02,  3.29743887e+00, -2.00728619e-14]), array([[ 0.17716392,
0.77712552,  0.40824829],  [ 0.5095763 ,  0.07620532, -0.81649658],  [ 0.84198868,
-0.62471488,  0.40824829]]))
```

```
# inv，计算方阵的逆
tmp = np.random.rand(3, 3)
j = np.linalg.inv(tmp)
j
```

```
array([[ -5.81993172,  15.83159278, -11.18756558],  [ 3.67058369,  -6.46065502,
4.50665723],  [ 2.11102657,  -5.91510414,  5.31932688]])
```

NumPy保存和导入文件

文件读写

NumPy可以方便的进行文件读写，如下面这种格式的文本文件

```
0.00632  18.00   2.310  0  0.5380  6.5750  65.20  4.0900   1  296.0  15.30 396.90   4.98  24.00
0.02731   0.00   7.070  0  0.4690  6.4210  78.90  4.9671   2  242.0  17.80 396.90   9.14  21.60
0.02729   0.00   7.070  0  0.4690  7.1850  61.10  4.9671   2  242.0  17.80 392.83   4.03  34.70
0.03237   0.00   2.180  0  0.4580  6.9980  45.80  6.0622   3  222.0  18.70 394.63   2.94  33.40
0.06905   0.00   2.180  0  0.4580  7.1470  54.20  6.0622   3  222.0  18.70 396.90   5.33  36.20
0.02985   0.00   2.180  0  0.4580  6.4300  58.70  6.0622   3  222.0  18.70 394.12   5.21  28.70
0.08829  12.50   7.870  0  0.5240  6.0120  66.60  5.5605   5  311.0  15.20 395.60  12.43  22.90
0.14455  12.50   7.870  0  0.5240  6.1720  96.10  5.9505   5  311.0  15.20 396.90  19.15  27.10
0.21124  12.50   7.870  0  0.5240  5.6310 100.00  6.0821   5  311.0  15.20 386.63  29.93  16.50
0.17004  12.50   7.870  0  0.5240  6.0040  85.90  6.5921   5  311.0  15.20 386.71  17.10  18.90
```

```
# 使用np.fromfile从文本文件'housing.data'读入数据
# 这里要设置参数sep = ' '，表示使用空白字符来分隔数据
# 空格或者回车都属于空白字符，读入的数据被转化成1维数组
d = np.fromfile('./work/housing.data', sep = ' ')
d
```

```
array([6.320e-03, 1.800e+01, 2.310e+00, ..., 3.969e+02, 7.880e+00, 1.190e+01])
```

文件保存

NumPy提供了`save`和`load`接口，直接将数组保存成文件(保存为.npy格式)，或者从.npy文件中读取数组。

```
# 产生随机数组a
a = np.random.rand(3,3)
np.save('a.npy', a)

# 从磁盘文件'a.npy'读入数组
b = np.load('a.npy')

# 检查a和b的数值是否一样
check = (a == b).all()
check
```

```
True
```

一个案例吃透深度学习

手写数字识别任务

数字识别是计算机从纸质文档、照片或其他来源接收、理解并识别可读的数字的能力

使用基于MNIST数据集的手写数字识别模型。MNIST是深度学习领域标准、易用的成熟数据集，包含50 000条训练样本和10 000条测试样本。

- 任务输入：一系列手写数字图片，其中每张图片都是28x28的像素矩阵。
- 任务输出：经过了大小归一化和居中处理，输出对应的0~9的数字标签。

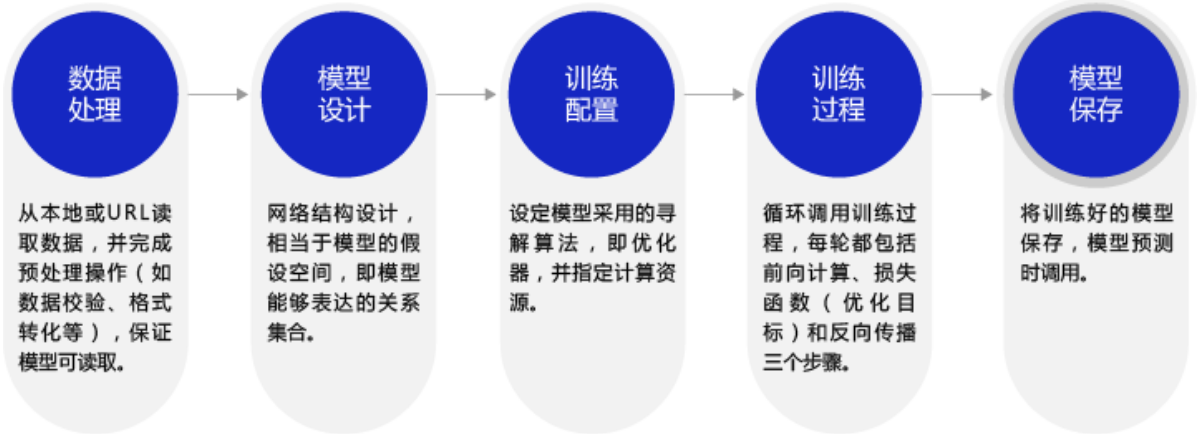


图2：使用飞桨完成手写数字识别任务的流程

2.1.2 飞桨各模型代码结构一致，大大降低了用户的编码难度

在探讨手写数字识别任务的实现方案之前，我们先"偷偷地看"一下程序代码。不难发现，与上一章学习的房价预测任务的代较，二者的程序结构是极为相似的，如 **图3** 所示。

房价预测任务代码

```
#加载飞桨、Numpy和相关类库
import paddle
from paddle.nn import Linear
import paddle.nn.functional as F
import numpy as np
import random

def load_data():
    data_file = './work/housing.data'
    data = np.fromfile(data_file, sep=' ')
    train_data, test_data = data[:offset], data[offset:]
    return train_data, test_data

class Regressor(paddle.nn.Layer):
    def __init__(self):
        super(Regressor, self).__init__()

        # 定义一层全连接层，输入维度是13，输出维度是1
        self.fc = Linear(in_features=13, out_features=1)

    # 网络的前向计算
    def forward(self, inputs):
        x = self.fc(inputs)
    return x

EPOCH_NUM = 10   # 设置外层循环次数
BATCH_SIZE = 10  # 设置batch大小

# 定义外层循环
for epoch_id in range(EPOCH_NUM):
# 在每轮迭代开始之前，将训练数据的顺序随机的打乱
```

数据处理

定义网络

手写数字识别任务代码

```
# 加载飞桨相关库
import paddle
from paddle.nn import Linear
import paddle.nn.functional as F
import numpy as np

train_loader = paddle.io.DataLoader(paddle.vision.datasets.MNIST(mode='train'),
                                    batch_size=16,
                                    shuffle=True)
test_loader = paddle.io.DataLoader(paddle.vision.dataset.Mnist(mode='test'),
                                   batch_size=16,
                                   shuffle=False)

# 定义mnist数据识别网络结构，同房价预测网络
class MNIST(paddle.nn.Layer):
    def __init__(self):
        super(MNIST, self).__init__()

        # 定义一层全连接层，输出维度是10
        self.fc = paddle.nn.Linear(in_features=784, out_features=10)

    # 定义网络结构的前向计算过程
    def forward(self, inputs):
        outputs = self.fc(inputs)
        return outputs

model = MNIST()
model.train()
# 加载训练集 batch_size 设为 16
train_loader = paddle.io.DataLoader(paddle.vision.datasets.MNIST(mode='train'),
                                    batch_size=16,
                                    shuffle=True)
```