

# 线性回归

## sec\_linear\_regression

学习本节，希望你能够掌握以下知识点：

1. 线性回归的基本元素：线性模型、损失函数、基础优化算法；
2. 正态分布的基本概念；
3. 实现线性回归模型的方法；

**回归 (regression)** 是能为一个或多个自变量与因变量之间关系建模的一类方法。

在自然科学和社会科学领域，回归经常用来表示输入和输出之间的关系。

机器学习领域中的大多数任务通常都与**预测 (prediction)** 有关。

当我们想预测一个数值时，就会涉及到回归问题。常见的例子包括：预测价格（房屋、股票等）、预测住院时间（针对住院病人等）、预测需求（零售销量等）。但不是所有的预测都是回归问题。在后面的章节中，我们将介绍分类问题。分类问题的目标是预测数据属于一组类别中的哪一个。

## 线性回归的基本元素

**线性回归 (linear regression)** 可以追溯到19世纪初，它在回归的各种标准工具中最简单而且最流行。

线性回归基于几个简单的假设：

首先，假设自变量 $\mathbf{x}$ 和因变量 $y$ 之间的关系是线性的，即 $y$ 可以表示为 $\mathbf{x}$ 中元素的加权和，这里通常允许包含观测值的一些噪声；其次，我们假设任何噪声都比较正常，如噪声遵循正态分布。

为了解释线性回归，我们举一个实际的例子：我们希望根据与房屋相关的信息来估算房屋价格。

这里的相关信息包括： $x_1$ :卧室个数， $x_2$ :卫生间个数， $x_3$ :居住面积。

那么我们可以得到这样一个式子： $y = w_1x_1 + w_2x_2 + w_3x_3 + b$ 。

### 【线性模型】

#### subsec\_linear\_model

线性假设如下面的式子：

$$y = w_1x_1 + w_2x_2 + w_3x_3 + b$$

**权重 (weight)**  $w$ 决定了每个特征对我们预测值的影响。

$b$ 称为**偏置 (bias)**、**偏移量 (offset)** 或**截距 (intercept)**。

偏置是指当所有特征都取值为0时，预测值应该为多少（原先已经存在）。在现实的任何情况下我们都需要偏置项。如果没有偏置项，我们模型的表达能力将受到限制。

给定一个数据集，我们的目标是寻找模型的权重 $w$ 和偏置 $b$ ，使得根据模型做出的预测大体符合数据里的真实价格。

而在**机器学习(machine learning)**领域，我们通常使用的是高维数据集，建模时采用线性代数表示法会比较方便。

当我们的输入包含 $d$ 个特征时，我们将预测结果 $\hat{y}$ （通常使用“尖角”符号表示 $y$ 的估计值）表示为：

$$\hat{y} = w_1 x_1 + \dots + w_d x_d + b$$

将所有特征放到向量  $\mathbf{x} \in \mathbb{R}^d$  中，并将所有权重放到向量  $\mathbf{w} \in \mathbb{R}^d$  中，我们可以用**点积(scalar product)**形式来简洁地表达模型：

$$\hat{y} = \mathbf{w} \cdot \mathbf{x} + b$$

无论我们使用什么手段来观察特征  $\mathbf{x}$  和标签  $y$ ，都可能会出现少量的**观测误差(observation error)**。因此，即使确信特征与标签的潜在关系是线性的，我们也会加入一个噪声项来考虑观测误差带来的影响。

在开始寻找最好的**模型参数(model parameters)**  $\mathbf{w}$  和  $b$  之前，我们还需要两个东西：

- (1) 一种模型质量的度量方式；
- (2) 一种能够更新模型以提高模型预测质量的方法。

## 【损失函数——一种模型质量的度量方式】

在我们开始考虑如何用模型**拟合(fit)**数据之前，我们需要确定一个拟合程度的度量。**损失函数(loss function)**能够量化目标的**实际值与预测值之间的差距**。通常我们会选择非负数作为损失，且数值越小表示损失越小，完美预测时的损失为0。

回归问题中最常用的损失函数是**平方误差函数(Squared error function)**。当样本的预测值为  $\hat{y}$ ，其相应的真实标签为  $y$  时，平方误差可以定义为以下公式：

$$Loss = \frac{1}{2} \left( \hat{y} - y \right)^2$$

常数  $\frac{1}{2}$  不会带来本质的差别，但这样在形式上稍微简单一些（因为当我们对损失函数求导后常数系数为1）。

我们为一维情况下的回归问题绘制图像：

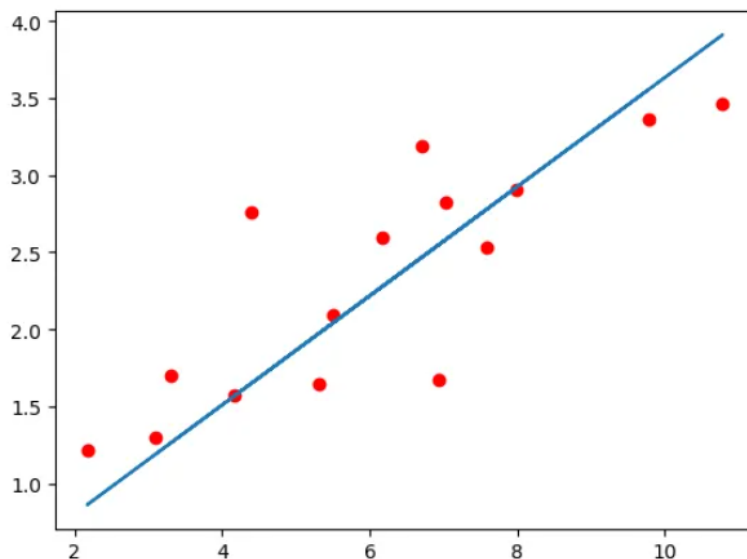


图1：一维线性回归

由于平方误差函数中的二次方项，估计值  $\hat{y}^{(i)}$  和观测值  $y^{(i)}$  之间较大的差异将导致更大的损失。为了度量模型在整个数据集上的质量，我们需计算在训练集  $n$  个样本上的损失均值（也等价于求和）。

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} \left( \mathbf{w} \cdot \mathbf{x}^{(i)} + b - y^{(i)} \right)^2$$

在训练模型时，我们希望寻找一组参数  $(\mathbf{w}^*, b^*)$ ，这组参数能最小化在所有训练样本上的总损失。如下式：

$$\mathbf{w}^*, b^* = \operatorname{argmin}_{\mathbf{w}, b} L(\mathbf{w}, b).$$

## 【基础优化算法——一种能够更新模型以提高模型预测质量的方法】

我们用到一种名为**梯度下降 (gradient descent)** 的方法，这种方法几乎可以优化所有深度学习模型。它通过不断地在损失函数递减的方向上更新参数来降低误差。

梯度下降最简单的用法是计算损失函数（数据集中所有样本的损失均值）关于模型参数的导数（在这里也可以称为梯度）。

但实际中的执行可能会非常慢：因为在每一次更新参数之前，我们必须遍历整个数据集。因此，我们通常会在每次需要计算更新的时候随机抽取一小批样本，这种变体叫做**小批量随机梯度下降 (minibatch stochastic gradient descent)**。

在每次迭代中，我们首先随机抽样一个小批量  $b$ ，它是由固定数量的训练样本组成的。然后，我们计算小批量的平均损失关于模型参数的导数（也可以称为梯度）。最后，我们将梯度乘以一个预先确定的正数  $\eta$ ，并从当前参数的值中减掉。我们用下面的数学公式来表示这一更新过程（ $\partial$  表示偏导数）：

$$\mathbf{w}, b \rightarrow \mathbf{w}, b - \frac{\eta}{|B|} \sum_{i \in B} \partial_{\mathbf{w}, b} L(\mathbf{w}, b).$$

$b$  表示每个小批量中的样本数，这也称为**批量大小 (batch size)**。 $\eta$  表示**学习率 (learning rate)**。

批量大小和学习率的值通常是手动预先指定，而不是通过模型训练得到的。这些可以调整但不在训练过程中更新的参数称为**超参数 (hyperparameter)**。

**调参 (hyperparameter tuning)** 是选择超参数的过程。超参数通常是我们根据训练迭代结果来调整的，而训练迭代结果是在独立的**验证数据集 (validation dataset)** 上评估得到的。

在训练了预先确定的若干迭代次数后（或者直到满足某些其他停止条件后），我们记录下模型参数的估计值，表示为  $\hat{\mathbf{w}}, \hat{b}$ 。

但是，即使我们的函数确实是线性的且无噪声，这些估计值也不会使损失函数真正地达到最小值。因为算法会使得损失向最小值缓慢收敛，但却不能在有限的步数内非常精确地达到最小值。


$$\mathbf{w}_0 \rightarrow \mathbf{w}_{\text{优}}$$

线性回归恰好是一个在整个域中只有一个最小值的学习问题。但是对于像深度神经网络这样复杂的模型来说，损失平面上通常包含多个最小值。深度学习实践者很少会去花费大力气寻找这样一组参数，使得在训练集上的损失达到最小。事实上，更难做到的是找到一组参数，这组参数能够在我们从未见过的数据上实现较低的损失，这一挑战被称为**泛化 (generalization)**。

总结一下，算法的步骤如下：

- （1）初始化模型参数的值，如随机初始化；
- （2）从数据集中随机抽取小批量样本且在负梯度的方向上更新参数，并不断迭代这一步骤。

## 正态分布与平方损失

 subsec\_normal\_distribution\_and\_squared\_loss

正态分布和线性回归之间的关系很密切。

**正态分布 (normal distribution)**，也称为**高斯分布 (Gaussian distribution)**，最早由德国数学家高斯 (Gauss) 应用于天文学研究。

简单的说，若随机变量 $x$ 具有均值 $\mu$ 和方差 $\sigma^2$ ，其正态分布概率密度函数如下：

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

下面我们定义一个Python函数来演示正态分布。

```
import math
import numpy as np
import paddle
from matplotlib import pyplot as plt
from IPython import display

def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 / sigma**2 * (x - mu)**2)
```

```
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-
packages/matplotlib/__init__.py:107: DeprecationWarning: Using or importing the
ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in
3.8 it will stop working
    from collections import MutableMapping
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-
packages/matplotlib/rcsetup.py:20: DeprecationWarning: Using or importing the
ABCs from 'collections' instead of from 'collections.abc' is deprecated, and in
3.8 it will stop working
    from collections import Iterable, Mapping
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-
packages/matplotlib/colors.py:53: DeprecationWarning: Using or importing the ABCs
from 'collections' instead of from 'collections.abc' is deprecated, and in 3.8 it
will stop working
    from collections import Sized
```

我们现在可视化正态分布。

```
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """设置matplotlib的轴"""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()

def use_svg_display(): #@save
    """使用svg格式在Jupyter中显示绘图"""
    display.set_matplotlib_formats('svg')

def set_figsize(figsize=(3.5, 2.5)): #@save
```

```

"""设置matplotlib的图表大小"""
use_svg_display()
plt.rcParams['figure.figsize'] = figsize

def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
        ylim=None, xscale='linear', yscale='linear',
        fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """绘制数据点"""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else plt.gca()

    # 如果X有一个轴, 输出True
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))

    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)

x = np.arange(-7, 7, 0.01)
print(x)

```

```
[-7.   -6.99 -6.98 ...  6.97  6.98  6.99]
```

```

# 均值和标准差对
params = [(0, 1), (0, 2), (3, 1)]

plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
      ylabel='p(x)', figsize=(4.5, 2.5),
      legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])

```

```

/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-
packages/ipykernel_launcher.py:15: DeprecationWarning: `set_matplotlib_formats`
is deprecated since IPython 7.23, directly use
`matplotlib_inline.backend_inline.set_matplotlib_formats()`
from ipykernel import kernelapp as app
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-
packages/matplotlib/cbook/__init__.py:2349: DeprecationWarning: Using or
importing the ABCs from 'collections' instead of from 'collections.abc' is
deprecated, and in 3.8 it will stop working
if isinstance(obj, collections.Iterator):
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-
packages/matplotlib/cbook/__init__.py:2366: DeprecationWarning: Using or
importing the ABCs from 'collections' instead of from 'collections.abc' is
deprecated, and in 3.8 it will stop working
return list(data) if isinstance(data, collections.MappingView) else data

```

就像我们所看到的，均值会产生沿 $x$ 轴的偏移，方差将会影响分布以及峰值。

**均方误差(Mean squared error function)** 损失函数可以用于线性回归的一个原因是：我们假设了观测中包含噪声，其中噪声服从正态分布。噪声正态分布如下式：

$$y = \mathbf{w} \cdot \mathbf{x} + b + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma^2)$$

其中， $\epsilon \sim \mathcal{N}(0, \sigma^2)$ 。

## 正态分布 vs 标准正态分布

在正态分布分布中，根据其概率密度函数，可以知道 $\mu$ 决定其位置，而 $\sigma$ 决定幅度，整体形状呈钟型。而标准正态分布是正态分布的一种，满足 $\mu = 0, \sigma = 1$ 的条件。简单来说幅度限定 $(\sigma = 1)$ ， $y$ 轴对称的正态分布就是标准正态分布。

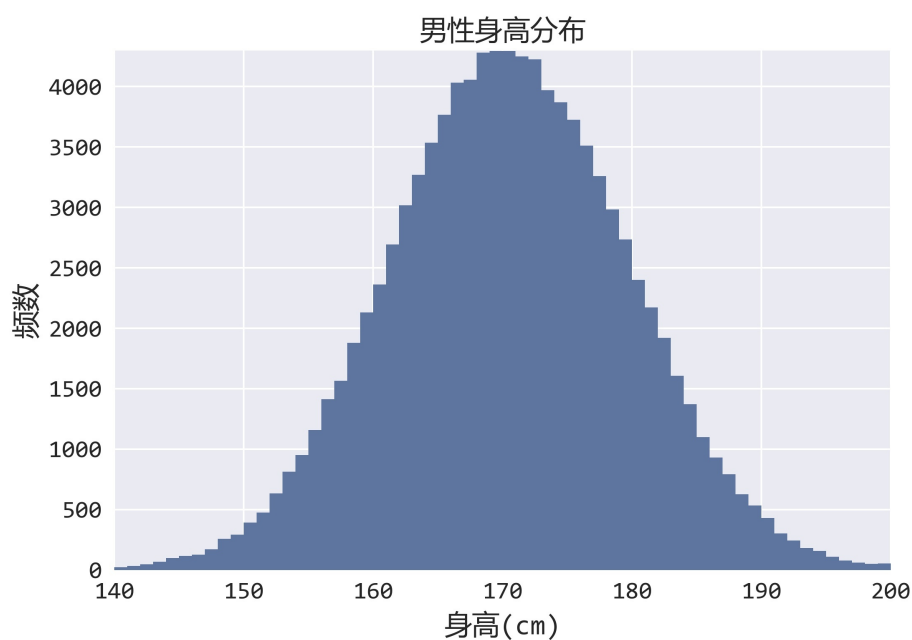



图2：正态分布

这里需要注意的是，以上仅为假设。噪声不见得服从高斯分布，但是对服从高斯分布的噪声，其估计量具有更容易推导的统计性质，也能进行小样本的统计假设的检验。

## 线性回归的从零开始实现

 sec\_linear\_scratch

在了解线性回归的关键思想之后，我们可以开始通过代码来动手实现线性回归了。

在这一节中，(我们将从零开始实现整个方法，包括数据流水线、模型、损失函数和小批量随机梯度下降优化器)。虽然现代的深度学习框架几乎可以自动化地进行所有这些工作，但从零开始实现可以确保你真正知道自己在做什么。同时，了解更细致的工作原理将方便我们自定义模型、自定义层或自定义损失函数。

在之后的章节中，我们会充分利用深度学习框架的优势，介绍更简洁的实现方式。

```
import random
import paddle
import matplotlib.pyplot as plt
```

### 【1.生成数据集】

为了简单起见，我们将[根据带有噪声的线性模型构造一个人造数据集。]

我们的任务是使用这个有限样本的数据集来恢复这个模型的参数。

我们将使用低维数据，这样可以很容易地将其可视化。在下面的代码中，我们生成一个包含1000个样本的数据集，每个样本包含从标准正态分布中采样的2个特征。我们的合成数据集是一个矩阵  $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ 。

我们使用线性模型参数  $\mathbf{w} = [2, -3.4]^{\text{top}}$ 、 $b = 4.2$  和噪声项  $\epsilon$  生成数据集及其标签：

$$y = \mathbf{X} \mathbf{w} + b + \epsilon$$

你可以将  $\epsilon$  视为模型预测和标签时的潜在观测误差。在这里我们认为标准假设成立，即  $\epsilon$  服从均值为0的正态分布。为了简化问题，我们将标准差设为0.01。

下面的代码生成合成数据集(Synthetic datasets)。

```
# 生成数据集
def synthetic_data(w, b, num_examples):
    X = paddle.normal(0, 1, (num_examples, len(w))) # 均值为0, 方差为1, n个样本, w长度的特征
    y = paddle.matmul(X, w) + b
    y += paddle.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1)) # reshape(-1,1) 转换成1列

true_w = paddle.to_tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```



注意, `features` 中的每一行都包含一个二维数据样本,  
`labels` 中的每一行都包含一维标签值 (一个标量) 。

```
print('features:', features[0], '\nlabel:', labels[0])
```

```
# 绘制散点图
```

```
plt.scatter(features[:, 1].detach().numpy(), labels.detach().numpy(), 5)
plt.show()
```

```
features: Tensor(shape=[2], dtype=float32, place=Place(cpu), stop_gradient=True,
      [ 0.88939798, -0.09182443])
label: Tensor(shape=[1], dtype=float32, place=Place(cpu), stop_gradient=True,
      [6.27633619])
```

## 【2.读取数据集】

回想一下, 训练模型时要对数据集进行遍历, 每次抽取一小批量样本, 并使用它们来更新我们的模型。由于这个过程是训练机器学习算法的基础, 所以有必要定义一个函数, 该函数能打乱数据集中的样本并以小批量方式获取数据。

在下面的代码中, 我们定义一个 `data_iter` 函数, 该函数接收批量大小、特征矩阵和标签向量作为输入, 生成大小为 `batch_size` 的小批量。

每个小批量包含一组特征和标签。

```
# 读取数据集
```

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    random.shuffle(indices) # 随机打乱
    for i in range(0, num_examples, batch_size):
        batch_indices = paddle.to_tensor(
            indices[i: min(i + batch_size, num_examples)]) # 随机读取的样本
        yield features[batch_indices], labels[batch_indices] # yield--构造生成器,
        不同于return, 此处函数会继续向下执行
```

通常, 我们利用GPU并行运算的优势, 处理合理大小的“小批量”。每个样本都可以并行地进行模型计算, 且每个样本损失函数的梯度也可以被并行计算。GPU可以在处理几百个样本时, 所花费的时间不比处理一个样本时多太多。

我们直观感受一下小批量运算: 读取第一个小批量数据样本并打印。每个批量的特征维度显示批量大小和输入特征数。同样的, 批量的标签形状与 `batch_size` 相等。



```
batch_size = 10
```

```
for X, y in data_iter(batch_size, features, labels):  
    print(X, '\n', y)  
    break
```

```
Tensor(shape=[10, 2], dtype=float32, place=Place(cpu), stop_gradient=True,  
      [[-0.49911308, -0.39810586],  
       [-0.76750195,  0.92735165],  
       [ 0.50302643, -0.39648017],  
       [ 1.14350212,  0.45846194],  
       [-0.58703244,  0.41355067],  
       [ 0.90497136, -0.58438981],  
       [-0.09798826,  1.66836739],  
       [ 0.07398646, -0.60611689],  
       [ 0.23126918, -0.46856895],  
       [-0.59221536,  0.55128318]])  
Tensor(shape=[10, 1], dtype=float32, place=Place(cpu), stop_gradient=True,  
      [[ 4.56645775],  
       [-0.48575446],  
       [ 6.56141758],  
       [ 4.92852592],  
       [ 1.62627256],  
       [ 8.01213646],  
       [-1.67052901],  
       [ 6.39333963],  
       [ 6.26368427],  
       [ 1.13665283]])
```

### 【3.初始化模型参数】

**[在我们开始用小批量随机梯度下降优化我们的模型参数之前],**

**(我们需要先有一些参数)**。在初始化参数之后，我们的任务是更新这些参数，直到这些参数足够拟合我们的数据。每次更新都需要计算损失函数关于模型参数的梯度。有了这个梯度，我们就可以向减小损失的方向更新每个参数。

在下面的代码中，我们通过从均值为0、标准差为0.01的正态分布中采样随机数来初始化权重，并将偏置初始化为0。

```
# 初始化模型参数  
w = paddle.normal(0, 0.01, shape=(2, 1))  
w.stop_gradient = False #更新参数  
b = paddle.zeros(shape=[1])  
b.stop_gradient = False  
print("w, b: ", w, b)
```

```
w, b: Tensor(shape=[2, 1], dtype=float32, place=Place(cpu),  
stop_gradient=False,  
      [[-0.00806116],  
       [ 0.00138983]]) Tensor(shape=[1], dtype=float32, place=Place(cpu),  
stop_gradient=False,  
      [0.]
```

## 【4.定义模型】

接下来，我们必须**定义模型，将模型的输入和参数同模型的输出关联起来。**]

回想一下，要计算线性模型的输出，我们只需计算输入特征 $\mathbf{X}$ 和模型权重 $\mathbf{w}$ 的矩阵-向量乘法后加上偏置 $b$ 。

注意，上面的 $\mathbf{X}\cdot\mathbf{w}$ 是一个向量，而 $b$ 是一个标量。我们可以通过**广播机制(broadcast mechanism)**：广播机制的本质就是**张量(tensor)**自动扩展，当我们用一个向量加一个标量时，标量会被加到向量的每个分量上。

```
# 定义模型
def linreg(X, w, b):
    # print("linreg: ", X, w, b)
    return paddle.matmul(X, w) + b
```

## 【5.定义损失函数】

因为需要计算损失函数的梯度，所以我们应该先定义损失函数。这里我们使用平方损失函数。

在实现中，我们需要将真实值  $y$  的形状转换为和预测值  $y_{\text{hat}}$  的形状相同。

```
# 定义损失函数
def squared_loss(y_hat, y):
    # print("squared_loss: ", y_hat, y)
    loss = (y_hat - y.reshape(y_hat.shape)) ** 2 / 2 # loss=1/2(y_hat-y)^2
    # print("loss>>>", loss)
    return loss
```

## 【6.定义优化算法】

这里我们介绍小批量随机梯度下降。

在每一步中，使用从数据集中随机抽取的一个小批量，然后根据参数计算损失的梯度。

接下来，朝着减少损失的方向更新我们的参数。

下面的函数实现小批量随机梯度下降更新。

该函数接受模型参数集合、学习速率和批量大小作为输入。每一步更新的大小由学习速率  $lr$  决定。

因为我们计算的损失是一个批量样本的总和，所以我们用批量大小 ( $batch\_size$ )

来规范化步长，这样步长大小就不会取决于我们对批量大小的选择。

```
# 定义优化算法
def sgd(params, lr, batch_size):# params存储参数w、b
    with paddle.no_grad():
        for param in params:
            # print("param.grad: ", param.grad)
            param -= lr * param.grad / batch_size# 梯度下降法公式
            # print("param: ", param)
            param.clear_grad()# 清除当前梯度值
```

## 【7.训练】

现在我们已经准备好了模型训练所有需要的要素，可以实现主要的[训练过程]部分了。理解这段代码至关重要，因为从事深度学习后，你会一遍又一遍地看到几乎相同的训练过程。

在每次迭代中，我们读取一小批量训练样本，并通过我们的模型来获得一组预测；计算完损失后，我们开始反向传播，存储每个参数的梯度；最后，我们调用优化算法 `sgd` 来更新模型参数。

概括一下，我们将执行以下循环：

- 初始化参数
- 重复以下训练，直到完成
  - 计算梯度 $\mathbf{g} \leftarrow \partial \{\mathbf{w}, \mathbf{b}\} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, \mathbf{b})$
  - 更新参数 $\{\mathbf{w}, \mathbf{b}\} \leftarrow \{\mathbf{w}, \mathbf{b}\} - \eta \mathbf{g}$

在每个**迭代周期 (epoch)** 中，我们使用 `data_iter` 函数遍历整个数据集，并将训练数据集中所有样本都使用一次（假设样本数能够被批量大小整除）。这里的迭代周期个数 `num_epochs` 和学习率 `lr` 都是超参数，分别设为3和0.03。设置超参数很棘手，需要通过反复试验进行调整。

```
# 训练
lr = 0.3
num_epochs = 3
net = linreg
loss = squared_loss

for epoch in range(num_epochs):
    for x, y in data_iter(batch_size, features, labels):
        l = loss(net(x, w, b), y)
        l.sum().backward()
        # print(">>>>>>>>>>>>", l)
        sgd([w, b], lr, batch_size)
        # print(">>>>>>>>>>>>", w.grad, b.grad)
        with paddle.no_grad():
            # for param in [w, b]:
            # print("param.grad: ", param.grad)
            w -= lr * w.grad / batch_size
            b -= lr * b.grad / batch_size
            # print("param: ", param)
            w.clear_grad()
            b.clear_grad()
            w.stop_gradient = False
            b.stop_gradient = False
        # print("=====", w, b)

    with paddle.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
/opt/conda/envs/python35-paddle120-env/lib/python3.7/site-
packages/paddle/fluid/dygraph/varbase_patch_methods.py:523: UserWarning: •[93m
warning:
tensor.grad will return the tensor value of the gradient. This is an incompatible
upgrade for tensor.grad API. It's return type changes from numpy.ndarray in
version 2.0 to paddle.Tensor in version 2.1.0. If you want to get the numpy
value of the gradient, you can use :code:`x.grad.numpy()` •[0m
warnings.warn(warning_msg)
```

```
epoch 1, loss 0.000051
epoch 2, loss 0.000050
epoch 3, loss 0.000049
```

因为我们使用的是自己合成的数据集，所以我们知道真正的参数是什么。因此，我们可以通过输出比较**真实参数**和**通过训练学到的参数**来评估训练的成功程度。


注意，我们不应该想当然地认为我们能够完美地求解参数。在机器学习中，**我们通常不太关心恢复真正的参数，而更关心如何高度准确预测参数。**

```
print(w,b)
print(f'w的估计误差: {true_w - w.reshape(true_w.shape)}')
print(f'b的估计误差: {true_b - b}')

```

```
Tensor(shape=[2, 1], dtype=float32, place=Place(cpu), stop_gradient=False,
      [[ 2.00002027],
       [-3.39890456]]) Tensor(shape=[1], dtype=float32, place=Place(cpu),
stop_gradient=False,
      [4.20090818])
w的估计误差: Tensor(shape=[2], dtype=float32, place=Place(cpu),
stop_gradient=False,
      [-0.00002027, -0.00109553])
b的估计误差: Tensor(shape=[1], dtype=float32, place=Place(cpu),
stop_gradient=False,
      [-0.00090837])
```

## 线性回归的简洁实现

 `sec_linear_concise`

### 【深度学习框架】

在过去的几年里，出于对深度学习强烈的兴趣，许多公司、学者和业余爱好者开发了各种成熟的开源框架。深度学习框架有助于建模者聚焦业务场景和模型设计本身，省去大量而繁琐的代码编写工作，其优势主要表现在如下两个方面：

- 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或移动端上，选择具有分布式性能的深度学习框架会使模型训练更高效。

在构建模型的过程中，每一步所需要完成的任务均可以拆分成个性化和通用化两个部分。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，这样可以在“节省投入”和“产出强大”之间达到一个平衡。

- 个性化部分：往往是指定模型由哪些逻辑元素组合，由建模者完成；
- 通用部分：聚焦这些元素的算法实现，由深度学习框架完成。

思考过程	工作内容	工作职责	
		个性化部分 — 建模人员负责	通用部分 - 平台框架负责
Step1 模型设计	假设一种网络结构	设计网络结构	网络模块的实现 (Layer、Tensor)，原子函数的实现 (Numpy)
	设计评价函数 (Loss)	指定Loss函数	Loss函数实现 (cross_entropy)
	寻找优化寻解方法	指定优化算法	优化算法实现 (optimizer)
Step2 准备数据	准备训练数据	提供数据格式与位置，模型接入数据方式	为模型批量送入数据 (io.Dataset、io.DataLoader)
Step3 训练设置	训练配置	单机和多机配置	单机到多机的转换 (transpile)，训练程序的实现 (run)
Step4 应用部署	部署应用或测试环境	确定保存模型和加载模型的环节点	保存模型的实现 (save / load、jit.save / jit.load)
Step5 模型评估	评估模型效果	指定评估指标	指标实现 (Accuracy)、图形化工具 (VisualDL)
Step6 基本过程	全流程串起来	主程序	无

图3：深度学习框架设计思路

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结构都是类似的，只是在每个环节指定的实现算法不同。因此，多数情况下，算法实现只是相对有限的一些选择，如常见的Loss函数不超过十种、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**。

## 【飞桨(PaddlePaddle)深度学习平台】

百度出品的深度学习平台飞桨(PaddlePaddle)是主流深度学习框架中一款完全国产化的产品，与Google TensorFlow、Facebook Pytorch齐名。相比国内其他产品，飞桨是一个功能完整的深度学习平台，也是唯一成熟稳定、具备大规模推广条件的深度学习开源开放平台。飞桨源于产业实践，始终致力于与产业深入融合，与合作伙伴一起帮助越来越多的行业完成AI赋能，并已广泛应用于智慧城市、智能制造、智慧金融、泛交通、泛互联网、智慧农业等领域。



图4：飞桨在各领域的应用

- 飞桨官方网站: <https://www.paddlepaddle.org.cn/>
- 飞桨GitHub: <https://github.com/paddlepaddle>
- 官方API文档: [https://www.paddlepaddle.org.cn/documentation/docs/zh/api/index\\_cn.html](https://www.paddlepaddle.org.cn/documentation/docs/zh/api/index_cn.html)

在本节中，我们将介绍如何通过使用深度学习框架来简洁地实现线性回归模型。

## 【1.生成数据集】

与:numref:sec\_linear\_scratch中类似，我们首先生成数据集。

```
import numpy as np
import paddle
from paddle.io import TensorDataset, DataLoader

# 生成数据集
def synthetic_data(w, b, num_examples):
    x = paddle.normal(0, 1, (num_examples, len(w)))
    y = paddle.matmul(x, w) + b
    y += paddle.normal(0, 0.01, y.shape)
    return x, y.reshape((-1, 1))

true_w = paddle.to_tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

## 【2.读取数据集】

我们可以[调用框架中现有的API来读取数据]。

我们将 features 和 labels 作为API的参数传递，并通过数据迭代器指定 batch\_size。

此外，布尔值 is\_train 表示是否希望数据迭代器对象在每个迭代周期内打乱数据。

```
def load_array(data_array, batch_size, is_train=True):
    # 构造数据迭代器
    dataset = TensorDataset(data_array) # 由张量列表定义的数据集
    print(dataset)
    return DataLoader(dataset, batch_size=batch_size, shuffle=is_train) # 之后从
DataLoader中随机挑选b个样本

batch_size = 10
data_iter = load_array((features, labels), batch_size)
next(iter(data_iter)) # Iterator--迭代器，next不断调用并返回下一个值
```

```
<paddle.fluid.data_loader.dataset.TensorDataset object at 0x7f5f52aaf790>
```

```
[Tensor(shape=[10, 2], dtype=float32, place=Place(cpu), stop_gradient=True,
      [[-0.54224837, -0.39263305],
       [-1.01299679,  0.94332039],
       [ 0.47301245,  0.70776784],
       [ 0.73045009, -1.13905752],
       [ 1.54503405,  0.48081467],
```

```

[ 0.65195972, -1.31317592],
[ 0.11326745, -0.05683252],
[-0.95777512,  0.04747771],
[-0.47813359, -0.18815866],
[-0.20791157,  0.94915169]])],
Tensor(shape=[10, 1], dtype=float32, place=Place(cpu), stop_gradient=True,
[[ 4.44495058],
[-1.02339399],
[ 2.72681808],
[ 9.53551197],
[ 5.67309809],
[ 9.94829559],
[ 4.62414885],
[ 2.12689281],
[ 3.88397408],
[ 0.55416900]])])

```

### 【3.定义模型】

全连接层在 `Linear` 类中定义。值得注意的是，我们将两个参数传递到 `nn.Linear` 中。第一个指定输入特征形状，即2，第二个指定输出特征形状，输出特征形状为单个标量，因此为1。

```

# nn是神经网络的缩写
from paddle import nn
# 线性回归是单层神经网络，Sequential是一个有序的容器，神经网络模块将按照在传入构造器的顺序依次
# 被添加到计算图中执行
net = paddle.nn.Sequential(paddle.nn.Linear(2, 1))

```

### 【4.初始化模型参数】

在使用 `net` 之前，我们需要初始化模型参数。如在线性回归模型中的权重和偏置。

深度学习框架通常有预定义的方法来初始化参数，我们先不使用。在这里，我们指定每个权重参数应该从均值为0、标准差为0.01的正态分布中随机采样，偏置参数将初始化为零。

```

# 初始化模型参数
w = paddle.normal(0, 0.01, shape=(2, 1))
w.stop_gradient = False
b = paddle.zeros(shape=[1])
b.stop_gradient = False
print("w, b: ", w, b)

```

```

w, b: Tensor(shape=[2, 1], dtype=float32, place=Place(cpu),
stop_gradient=False,
[[ -0.00905993],
[ -0.00498687]]) Tensor(shape=[1], dtype=float32, place=Place(cpu),
stop_gradient=False,
[0.])

```



## 【5.定义损失函数】

计算均方误差使用的是 `MSELoss` 类，也称为 `L2` 范数。

默认情况下，它返回所有样本损失的平均值。

```
loss = paddle.nn.MSELoss() # 均方误差
```

## 【6.定义优化算法】

小批量随机梯度下降算法是一种优化神经网络的标准工具，

当我们实例化一个 `SGD` 实例时，我们要指定优化的参数（可通过 `net.parameters()` 从我们的模型中获得）以及优化算法所需的超参数字典。小批量随机梯度下降只需要设置 `lr` 值，这里设置为0.03。

```
trainer = paddle.optimizer.SGD(learning_rate=0.03, parameters =  
net.parameters())
```

## 【7.训练】

通过深度学习框架的高级API来实现我们的模型只需要相对较少的代码。我们不必单独分配参数、不必定义我们的损失函数，也不必手动实现小批量随机梯度下降。当我们需要更复杂的模型时，高级API的优势将大大增加。当我们有了所有的基本组件，训练过程代码与我们从零开始实现时所做的非常相似。

回顾一下：在每个迭代周期里，我们将完整遍历一次数据集（`train_data`），不停地从中获取一个小批量的输入和相应的标签。对于每一个小批量，我们会进行以下步骤：

- 通过调用 `net(x)` 生成预测并计算损失 `l`（前向传播）。
- 通过进行反向传播来计算梯度。
- 通过调用优化器来更新模型参数。

为了更好的衡量训练效果，我们计算每个迭代周期后的损失，并打印它来监控训练过程。

```
num_epochs = 10  
for epoch in range(num_epochs):  
    for x, y in data_iter():  
        l = loss(net(x), y)  
        # 清除梯度  
        trainer.clear_grad()  
        # 反向传播  
        l.backward()  
        # 最小化loss，更新参数  
        trainer.step()  
  
    l = loss(net(features), labels)  
    train_cost = l.numpy()[0]  
    print(f'epoch {epoch + 1}, loss', train_cost)
```

```
epoch 1, loss 0.00017633408
epoch 2, loss 9.7692595e-05
epoch 3, loss 9.7552016e-05
epoch 4, loss 9.749315e-05
epoch 5, loss 9.774655e-05
epoch 6, loss 9.733056e-05
epoch 7, loss 9.7501295e-05
epoch 8, loss 9.8026896e-05
epoch 9, loss 9.82324e-05
epoch 10, loss 9.7922566e-05
```

下面我们比较生成数据集的真实参数和通过有限数据训练获得的模型参数。要访问参数，我们首先从 `net` 访问所需的层，然后读取该层的权重和偏置。正如在从零开始实现中一样，我们估计得到的参数与生成数据的真实参数非常接近。

```
w = net[0].weight
print('w的估计误差: ', true_w - w.reshape(true_w.shape))
b = net[0].bias
print('b的估计误差: ', true_b - b)
```

```
w的估计误差:  Tensor(shape=[2], dtype=float32, place=Place(cpu),
stop_gradient=False,
      [-0.00050569, -0.00013828])
b的估计误差:  Tensor(shape=[1], dtype=float32, place=Place(cpu),
stop_gradient=False,
      [0.00014687])
```

## 小结

- 机器学习模型中的关键要素是训练数据、损失函数、优化算法，还有模型本身。
- 矢量化使数学表达上更简洁，同时运行的更快。
- 线性回归模型也是一个简单的神经网络。
- 梯度下降法通过不断沿着反梯度方向更新参数求解。
- 小批量随机梯度下降是深度学习默认的求解算法。
- 两个重要的超参数是批量大小和学习率。