线性回归

主要内容包括:

1. 线性回归的基本要素
2. 线性回归模型从零开始的实现
3. 线性回归模型使用pytorch的简洁实现

线性回归的基本要素

模型

为了简单起见，这里我们假设价格只取决于房屋状况的两个因素，即面积（平方米）和房龄（年）。接下来我们希望探索价格与这两个因素的具体关系。线性回归假设输出与各个输入之间是线性关系:

$$\mathrm{price} = w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b$$

数据集

我们通常收集一系列的真实数据，例如多栋房屋的真实售出价格和它们对应的面积和房龄。我们希望在这个数据上面寻找模型参数来使模型的预测价格与真实价格的误差最小。在机器学习术语里，该数据集被称为训练数据集（training data set）或训练集（training set），一栋房屋被称为一个样本（sample），其真实售出价格叫作标签（label），用来预测标签的两个因素叫作特征（feature）。特征用来表征样本的特点。

损失函数

在模型训练中，我们需要衡量价格预测值与真实值之间的误差。通常我们会选取一个非负数作为误差，且数值越小表示误差越小。一个常用的选择是平方函数。 它在评估索引为 i 的样本误差的表达式为

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)}\right)^2,$$

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

优化函数 - 随机梯度下降

当模型和损失函数形式较为简单时，上面的误差最小化问题的解可以直接用公式表达出来。这类解叫作解析解（analytical solution）。本节使用的线性回归和平方误差刚好属于这个范畴。然而，大多数深度学习模型并没有解析解，只能通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。这类解叫作数值解（numerical solution）。

在求数值解的优化算法中，小批量随机梯度下降（mini-batch stochastic gradient descent）在深度学习中被广泛使用。它的算法很简单：先选取一组模型参数的初始值，如随机选取；接下来对参数进行多次迭代，使每次迭代都可能降低损失函数的值。在每次迭代中，先随机均匀采样一个由固定数目训练数据样本所组成的小批量（mini-batch）$\mathcal{B}$，然后求小批量中数据样本的平均损失有关模型参数的导数（梯度），最后用此结果与预先设定的一个正数的乘积作为模型参数在本次迭代的减小量。

$$(\mathbf{w},b) \leftarrow (\mathbf{w},b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w},b)$$

学习率: η代表在每次优化中，能够学习的步长的大小
批量大小: $\mathcal{B}$是小批量计算中的批量大小batch size

总结一下，优化函数的有以下两个步骤:

- (i)初始化模型参数，一般来说使用随机初始化;
- (ii)我们在数据上迭代多次，通过在负梯度方向移动参数来更新每个参数。

矢量计算

在模型训练或预测时，我们常常会同时处理多个数据样本并用到矢量计算。在介绍线性回归的矢量计算表达式之前，让我们先考虑对两个向量相加的两种方法。

1. 向量相加的一种方法是，将这两个向量按元素逐一做标量加法。
2. 向量相加的另一种方法是，将这两个向量直接做矢量加法。

In [1]:

```
import torch
import time

# init variable a, b as 1000 dimension vector
n = 1000
a = torch.ones(n)
b = torch.ones(n)
```

In [2]:

```
# define a timer class to record time
class Timer(object):
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        # start the timer
        self.start_time = time.time()

    def stop(self):
        # stop the timer and record time into a list
        self.times.append(time.time() - self.start_time)
        return self.times[-1]

    def avg(self):
        # calculate the average and return
        return sum(self.times)/len(self.times)

    def sum(self):
        # return the sum of recorded time
        return sum(self.times)
```

现在我们可以来测试了。首先将两个向量使用for循环按元素逐一做标量加法。

In [3]:

```
timer = Timer()
c = torch.zeros(n)
for i in range(n):
    c[i] = a[i] + b[i]
'%.5f sec' % timer.stop()
```

Out[3]:

```
'0.02117 sec'
```

另外是使用torch来将两个向量直接做矢量加法：

In [4]:

```
timer.start()
d = a + b
'%.5f sec' % timer.stop()
```

Out[4]:

```
'0.00010 sec'
```

结果很明显,后者比前者运算速度更快。因此，我们应该尽可能采用矢量计算，以提升计算效率。

线性回归模型从零开始的实现

In [5]:

```
# import packages and modules
%matplotlib inline
import torch
from IPython import display
from matplotlib import pyplot as plt
import numpy as np
import random

print(torch.__version__)
```

```
1.3.0
```

生成数据集

使用线性模型来生成数据集，生成一个1000个样本的数据集，下面是用来生成数据的线性关系：

$$\mathrm{price} = w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b$$

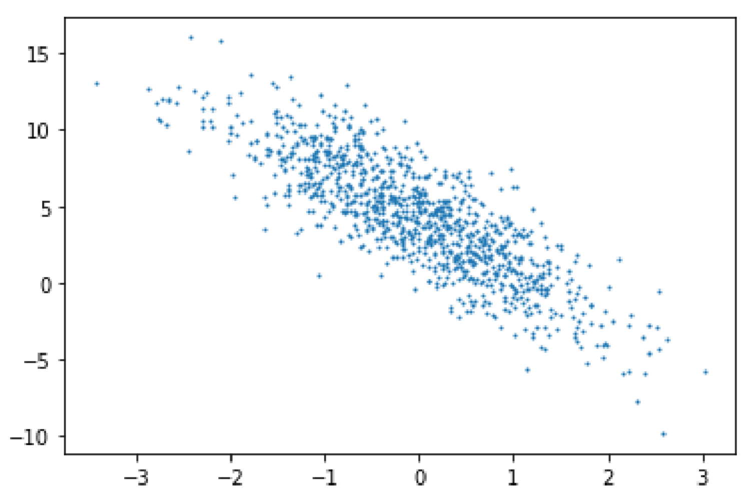
In [6]:

```
# set input feature number
num_inputs = 2
# set example number
num_examples = 1000

# set true weight and bias in order to generate corresponded label
true_w = [2, -3.4]
true_b = 4.2

features = torch.randn(num_examples, num_inputs,
                       dtype=torch.float32)
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()),
                       dtype=torch.float32)
```

使用图像来展示生成的数据

In [7]:

```
plt.scatter(features[:, 1].numpy(), labels.numpy(), 1);
```



读取数据集

In [8]:

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    random.shuffle(indices)  # random read 10 samples
    for i in range(0, num_examples, batch_size):
        j = torch.LongTensor(indices[i: min(i + batch_size, num_examples)]) # the last time may be not enough for a whole batch
        yield  features.index_select(0, j), labels.index_select(0, j)
```

In [9]:

```
batch_size = 10

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

```
tensor([[ 1.4899, -0.1887],
        [ 1.1474,  0.3198],
        [ 0.9972, -1.2064],
        [-2.0386, -1.0788],
        [ 1.6926, -0.1910],
        [ 0.0657, -1.0732],
        [ 0.4882, -0.3328],
        [-1.1173, -1.2231],
        [-1.0749, -0.7744],
        [ 0.1137, -0.2614]])
 tensor([ 7.8333,  5.4163, 10.2924,  3.7887,  8.2214,  7.9877,  6.2968,  6.1212,
         4.6685,  5.3080])
```

初始化模型参数

In [10]:

```
w = torch.tensor(np.random.normal(0, 0.01, (num_inputs, 1)), dtype=torch.float32)
b = torch.zeros(1, dtype=torch.float32)

w.requires_grad_(requires_grad=True)
b.requires_grad_(requires_grad=True)
```

Out[10]:

```
tensor([0.], requires_grad=True)
```

定义模型

定义用来训练参数的训练模型：

$$\mathrm{price} = w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b$$

In [11]:

```
def linreg(X, w, b):
    return torch.mm(X, w) + b
```

定义损失函数

我们使用的是均方误差损失函数：

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2,$$

In [12]:

```
def squared_loss(y_hat, y):
    return (y_hat - y.view(y_hat.size())) ** 2 / 2
```

定义优化函数

在这里优化函数使用的是小批量随机梯度下降：

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b)$$

In [13]:

```
def sgd(params, lr, batch_size):
    for param in params:
        param.data -= lr * param.grad / batch_size # ues .data to operate param without gradient track
```

训练

当数据集、模型、损失函数和优化函数定义完了之后就可来准备进行模型的训练了。

In [14]:

```
# super parameters init
lr = 0.03
num_epochs = 5

net = linreg
loss = squared_loss

# training
for epoch in range(num_epochs):  # training repeats num_epochs times
    # in each epoch, all the samples in dataset will be used once

    # X is the feature and y is the label of a batch sample
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y).sum()
        # calculate the gradient of batch sample loss
        l.backward()
        # using small batch random gradient descent to iter model parameters
        sgd([w, b], lr, batch_size)
        # reset parameter gradient
        w.grad.data.zero_()
        b.grad.data.zero_()
    train_l = loss(net(features, w, b), labels)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().item()))
```

```
epoch 1, loss 0.036378
epoch 2, loss 0.000132
epoch 3, loss 0.000050
epoch 4, loss 0.000049
epoch 5, loss 0.000049
```

In [15]:

```
w, true_w, b, true_b
```

Out[15]:

```
(tensor([[ 2.0005],
         [-3.4001]], requires_grad=True),
 [2, -3.4],
 tensor([4.1998], requires_grad=True),
 4.2)
```

线性回归模型使用pytorch的简洁实现

In [16]:

```
import torch
from torch import nn
import numpy as np
torch.manual_seed(1)

print(torch.__version__)
torch.set_default_tensor_type('torch.FloatTensor')
```

```
1.3.0
```

生成数据集

在这里生成数据集跟从零开始的实现中是完全一样的。

In [17]:

```
num_inputs = 2
num_examples = 1000

true_w = [2, -3.4]
true_b = 4.2

features = torch.tensor(np.random.normal(0, 1, (num_examples, num_inputs)), dtype=torch.float)
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.float)
```

读取数据集

In [18]:

```
import torch.utils.data as Data

batch_size = 10

# combine featues and labels of dataset
dataset = Data.TensorDataset(features, labels)

# put dataset into DataLoader
data_iter = Data.DataLoader(
    dataset=dataset,            # torch TensorDataset format
    batch_size=batch_size,      # mini batch size
    shuffle=True,               # whether shuffle the data or not
    num_workers=2,              # read data in multithreading
)
```

In [19]:

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

```
tensor([[ 0.5907, -1.1609],
        [ 1.0374,  1.0586],
        [-0.0293, -0.4206],
        [-0.0665,  0.9182],
        [ 0.1232, -0.6791],
        [-0.1464,  0.2804],
        [-0.4024,  1.0455],
        [ 0.4956,  0.6140],
        [ 0.8651, -1.5489],
        [-1.7464, -1.6132]])
 tensor([ 9.3312,  2.6703,  5.5726,  0.9442,  6.7577,  2.9510, -0.1692,  3.1062,
        11.1706,  6.1710])
```

定义模型

In [20]:

```
class LinearNet(nn.Module):
    def __init__(self, n_feature):
        super(LinearNet, self).__init__()      # call father function to init
        self.linear = nn.Linear(n_feature, 1)  # function prototype: `torch.nn.Linear(in_features, out_features, bias=True)`

    def forward(self, x):
        y = self.linear(x)
        return y

net = LinearNet(num_inputs)
print(net)
```

```
LinearNet(
  (linear): Linear(in_features=2, out_features=1, bias=True)
)
```

In [21]:

```python
# ways to init a multilayer network
# method one
net = nn.Sequential(
    nn.Linear(num_inputs, 1)
    # other layers can be added here
    )

# method two
net = nn.Sequential()
net.add_module('linear', nn.Linear(num_inputs, 1))
# net.add_module ......

# method three
from collections import OrderedDict
net = nn.Sequential(OrderedDict([
          ('linear', nn.Linear(num_inputs, 1))
          # ......
        ]))

print(net)
print(net[0])
```

```
Sequential(
  (linear): Linear(in_features=2, out_features=1, bias=True)
)
Linear(in_features=2, out_features=1, bias=True)
```

初始化模型参数

In [22]:

```python
from torch.nn import init

init.normal_(net[0].weight, mean=0.0, std=0.01)
init.constant_(net[0].bias, val=0.0)  # or you can use `net[0].bias.data.fill_(0)` to modify it directly
```

Out[22]:

```
Parameter containing:
tensor([0.], requires_grad=True)
```

In [23]:

```python
for param in net.parameters():
    print(param)
```

```
Parameter containing:
tensor([[-0.0142, -0.0161]], requires_grad=True)
Parameter containing:
tensor([0.], requires_grad=True)
```

定义损失函数

In [24]:

```python
loss = nn.MSELoss()    # nn built-in squared loss function
                       # function prototype: `torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')`
```

定义优化函数

In [25]:

```python
import torch.optim as optim

optimizer = optim.SGD(net.parameters(), lr=0.03)   # built-in random gradient descent function
print(optimizer)  # function prototype: `torch.optim.SGD(params, lr=, momentum=0, dampening=0, weight_decay=0, nesterov=False)`
```

```
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.03
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

训练

In [26]:

```python
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        output = net(X)
        l = loss(output, y.view(-1, 1))
        optimizer.zero_grad() # reset gradient, equal to net.zero_grad()
        l.backward()
        optimizer.step()
    print('epoch %d, loss: %f' % (epoch, l.item()))
```

```
epoch 1, loss: 0.000228
epoch 2, loss: 0.000068
epoch 3, loss: 0.000067
```

In [27]:

```python
# result comparision
dense = net[0]
print(true_w, dense.weight.data)
print(true_b, dense.bias.data)
```

```
[2, -3.4] tensor([[ 2.0005, -3.3991]])
4.2 tensor([4.2002])
```

两种实现方式的比较

1. 从零开始的实现（推荐用来学习）

能够更好的理解模型和神经网络底层的原理

1. 使用pytorch的简洁实现

   能够更加快速地完成模型的设计与实现