线性回归的从零开始实现

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。

在这一节中，只使用张量和自动求导。

In [1]:

```
%matplotlib inline
import random
import torch
```

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。** 使用这个有限样本的数据集来恢复模型参数。使用低维数据，易于可视化。

首先，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。 该合成数据集是一个矩阵$\mathbf{X} \in \mathbb{R}^{1000 \times 2}$。

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

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

可以将 ϵ 做为模型预测和标签时的潜在观测误差。 在这里认为标准假设成立，即 ϵ 服从均值为0的正态分布。简便起见，将标准差设为0.01。 下面的代码生成合成数据集。

In [2]:

```
def synthetic_data(w, b, num_examples):  #@save
    """生成y=Xw+b+噪声"""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    print(y.shape)
    return X, y.reshape((-1, 1))
```

In [3]:

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
print(labels.shape)
```

```
torch.Size([1000])
torch.Size([1000, 1])
```

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）。**

In [4]:

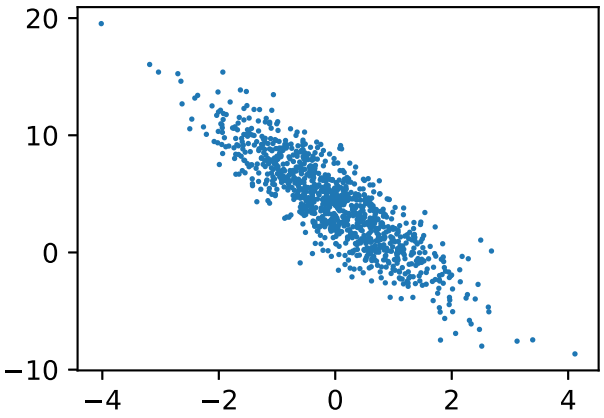
```
print('features:', features[0],'\nlabel:', labels[0])
```

```
features: tensor([0.0899, 0.5965])
label: tensor([2.3570])
```

通过生成第二个特征 `features[:, 1]` 和 `labels` 的散点图， 可以直观观察到两者之间的线性关系。

In [5]:

```
from d2l import torch as d2l
d2l.set_figsize()
d2l.plt.scatter(features[:, (1)].detach().numpy(), labels.detach().numpy(), 1);
```



2.读取数据集

训练模型时要对数据集进行遍历，每次抽取一小批量样本，并使用它们来更新模型。 这个过程是训练机器学习算法的基础，可以定义一个函数， **该函数能打乱数据集中的样本并以小批量方式获取数据**。

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。

In [6]:

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # 这些样本是随机读取的，没有特定的顺序
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
        # yield 返回生成器，下次调用data_iter时从上次yield位置继续执行
```

利用GPU并行运算的优势，处理合理大小的“小批量”（mini batch）。

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。

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。

In [7]:

```
batch_size = 10

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

```
tensor([[-0.8472,  1.4521],
        [-0.5815,  1.0781],
        [-0.1603,  1.0908],
        [ 0.3873,  0.9199],
        [ 0.5644, -2.1162],
        [ 1.2675, -1.3208],
        [ 0.3479,  0.3131],
        [-0.5763,  0.0291],
        [ 0.3370,  0.1551],
        [-2.1485,  1.1329]]) 
 tensor([[-2.4271],
        [-0.6391],
        [ 0.1718],
        [ 1.8541],
        [12.5084],
        [11.2235],
        [ 3.8159],
        [ 2.9393],
        [ 4.3565],
        [-3.9428]])
```

当运行迭代时，可以连续地获得不同的小批量，直至遍历完整个数据集。

上面实现的迭代对于教学来说很好，但它的**执行效率很低**，可能会在实际问题上陷入麻烦。

例如，它要求我们**将所有数据加载到内存中**，并执行**大量的随机内存访问。**

在深度学习框架中实现的内置迭代器效率要高得多， 它可以处理存储在文件中的数据和数据流提供的数据。

3.初始化模型参数

在开始用小批量随机梯度下降优化模型参数之前，（需要先定义一些参数）。

在下面的代码中，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。

In [8]:

```
w = torch.normal(0, 0.01, size=(2,1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

在初始化参数之后，需要在每次迭代更新这些参数，直到它们足够拟合数据。

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，就可以向减小损失的方向更新每个参数。

因为手动计算梯度很枯燥而且容易出错，所以没有人会手动计算梯度。 使用**自动微分来计算梯度**。

4.定义模型

接下来，**定义模型，将模型的输入和参数同模型的输出关联起来。**

要计算线性模型的输出， 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。

上面的$\mathbf{X}\mathbf{w}$是一个向量，而b是一个标量。

PyTorch的**广播机制： 当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。**

In [9]:

```
def linreg(X, w, b):  #@save
    """线性回归模型"""
    return torch.matmul(X, w) + b
# torch.matmul(X, w)结果为向量，b为标量，b被加到torch.matmul(X, w)的每一个分量上
```

5.定义损失函数

为了计算损失函数的梯度，应该先定义损失函数。 这里使用**平方损失函数**：$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$。 在实现中，需要将真实值 `y` 的形状转换为和预测值 `y_hat` 的形状相同。

In [10]:

```
def squared_loss(y_hat, y):  #@save
    """均方损失"""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

6.定义优化算法

线性回归有解析解，但大部分深度神经网络模型却没有。

小批量随机梯度下降：

1. 在每一步中，使用从数据集中随机抽取的一个小批量，根据参数计算损失函数的梯度。
2. 朝着减少损失的方向更新参数。

下面的函数 sgd 实现小批量随机梯度下降更新。 该函数的输入为：模型参数集合、学习速率和批量大小。每一步更新的大小由学习速率 lr 决定。 计算的损失 grad 是一个批量样本损失的总和，所以用批量大小 batch_size 。 来规范化损失（即 grad / batch_size ），这样损失大小就不会取决于我们对批量大小的选择。

```
    def sgd(params, lr, batch_size):  #@save
        """小批量随机梯度下降"""
        with torch.no_grad():
            for param in params:
                param -= lr * param.grad / batch_size
                param.grad.zero_()
```

In [11]:

```
def sgd(params, lr, batch_size):  #@save
    """小批量随机梯度下降"""
    with torch.no_grad():
        for param in params:
            param -= lr * param.grad / batch_size
            param.grad.zero_()
```

7.训练

现在，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。 在每次迭代中，

1. 读取一小批量训练样本，并通过模型来获得一组预测。
2. 计算完损失后，开始反向传播，存储每个参数的梯度。
3. 最后，调用优化算法 sgd 来更新模型参数。

算法流程：

- 初始化参数
- 重复以下训练，直到完成
  - 计算梯度

$$\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$$

  - 更新参数

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$$

在每个**迭代周期**（epoch）中，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。

In [12]:

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
```

In [13]:

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y)  # X和y的小批量损失
        # 因为l形状是(batch_size,1)，而不是一个标量。l中的所有元素被加到一起，
        # 并以此计算关于[w,b]的梯度
        l = l.sum()
        l.backward()
        sgd([w, b], lr, batch_size)  # 使用参数的梯度更新参数
    with torch.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
epoch 1, loss 0.042213
epoch 2, loss 0.000164
epoch 3, loss 0.000051
```

8. 测试结果

因为使用的是人工合成的数据集，所以我们知道真正的参数是什么。 可以通过[**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**]。 通过实验结果可以看出，真实参数和通过训练学到的参数确实非常接近。

In [14]:

```
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}')
```

```
w的估计误差: tensor([ 0.0004, -0.0010], grad_fn=<SubBackward0>)
b的估计误差: tensor([0.0011], grad_fn=<RsubBackward1>)
```

注意，我们不应该想当然地认为能够完美地求解参数。 在机器学习中，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。

幸运的是，即使是在复杂的优化问题上，随机梯度下降通常也能找到非常好的解。 其中一个原因是，在深度网络中存在许多参数组合能够实现高度精确的预测。

小结

- 学习了深度网络是如何实现和优化的。在这一过程中只使用张量和自动微分，不需要定义层或复杂的优化器。
- 本节只触及到了表面知识。在后面章节中，我们将基于刚刚介绍的概念描述其他模型，并学习如何更简洁地实现它们。