线性回归的从零开始实现

这次的学习主要通过阅读并理解代码，了解线性回归是怎么实现的。

生成数据集

> 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我们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，我们将只使用张量和自动求导。 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。\

下面是一段D2L中生成数据集的代码：

```
def synthetic_data(w, b, num_examples):
    """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)
    return X, y.reshape((-1, 1))

true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
print('features:', features[0], '\nlabel:', labels[0])
```

解读这段代码，我发现：

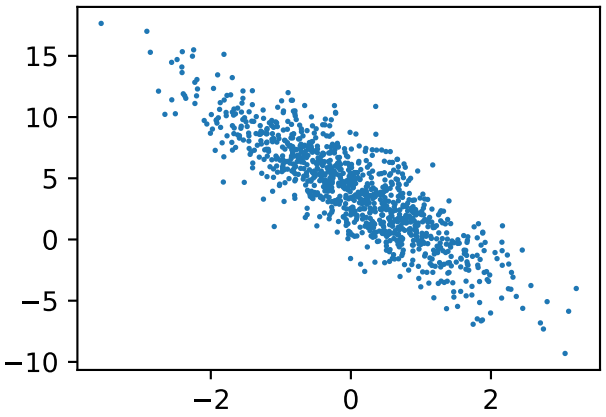
1. `true_w`、`true_b`分别代表着**模型参数**，是固定设置好的。
2. > `torch.normal`用于从正态分布（高斯分布）中生成随机数。这个函数可以根据指定的均值和标准差生成与之匹配的随机数，返回一个张量。你可以指定单个均值和标准差，或者分别指定它们的张量。

   由此我学习了`torch.normal`的用法，函数原型：`torch.normal(mean, std, size)`这是一个按照正态分布随机生成随机张量的函数，三个参数分别是**平均值、标准差、随机张量形状。**

   本段代码中，`X`是一个1000x2的矩阵，一个随机数据集。

   `X`是随机生成的，除此之外，`y`还加上了一个随机的很小的随机数，作为**噪音。**
3. 关于矩阵乘法：`torch.matmul`比`torch.mm`（二维x二维）、`torch.mv`（二维x一维）支持更广泛的情况。并且可以自动广播。

这样我们就得到1000对数据。



可以从图中看出，`features`（x）、`labels`（y）**是呈线性相关的。**

读取数据集

> 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。

```
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]


batch_size = 10

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

1. 这个函数实现了数据的随机读取，其中`random.shuffle`打乱了序列`indices`，实现随机访问。
2. 一个比较巧妙的细节就是，`torch.tensor`可以很好的和python中的`list`结合。具体请看for循环中第一句，这个for循环每次把`[i, i + batch_size)`的数据取出，放入`batch_indices`中，如果最后一次拿取数据不够，那就全拿完。
3. `yield`是我第一次见。搜索信息后，我认为这就是一个特化的`return`，可以把一组值装进一个容器中返回，返回类型是`generator`。在调用函数的时候，我们可以通过`for ... in`来遍历这个容器，获得具体地每一个数据。

> 当我们运行迭代时，我们会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（应该是下文提到的DataLoader）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。

4. 以后基本是用框架，现在只是了解线性回归是如何实现的。

定义模型并训练

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

这段其实没啥好说的，要做模型肯定要有参数，注意要写`requires_grad=True`，因为我们需要通过梯度下降来优化模型。

```
def linreg(X, w, b):
    """线性回归模型"""
    return torch.matmul(X, w) + b


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


def sgd(params, lr, batch_size):
    """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_()
```

结合上一节**线性回归**中的数学公式：

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right),$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right).$$

可以看懂`sgd`函数。函数中，`lr`是学习率，`batch_size`是样本批量。

1. `squared_loss`的函数中把真实值变成预测值的形状。
2. 使用`with torch.no_grad()`可以不计算`params`的梯度，因为`sgd`函数只是在用**已有的梯度（需要在调用之前将损失函数反向传播得到）**做梯度下降，不是要计算梯度。
3. `params`可以看做是**引用**，所以才能修改`w`和`b`的值，这里原文没说。这是因为不同于**C/C++**，**Python**使用的是引用类型，而**C/C++**使用的是值类型。

```
lr = 0.03
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)  # 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.sum().backward()
```

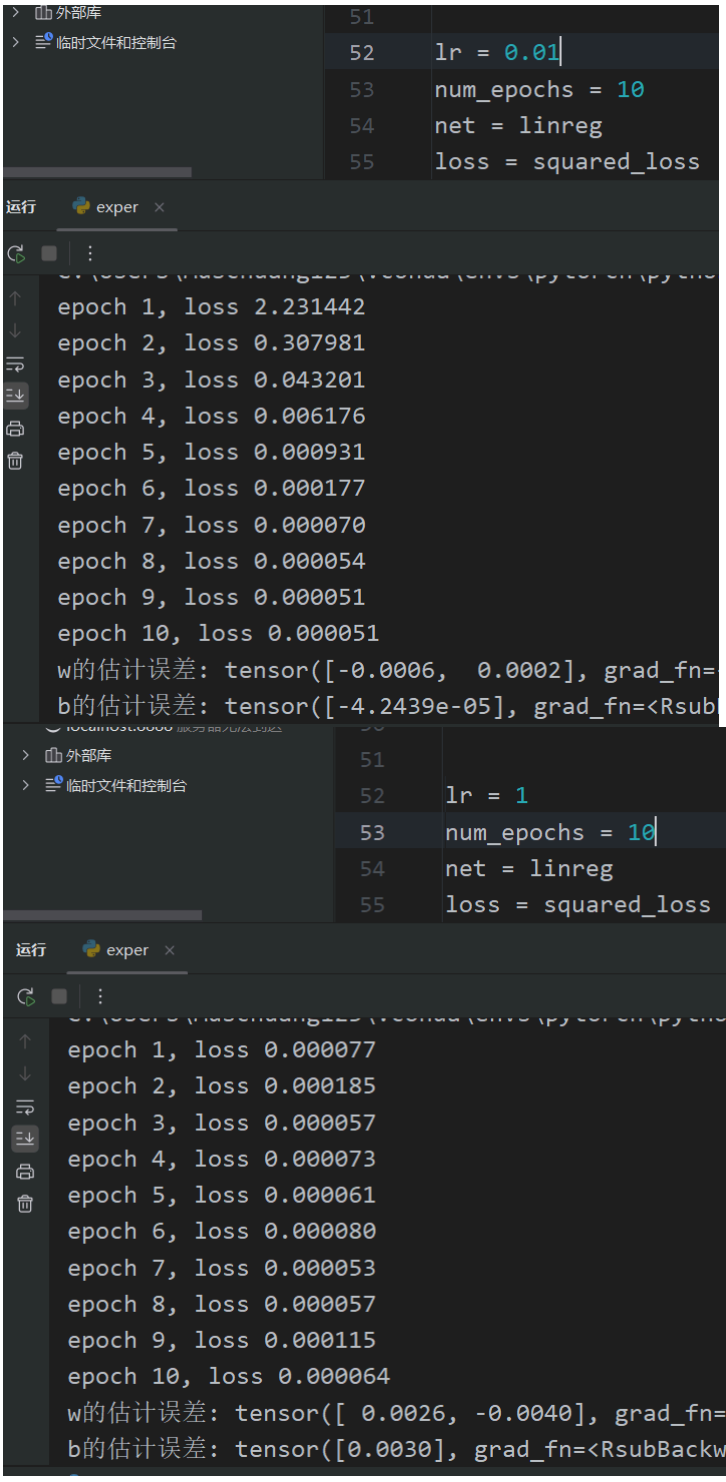
```
            sgd([w, b], lr, batch_size)  # 使用参数的梯度更新参数
        with torch.no_grad():
            train_l = loss(net(features, w, b), labels)
            print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')


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}')
```

1. 这个代码块是训练过程，对于数据集中的每个样本`X`，我们都求出他的**预测值**`net(X, w, b)`并且与**真实值**`y`作出损失函数，用对损失函数反向传播得到的梯度调整`w`和`b`（梯度下降）。这个过程反复执行`num_epochs`次，每次执行完之后，都会输出一下预测值与真实值的偏差。
2. 这里的`num_epochs`和`lr`就是超参数。

   > 设置超参数很棘手，需要通过反复试验进行调整。 我们现在忽略这些细节，以后会在 11节中详细介绍。

3. 修改`num_epochs`，可以增加训练次数，观察参数变化。
4. 通过修改`lr`，我发现学习率不能太高也不能太低。



太低的话需要训练好多次，太高的话会导致偏离真实值（每次都梯度下降过度，一直下降不到最低点）。

5. 文章中有一点没提到，但视频中提到了。那就是**样本个数**不能被**样本批量大小**整除的情况，这样写就是不准确的。**这篇博客**提供了解决方案。不过好像之后可以使用DataLoader（在**读取数据集**标题下提到过），并在其中添加`drop_last=True`解决这个问题，所以学到Dataloader再解决。

未来

3.3. 线性回归的简洁实现 — 动手学深度学习 2.0.0 documentation

这周学过从零开始实现了，下周学简洁实现可能会快一点。

3.4. softmax回归 — 动手学深度学习 2.0.0 documentation

参考文献 / 学习资料

1. 3.2. 线性回归的从零开始实现 — 动手学深度学习 2.0.0 documentation
2. 08 线性回归 + 基础优化算法【动手学深度学习v2】
3. ChatGPT（询问了关于`torch.normal`的用法和`torch`中张量乘法函数的内容）
4. 【Python基础】random.shuffle()的用法-CSDN博客
5. 如何理解Python中的yield用法? - 知乎
6. Why are variables in Python different from other programming languages'? - Software Engineering Stack Exchange
7. 3.1. 线性回归 — 动手学深度学习 2.0.0 documentation
8. 【Pytorch】一文向您详尽解析 with torch.no_grad(): 的高效用法-CSDN博客（这篇文章一看就是用AI写的，并且这个AI上下文理解能力很差，逻辑不清。所以没细看这篇）
9. with torch.no_grad()或@torch.no_grad() 用法-CSDN博客
10. pytorch：batchsize不能整除训练数据大小的解决方案-CSDN博客