线性神经网络

> 本章我们将介绍神经网络的整个训练过程，包括：定义简单的神经网络架构、数据处理、指定损失函数和如何训练模型。为了更容易学习，我们将从经典算法————线性神经网络开始，介绍神经网络的基础知识。

1.手动实现线性回归

手动设置公式应该为：
$y = 2 * x_1 + (-3.4) * x_2 + 4.2$

- 生成数据的函数：

```
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)
      return X, y.reshape((-1, 1))
```

返回的X是features，都是随机生成的，y是根据上面公式 `实际计算` 得到的最后的真实值y。

生成数据：

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

- 有必要写一个函数，每一次返回一个批次的数据：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。

```
def data_iter(batch_size, features, labels):
      num_examples = len(features) #样本数目
      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]
```

  - 对yield和return的比较：
    当函数遇到 `return` 时，会直接返回一个值，并且函数的执行终止。
    当函数执行到 `yield` 时，会返回一个值，同时函数的状态（包括局部变量、函数的当前执行位置等）被保留，暂停执行。函数不会终止，而是暂停，下一次调用生成器时，从 `yield` 后面的位置继续执行。可以多次调用生成器，并每次产生一个新的值，直到函数执行完毕（没有更多 `yield` ）。
    后面使用data_iter的语句是：
    `for X, y in data_iter(batch_size, features, labels):`
- 定义线性回归模型和 `损失函数` ：

```python
def linreg(X, w, b): #@save
    """线性回归模型"""
    return torch.matmul(X, w) + b
def squared_loss(y_hat, y): #@save
    """均方损失"""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

- sgd:
  接受当前的权重参数，学习率，bateh_size，进行更新。

```python
def sgd(params, lr, batch_size):  #@save
    #第一个变量是 当前选择的模型的集合
    """小批量随机梯度下降"""
    with torch.no_grad():
        for param in params:
            # print(param)
            param -= lr * param.grad / batch_size
            param.grad.zero_()
```

- 上面已经把需要的函数都定义完全，开始进行主函数的进行：

```python
#变量的声明
lr = 0.03
num_epochs = 3
net = linreg  #回归模型
loss = squared_loss #损失函数

#上面已经生成了数据，所以可以直接开始训练了。
batch_size = 10
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}')

```

- 对于l.sum().backward的说明：
  如果我们直接调用 `l.backward()`，PyTorch 会尝试计算每个样本对应的损失对模型参数的梯度。然而，通常我们希望更新参数时考虑整个批次的平均误差，**而不是单个样本的误差。因此，我们将这个批次的所有损失加总起来再计算梯度。**

2.调库，简洁实现线性回归

- 还是按照之前一样生成数据，这一部分不变：

```
import numpy as np
import torch
from torch.utils import data
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

- 调用框架中现有的API来读取数据:

```
def load_array(data_arrays, batch_size, is_train=True): #@save
      """构造一个PyTorch数据迭代器"""
      dataset = data.TensorDataset(*data_arrays)
      return data.DataLoader(dataset, batch_size, shuffle=is_train)
```

这个时候，用的是return返回一个结果，要先把这个结果保存下来，之后再用for遍历:

```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

后来的遍历:

```
for X, y in data_iter:
```

- **接下来借助框架中已经定义好的层，进行初始化，达到我们想要的目的:**

对于标准深度学习模型，我们可以使用框架的预定义好的层。这使我们只需关注使用哪些层来构造模型，而不必关注层的实现细节。我们首先定义一个模型变量net，它是一个Sequential类的实例。Sequential类将多个层串联在一起。当给定输入数据时，Sequential实例将数据传入到第一层，然后将第一层的输出作为第二层的输入，以此类推。在下面的例子中，我们的模型只包含一个层，因此实际上不需要Sequential。但是由于以后几乎所有的模型都是多层的，在这里使用Sequential会让你熟悉“标准的流水线”。

线性回归是一个 全连接层 ，我们的这个问题，是2个输入，1个输出。
官方说明：在PyTorch中，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。
所以声明为:

```
from torch import nn
net = nn.Sequential(nn.Linear(2,1))
```

  - 初始化模型的参数:
    正如我们在构造nn.Linear时指定输入和输出尺寸一样，现在我们能直接访问参数以设定它们的初始值。我们通过net[0]选择网络中的第一个图层，然后使用weight.data和bias.data方法访问参数。我们还可以使用替换方法normal_和fill_来重写参数值。
    这里我们使用:

```
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)
```

  - 损失函数:
    计算均方误差使用的是MSELoss类，也称为平方L 2范数。
    优化:
    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，PyTorch在optim模块中实现了该算法的许多变种。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。

```
loss = nn.MSELoss()
trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

- **说明一下整个过程**：

  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。

  最后训练部分的代码：

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        l = loss(net(X) ,y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')

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

- 说明一下：

  每一次更新的代码过程：首先使用net(X)，往前传播，得到传播一层后的结果；使用这个结果和y计算loss；然后使用l.backward()反向传播，程序会自动计算参数的梯度;然后trainer.step()里面会自动进行权重、参数的更新。

  SGD实例里面的step()自动实现了梯度下降的参数更新。

3.SOFTMAX回归：

我们可以用神经网络图 图3.4.1来描述这个计算过程。与线性回归一样，softmax回归也是一个单层神经网络。由于计算每个输出o_1、o_2和o_3取决于所有输入x_1、x_2、x_3和x_4，所以softmax回归的输出层也是全连接层。

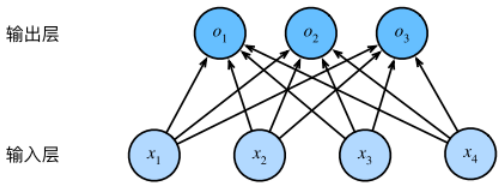

图3.4.1: softmax回归是一种单层神经网络

softmax回归网络也是单层神经网络。

- softmax公式说明 以及 进行如此变换的好处：

  然而我们能否将未规范化的预测o直接视作我们感兴趣的输出呢？答案是否定的。因为将线性层的输出直接视为概率时存在一些问题：一方面，我们没有限制这些输出数字的总和为1。另一方面，根据输入的不同，它们可以为负值。这些违反了2.6节中所说的概率基本公理。

  要将输出视为概率，我们必须保证在任何数据上的输出都是非负的且总和为1。此外，我们需要一个训练的目标函数，来激励模型精准地估计概率。例如，在分类器输出0.5的所有样本中，我们希望这些样本是刚好有一半实际上属于预测的类别。这个属性叫做校准（calibration）。

  社会科学家邓肯·卢斯于1959年在选择模型（choice model）的理论基础上发明的*softmax*函数正是这样做的：softmax函数能够将未规范化的预测变换为非负数并且总和为1，同时让模型保持可导的性质。为了完成这一目标，我们首先对每个未规范化的预测求幂，这样可以确保输出非负。为了确保最终输出的概率值总和为1，我们再让每个求幂后的结果除以它们的总和。如下式：

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{其中} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)} \tag{3.4.3}$$

  这里，对于所有的j总有$0 \leq \hat{y}_j \leq 1$。因此，$\hat{\mathbf{y}}$可以视为一个正确的概率分布。softmax运算不会改变未规范化

4.SOFTMAX回归手动实现：

- 导入库，得到训练集、测试集合。
  定义batch_size为256。

```
import torch
from IPython import display
from d2l import torch as d2l
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

- 每一个样本是一个图片：（1，28，28）。
  展开为一个784维度的向量。 也就是现在，我们把每一个维度都当作一个特征 。

  数据集有10个类别，784维度，所以权重的size：784×10。
  偏置是1×10的行向量。
  初始化最开始的权重和偏置：(权重依然使用正态分布，偏置初始化为0)

```
num_inputs = 784
num_outputs = 10
W = torch.normal(0, 0.01, size=(num_inputs, num_outputs), requires_grad=True)
b = torch.zeros(num_outputs, requires_grad=True)
```

结果：

```
print(W)
print(W.shape)
print(type(W))
```

```
[40]  ✓ 0.0s
...  tensor([[-0.0072,  0.0089,  0.0055,  ..., -0.0059,  0.0043,  0.0071],
          [-0.0056, -0.0037, -0.0176,  ..., -0.0044, -0.0089,  0.0175],
          [ 0.0025, -0.0087,  0.0043,  ...,  0.0145, -0.0052, -0.0010],
          ...,
          [-0.0290, -0.0083,  0.0564,  ...,  0.0016, -0.0462, -0.0143],
          [-0.0118, -0.0055,  0.0133,  ...,  0.0162, -0.0188, -0.0085],
          [ 0.0220,  0.0138, -0.0162,  ...,  0.0009,  0.0041,  0.0043]],
         requires_grad=True)
torch.Size([784, 10])
<class 'torch.Tensor'>
```

- softmax函数实现：
  回想一下，实现softmax由三个步骤组成：

  1. 对每个项求幂（使用exp）；

  2. 对每一行求和（小批量中每个样本是一行），得到每个样本的规范化常数；

  3. 将每一行除以其规范化常数，确保结果的和为1。

  在查看代码之前，我们回顾一下这个表达式：

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}.$$

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdim=True)
    return X_exp / partition  # 这里应用了广播机制
```

- 定义模型：**定义对于数据输入是如何得到最后输出的。**
  - 首先，假定输入是X就是训练集合，size应该是(256 × [1,28,28])。
    因为之前我们说明了每一个batch_size是256，然后每一个图片的尺寸就是一个1*28*28的张量。
    要进行reshape操作：`X.reshape(-1,W.shape[0])`
    -1的意思是，让计算机自己计算是几行。W.shape[0]是权重的size的第一个值，也就是一张图片有多少特征。28 × 28.
  - 输入通过reshape之后，变为2维度的张量。
    和权重相乘同时加上偏置项：
    `torch.matmul(X .reshape((-1, W.shape[0])), W) + b`
    torch.matmul是torch里用于矩阵相乘的库。
    最后的结果还是一个二维的张量。
  - 上述过程其实已经对应了整个fc层次，只不过最后结果需要进行 `softmax` .
  - ```
    def net(X):
        return softmax(torch.matmul(X.reshape((-1, W.shape[0])), W) + b)
    ```

- 使用crossentropy损失函数：

```
def cross_entropy(y_hat, y):
    return - torch.log(y_hat[range(len(y_hat)), y])

cross_entropy(y_hat, y)
```

  y_hat是预测值，y是真实值。
  在这里面**y_hat是一个矩阵**，第一行表示第一个样本预测所有类别的概率，第一行里面第i列的值，表示预测第1个样本就是第i个样本的概率。

- 计算分类中有多少分类正确：
  每一个样本最后的分类结果取所有的类别中概率最大的类别。
  y_hat = y_hat.argmax(axis=1)可直接实现这一步骤。
  之后使用y_hat == y的结果里面有多少个True即可。
  但是注意有数据类型一样时，才能进行==操作。
```

```
def accuracy(y_hat, y):  #@save
    """计算预测正确的数量"""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())
```

- updater的定义：（更新参数的过程）

```
lr = 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)
```

这个过程和上面线性回归的类似，只不过这里已经封装好了。

- 定义每一个epoch里面的过程：
  - 对于训练参数：X,y。X是样本特征，y是真实值。
    y_hat = net(X)。从输入得到输出。
    l = loss(y_hat, y)。计算出来计算值和真实值之间的损失。
  - 之后通过l.sum.backward()和updater(X.shape[0])进行参数更新。
    此时x.shape[0]就是当前的batch_size。
  - 代码：

```
def train_epoch_ch3(net, train_iter, loss, updater):  #@save
    """训练模型一个迭代周期（定义见第3章）"""
    # 将模型设置为训练模式
    if isinstance(net, torch.nn.Module):
        net.train()
    # 训练损失总和、训练准确度总和、样本数
    metric = Accumulator(3)
    for X, y in train_iter:
        # 计算梯度并更新参数
        y_hat = net(X)
        l = loss(y_hat, y)
        if isinstance(updater, torch.optim.Optimizer):
            # 使用PyTorch内置的优化器和损失函数
            updater.zero_grad()
            l.mean().backward()
            updater.step()
        else:
            # print('ok2')
            # 使用定制的优化器和损失函数
            l.sum().backward()
            updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.numel())
    # 返回训练损失和训练精度
    return metric[0] / metric[2], metric[1] / metric[2]
```

- 整个训练过程：

```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """训练模型（定义见第3章）"""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

重点在for循环里面，第一行是进行一个epoch的参数更新、训练，然后借助evaluate_accuracy计算整个batch的精度。

- 训练：
  上面已经封装好了`每一个epoch是如何更新的`，`train函数也写完了`，训练直接调用train_ch3函数即可。

```
num_epochs = 10
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```