PyTorch 从入门到精通

闫涛
科技有限公司
北京
{yt7589}@qq.com

第 1 章张量

Abstract

在本章中我们将讨论 PyTorch 的基础概念张量（Tensor），包括创建、基本属性、基本操作，同时会稍微涉及一下底层原理，是后续学习的基础。

1 张量概述

在 PyTorch 中，尤其是在深度学习应用中，几乎所有运算都是以张量（Tensor）作为输入、输出类型，因此掌握张量（Tensor）的使用，是非常重要的。对于张量的官方介绍大家可以参考facebook [2019]，简单说明见最老程序员 [2019] 的博文。

1.1 创建

我们首先需要可以创建张量（Tensor），在 PyTorch 中有多种方式可以创建张量（Tensor），在这里我们对各种创建方式都会有所提及，但是会重点讲述在实际中最常用的方法。创建张量（Tensor）程序如下所示：

```
import numpy as np
import torch

class Chp001C001(object):
    def __init__(self):
        self.name = 'app.pytorch.book.chp001.Chp001C001'

    def run(self):
        t1 = torch.empty(5)
        t2 = torch.empty([5, 3]) # 创建空张量，元素值为随机数5*3
        t3 = torch.rand(2, 3) # 采用的均匀分布随机数初始化二维张量0~1
        t4 = torch.randn(2, 3) # 以均值为，方差为的正态分布初始化二维张量01
        t5 = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0]) # 最常用方法
        t6 = torch.Tensor([1.0, 2.0, 3.0, 4.0, 5.0]) # 利用构造函数
        # 元素全为的张量1
        t7 = torch.ones(2, 3, dtype=torch.float32)
```

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

```
17          t7_1 = torch.ones_like(t2) # 生成二维张量，元素全为5*31
18          # 创建元素值全为的张量0
19          t8 = torch.zeros(2, 3, dtype=torch.int)
20          t8_1 = torch.zeros_like(t2)
21
22          print('t1:{0}'.format(t1))
23          print('t2:{0}'.format(t2))
24          print('t3:{0}'.format(t3))
25          print('t4:{0}'.format(t4))
26          print('t5:{0}'.format(t5))
27          print('t6:{0}'.format(t6))
28          print('t7:{0}'.format(t7))
29          print('t7_1:{0}'.format(t7_1))
30          print('t8:{0}'.format(t8))
31          print('t8_1:{0}'.format(t8_1))
```

Listing 1: 创建张量（app.pytorch.book.chp001.chp001_c001.py）

运行结果如下所示：

Figure 1: 创建张量（Tensor）运行结果

```
t1:tensor([9.0919e-39, 8.4490e-39, 1.0194e-38, 9.8266e-39, 9.0918e-39])
t2:tensor([[9.0919e-39, 8.4490e-39, 1.0194e-38],
        [9.8266e-39, 8.7245e-39, 9.6429e-39],
        [9.6429e-39, 8.7245e-39, 4.2246e-39],
        [1.0286e-38, 5.0510e-39, 1.0928e-38],
        [1.0102e-38, 8.9082e-39, 9.1837e-39]])
t3:tensor([[0.3666, 0.3690, 0.5528],
        [0.9892, 0.1877, 0.3073]])
t4:tensor([[-0.6425,  0.8573, -1.5098],
        [-2.5404, -0.5440, -1.2088]])
t5:tensor([1., 2., 3., 4., 5.])
t6:tensor([1., 2., 3., 4., 5.])
t7:tensor([[1., 1., 1.],
        [1., 1., 1.]])
t7_1:tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
t8:tensor([[0, 0, 0],
        [0, 0, 0]], dtype=torch.int32)
t8_1:tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

代码解析如下所示：

- 第 9 行：创建 5 个元素的一维张量，并且元素的值为对应内存位置的值，完全随机的值；
- 第 10 行：创建 5×3 的二维张量，元素的值为随机内容；
- 第 11 行：创建 2×3 的二维张量，元素的值由 0~1 之间的均匀分布的随机数进行初始化；
- 第 12 行：创建 2×3 的二维张量，元素的值由均值为 0 方差为 1 的正态分布的随机数进行初始化；
- 第 13 行：由 Python 列表对象创建张量，这是我们最常使用的方法之一，在后面会详细讲解其用法；

- 第 14 行：同样我们还可以用张量类的构造函数来创建，在实际中用的较少，了解即可；
- 第 16 行：创建 2×3 的二维张量，元素值全为 1，并且设置数据类型为 float32，在 PyTorch 中，缺省类型就是 float32；
- 第 17 行：创建与参数张量形状相同的张量，元素值全为 1；
- 第 19 行：创建 2×3 的二维张量，元素值全为 0，并且设置数据类型为 int；
- 第 20 行：创建与参数张量形状相同的张量，元素值全为；

我们在实际应用中，使用 tensor 方法来创建张量最常用，这个函数的使用方式为：

```
1     def test(self):
2         t1 = torch.tensor([1.0, 2.0, 3.0])
3         print('t1 dtype:{0}; device:{1}; layout:{2}'.format(t1.dtype, t1.
      device, t1.layout))
4         t2 = torch.tensor([1.0, 2.0, 3.0], dtype=torch.float64, device=
      torch.device('cuda:0'))
5         print(t2)
```

Listing 2: 张量基本属性（app.pytorch.book.chp001.chp001_c001.py）

运行结果如下所示：

Figure 2: 张量基本属性（Tensor）运行结果

```
t1 dtype:torch.float32; device:cpu; layout:torch.strided
tensor([1., 2., 3.], device='cuda:0', dtype=torch.float64)
```

在上面的代码中，我们首先用缺省的方法创建张量 t1，我们可以看到其缺省的 dtype 为 float32，缺省创建在 CPU 上，缺省的内存布局为 strided。接着我们创建张量 t2，其类型为 float64，并且创建在第一块 GPU 上。

1.2 张量属性

张量最重要的属性是其形状，其次是其轴，我们通过具体的代码来熟悉这些概念。

```
1     def run(self):
2         t1 = torch.tensor([
3             [11.0, 12.0, 13.0, 14.0, 15.0],
4             [21.0, 22.0, 23.0, 24.0, 25.0],
5             [31.0, 32.0, 33.0, 34.0, 35.0]
6         ])
7         print('t1 size:{0}; shape:{1}; axis0:{2}'.format(t1.size(), t1.
      shape, t1.shape[0]))
8         t2 = torch.tensor([
9             [41.0, 42.0, 43.0, 44.0, 45.0]
10        ])
11        t3 = torch.cat((t1, t2), dim=0)
12        print('t3 size:{0}\n{1}'.format(t3.shape, t3))
13        t4 = torch.tensor([
14            [101.0],
15            [102.0],
```

```
            [103.0],
            [104.0]
        ])
        t5 = torch.cat((t3, t4), dim=1)
        print('t5 size:{0}\n{1}'.format(t5.shape, t5))
```

Listing 3: 张量基本属性（app.pytorch.book.chp001.chp001_c002.py）

运行结果如下所示：

Figure 3: 张量性质（Tensor）运行结果

```
t1 size:torch.Size([3, 5]); shape:torch.Size([3, 5]); axis0:3
t3 size:torch.Size([4, 5])
tensor([[11., 12., 13., 14., 15.],
        [21., 22., 23., 24., 25.],
        [31., 32., 33., 34., 35.],
        [41., 42., 43., 44., 45.]])
t5 size:torch.Size([4, 6])
tensor([[ 11.,  12.,  13.,  14.,  15., 101.],
        [ 21.,  22.,  23.,  24.,  25., 102.],
        [ 31.,  32.,  33.,  34.,  35., 103.],
        [ 41.,  42.,  43.,  44.,  45., 104.]])
```

代码解析如下所示：

- 第 2~6 行：创建一个 3×5 的二维张量 t1；
- 第 7 行：其形状可以由 size() 函数或 shape 属性得到，返回值为一个 tuple，所以可以通过下标分别获取；
- 第 8~10 行：声明一个 1×5 的二维张量 t2；
- 第 11 行：将 t2 作为最后一行叠加到 t1 上，这里 dim=0，表明叠加的轴是第 0 维；
- 第 12 行：此时 t3 的形状为 4×5，最后一行为 t2；
- 第 13~18 行：生成一个 4×1 的二维向量；
- 第 19 行：将 t4 叠加到 t3 的后面，指定 dim=1，表明在第 1 维上进行叠加；
- 第 20 行：此时 t5 形状为 4×6，最后一列为 t4；

1.3 与 numpy 互操作

在科学计算中，最常用的库是 numpy，PyTorch 可以视为是对 numpy 在 GPU 上的扩展。通常我们会需要将 numpy 的 ndarray 转换为张量，同时也需要将张量转换为 ndarray，我们推荐大家使用的方法如下所示：

```
    def run(self):
        v1 = np.array([
            [11.0, 12.0, 13.0, 14.0, 15.0],
            [21.0, 22.0, 23.0, 24.0, 25.0],
            [31.0, 32.0, 33.0, 34.0, 35.0]
        ], dtype=np.float32)
        t1 = torch.from_numpy(v1)
        print('t1:{0}'.format(t1))
        t2 = torch.tensor([101.0, 202.0, 303.0])
        v2 = t2.numpy()
        print('v2:{0}'.format(v2))
```

```
12          v1[0][0] = 0.0
13          v1[0][1] = 0.0
14          print('修改后的：t1{0}'.format(t1))
15          t2[0] = 0.0
16          print('修改后的：v2{0}'.format(v2))
```

Listing 4: tensor 和 numpy 互操作（app.pytorch.book.chp001.chp001_c003.py）

运行结果如下所示：

Figure 4: tensor 和 numpy 互操作运行结果

```
t1:tensor([[11., 12., 13., 14., 15.],
        [21., 22., 23., 24., 25.],
        [31., 32., 33., 34., 35.]])
v2:[101. 202. 303.]
修改后的t1：tensor([[ 0.,  0., 13., 14., 15.],
        [21., 22., 23., 24., 25.],
        [31., 32., 33., 34., 35.]])
修改后的v2：[  0. 202. 303.]
```

代码解读如下所示：

- 第 2~6 行：生成一个 numpy 的 ndarray 对象；
- 第 7 行：将其通过 torch.from_numpy 函数，将其转变为对应的 tensor；
- 第 8 行：由打印的内容来看，其与 ndarray 形状和内容相同；
- 第 9 行：生成一个张量；
- 第 10 行：通过 numpy() 函数，将其转变为 numpy 的 ndarray；
- 第 11 行：打印该 ndarray；
- 第 12、13 行：修改原始的 ndarray 中的值；
- 第 14 行：我们在打印由 v1 转换的张量 t1 时，发现其值也已经进行了修改，这说明二者是共用内存的；
- 第 15 行：我们修改原始张量 t2 中的值；
- 第 16 行：打印由 t2 转换的数组 v2 时，其值也发生了改变，这说明二者是共用内存的；

这里需要说明的一点就是 PyTorch 中的 tensor 和 numpy 中的 ndarray 是共用内存的，所以二者之间的转换，由于无需拷贝元素，所以开销是非常小的。

1.4 张量操作

PyTorch 中张量主要有四大类运算：Reshape、Elementwise Operation、Reduction、Access，下面我们分别来进行介绍。

1.4.1 改变形状

在深度学习当中，我们需要知道张量的形状，经常需要改变张量的形状，来满足各种算法要求，程序如下所示：

```
1       def run(self):
2           t1 = torch.tensor([
```

```
            [1.1, 1.2, 1.3],
            [2.1, 2.2, 2.3],
            [3.1, 3.2, 3.3],
            [4.1, 4.2, 4.3]
        ])
        print('t1.shape:{0}'.format(t1.shape))
        t2 = t1.reshape(2, 6)
        print('t2.shape:{0}\r\n{1}'.format(t2.shape, t2))
        t3 = t1.reshape(-1, 4)
        print('t3.shape:{0}\r\n{1}'.format(t3.shape, t3))
        print('元素个数：t1={0}; t2={1}'.format(t1.numel(), t2.numel()))
        t4 = t1.reshape(1, -1).squeeze()
        print('t4:{0}; {1}'.format(t4.shape, t4))
        t7 = torch.rand(28, 28)
        t8 = t7.flatten()
        print(t8.shape)
        t5 = t1.unsqueeze(1)
        print('t5:{0}; {1}'.format(t5.shape, t5))
        t6 = t5.squeeze(1)
        print('t6:{0}; {1}'.format(t6.shape, t6))
        img1 = torch.rand(28, 28)
        img2 = torch.rand(28, 28)
        img3 = torch.rand(28, 28)
        X_raw = torch.cat((img1, img2, img3), dim=0).reshape(3, 28, 28)
        print('X_raw:{0}'.format(X_raw.shape))
        X = X_raw.unsqueeze(1)
        print('X:{0}'.format(X.shape))
        a0 = X.flatten(start_dim=1)
        print('a0.shape:{0}'.format(a0.shape))
```

Listing 5: 张量形状和变形（app.pytorch.book.chp001.chp001_c004.py）

运行结果如下所示：

Figure 5: 张量形状及变形

```
t1.shape:torch.Size([4, 3])
t2.shape:torch.Size([2, 6])
tensor([[1.1000, 1.2000, 1.3000, 2.1000, 2.2000, 2.3000],
        [3.1000, 3.2000, 3.3000, 4.1000, 4.2000, 4.3000]])
t3.shape:torch.Size([3, 4])
tensor([[1.1000, 1.2000, 1.3000, 2.1000],
        [2.2000, 2.3000, 3.1000, 3.2000],
        [3.3000, 4.1000, 4.2000, 4.3000]])
元素个数：t1=12; t2=12
t4:torch.Size([12]); tensor([1.1000, 1.2000, 1.3000, 2.1000, 2.2000, 2.3000, 3.1000, 3.2000, 3.3000,
        4.1000, 4.2000, 4.3000])
torch.Size([784])
t5:torch.Size([4, 1, 3]); tensor([[[1.1000, 1.2000, 1.3000]],

        [[2.1000, 2.2000, 2.3000]],

        [[3.1000, 3.2000, 3.3000]],

        [[4.1000, 4.2000, 4.3000]]])
t6:torch.Size([4, 3]); tensor([[1.1000, 1.2000, 1.3000],
        [2.1000, 2.2000, 2.3000],
        [3.1000, 3.2000, 3.3000],
        [4.1000, 4.2000, 4.3000]])
X_raw:torch.Size([3, 28, 28])
X:torch.Size([3, 1, 28, 28])
a0.shape:torch.Size([3, 784])
```

代码解读如下：

- 第 2~7 行：定义一个 4×3 的二维张量；
- 第 8 行：打印张量的原始形状；
- 第 9 行：将张量的形状变为 2×6 并赋给 t2，注意这时 t1 的形状仍然为 4×3，t1 和 t2 会共用内存，所以变形操作开销是非常小的；
- 第 10 行：打印 t2 的内容，我们可以看到张量变形的算法原理：将原始张量展开为一维，然后重新进行分配；
- 第 11 行：我们改变张量的形状，但是这时我们可以将一维指定为 -1，PyTorch 可以根据原始张量元素总数和其他维的数量推出此维数量；
- 第 12 行：从打印的结果我们可以看出，PyTorch 正确推出指定为 -1 的维的数量为 3；
- 第 13 行：可以通过 numel() 函数求出张量的元素数量；
- 第 14 行：我们先将原始张量 t1 变为 1×12 的张量，然后去掉第 1 维的 1，变为 [12] 的张量，在后面我们将看到，这是一个常用的技巧；
- 第 15 行：从打印结果可以看出，我们将原始的二维张量变为一维张量；
- 第 16~18 行：我们声明一个 28×28 的二维张量，通过调用 torch.flatten 函数，将其变为一维张量 [784]，其实这里实现的与第 14 行的方法相同；
- 第 19、20 行：在第 1 维后增加一维，我们看到张量形状由 4×3，变为 4×1×3；
- 第 21、22 行：将上步加入的维度去掉，使形状由 4×1×3 重新变回 4×3；
- 第 23~25 行：定义三个张量用来保存 28×28 的图像数据；
- 第 26、27 行：将三个图像数据按行叠加起来生成一个新张量，其形状为 3×28×28；
- 第 28、29 行：在第 1 维处加入一列，使其形状变为 3×1×28×28，这是图像处理应用中一种常见的格式，其中第一维代表迷你批次的索引号，第 2 维代表颜色，第 3、4 维为图像的宽度和高度；
- 第 30、31 行：在实际神经网络中，需要将迷你批次中的图像，每一行一个图像的形式输入，即将张量的形状变为 3×784；

1.4.2 元素操作

我们经常需要对张量中的元素进行操作，可以分为四种情况：

1. 相同形状张量之间；
2. 张量与标量；
3. 不同形状张量之间；
4. 函数以张量作为参数；

其中第一、二种情况，可以统一用 Broadcast 来进行理解。具体应用代码如下所示：

```
1     def run(self):
2         t1 = torch.tensor([
3             [1.1, 1.2, 1.3],
4             [2.1, 2.2, 2.3]
```

```
        ])
        t2 = torch.tensor([
            [10.1, 11.0, 12.0],
            [20.0, 21.0, 22.0]
        ])
        print('操作符：\r\nt1+t2={0};\r\nt1-t2={1};\r\nt1*t2={2};'\
            '\r\nt1/t2={3}'\
            .format(t1+t2, t1-t2, t1*t2, t1/t2))
        print('函数：\r\nt1+t2={0};\r\nt1-t2={1};\r\nt1*t2={2};'\
            '\r\nt1/t2={3}'\
            .format(t1.add(t2), t1.sub(t2), t1.mul(t2), t1.div(t2)))
        print('张量加标量（操作符形式）：\r\nt1+t2={0};\r\n'\
            't1-t2={1};\r\nt1*t2={2};\r\nt1/t2={3}'\
            .format(t1+2, t1-2, t1*2, t1/2))
        print('逻辑运算符：\r\nt1>1.2{0};\r\n'\
            't1<1.2{1};\r\nt1==1.1{2}'\
            .format(t1>1.2, t1<1.2, t1==1.1))
        t3 = torch.tensor([100.0, 200.0, 300.0])
        print('不同形状标量运算：
    \r\nt1+t3={0};\r\nt1-t3={1};\r\nt1*t3={2};'\
            '\r\nt1/t3={3}'\
            .format(t1+t3, t1-t3, t1*t3, t1/t3))
        print('以张量为参数函数：{0}'.format(self.fx(t1)))

    def fx(self, x):
        return x*100 + 8
```

Listing 6: 张量元素运算（app.pytorch.book.chp001.chp001_c005.py）

运行结果如下所示：

Figure 6: 张量元素运算运行结果

```
操作符：
t1+t2=tensor([[11.2000, 12.2000, 13.3000],
        [22.1000, 23.2000, 24.3000]]);
t1-t2=tensor([[ -9.0000,  -9.8000, -10.7000],
        [-17.9000, -18.8000, -19.7000]]);
t1*t2=tensor([[11.1100, 13.2000, 15.6000],
        [42.0000, 46.2000, 50.6000]]);
t1/t2=tensor([[0.1089, 0.1091, 0.1083],
        [0.1050, 0.1048, 0.1045]])
函数：
t1+t2=tensor([[11.2000, 12.2000, 13.3000],
        [22.1000, 23.2000, 24.3000]]);
t1-t2=tensor([[ -9.0000,  -9.8000, -10.7000],
        [-17.9000, -18.8000, -19.7000]]);
t1*t2=tensor([[11.1100, 13.2000, 15.6000],
        [42.0000, 46.2000, 50.6000]]);
t1/t2=tensor([[0.1089, 0.1091, 0.1083],
        [0.1050, 0.1048, 0.1045]])
张量加标量（操作符形式）：
t1+t2=tensor([[3.1000, 3.2000, 3.3000],
        [4.1000, 4.2000, 4.3000]]);
t1-t2=tensor([[-0.9000, -0.8000, -0.7000],
        [ 0.1000,  0.2000,  0.3000]]);
t1*t2=tensor([[2.2000, 2.4000, 2.6000],
        [4.2000, 4.4000, 4.6000]]);
t1/t2=tensor([[0.5500, 0.6000, 0.6500],
        [1.0500, 1.1000, 1.1500]])
逻辑运算符：
t1>1.2tensor([[False, False,  True],
        [ True,  True,  True]]);
t1<1.2tensor([[ True, False, False],
        [False, False, False]]);
t1==1.1tensor([[ True, False, False],
        [False, False, False]])
不同形状标量运算：
t1+t3=tensor([[101.1000, 201.2000, 301.3000],
        [102.1000, 202.2000, 302.3000]]);
t1-t3=tensor([[ -98.9000, -198.8000, -298.7000],
        [ -97.9000, -197.8000, -297.7000]]);
t1*t3=tensor([[110.0000, 240.0000, 390.0000],
        [210.0000, 440.0000, 690.0000]]);
t1/t3=tensor([[0.0110, 0.0060, 0.0043],
        [0.0210, 0.0110, 0.0077]])
以张量为参数函数：tensor([[118.0000, 128.0000, 138.0000],
        [218.0000, 228.0000, 238.0000]])
```

代码解读如下所示：

- 第 2∼5 行：生成一个 2×3 的张量；
- 第 6∼9 行：生成一个 2×3 的张量 t2；
- 第 10∼12 行：张量 t1 和 t2 之间加、减、乘、除运算，为对应位置元素进行相应的运算；
- 第 13∼15 行：除使用 +、-、*、/运算外，还可以使用 add、sub、mul、div 函数，二者的效果相同；
- 第 16∼18 行：张量加一个标量，虽然我们可以简单的理解为拿标量对所有张量元素进行计算，但是实际上，PyTorch 是进行了 Broadcast 操作，即在运算前将标量修改为：

$$2 \rightarrow \begin{bmatrix} 2.0 & 2.0 & 2.0 \\ 2.0 & 2.0 & 2.0 \end{bmatrix} \tag{1}$$

  这样就将问题转化为相同形状的张量之间的运算了。
- 第 19∼21 行：与张量与标量运算类似，也是先进行 Broadcast 操作，然对对应位置的元素进行逻辑运算，最后运算结果为 True 或 False 组成的同形状张量；
- 第 22 行：我们定义一个与 t1 行形状相同的向量 t3；

- 第 23~25 行：PyTorch 也首先进行 Broadcast 操作，将 t3 变为如下形式：

$$\begin{bmatrix} 100.0 & 200.0 & 300.0 \end{bmatrix} \rightarrow \begin{bmatrix} 100.0 & 200.0 & 300.0 \\ 100.0 & 200.0 & 300.0 \end{bmatrix} \tag{2}$$

  这时就将问题转换为同形状张量之间的运算了。

- 第 26 行：当函数参数为张量时，需要对张量每个元素应用该函数，结果为同形状张量；

1.4.3 Reduction

Reduction 操作主要有求和、均值、方差，还有一个应用很广的函数 argmax，用于多分类问题中找出正确的类别，代码如下所示：

```
1     def run(self):
2         t1 = torch.tensor([
3             [1.0, 2.0, 3.0],
4             [4.0, 5.0, 6.0]
5         ])
6         print('全部和：{0}; 行和：{1}; 列合：{2}'.format(t1.sum(),
7                     t1.sum(dim=1), t1.sum(dim=0)))
8         print('均值：{0}; 行：{1}; 列：{2}'.format(t1.mean(),
9                     t1.mean(dim=1), t1.mean(dim=0)))
10        print('方差：{0}; 行：{1}; 列：{2}'.format(t1.std(),
11                    t1.std(dim=1), t1.std(dim=0)))
12        print('最大值：全部：{0}; 行：{1}; 列：{2}'.format(t1.max(),
13                    torch.max(t1, 1), torch.max(t1, 0)))
14        t2 = torch.tensor([
15            [0.1, 0.5, 0.4],
16            [0.9, 0.05, 0.05]
17        ])
18        print('类别：{0}'.format(t2.argmax(dim=1)))
```

Listing 7: 张量 Reduction（app.pytorch.book.chp001.chp001_c006.py）

运行结果如下所示：

Figure 7: 张量 Reduction 运算运行结果

```
全部和：21.0; 行和：tensor([ 6., 15.]); 列合：tensor([5., 7., 9.])
均值：3.5; 行：tensor([2., 5.]); 列：tensor([2.5000, 3.5000, 4.5000])
方差：1.8708287477493286; 行：tensor([1., 1.]); 列：tensor([2.1213, 2.1213, 2.1213])
最大值：全部：6.0; 行：torch.return_types.max(
values=tensor([3., 6.]),
indices=tensor([2, 2])); 列：torch.return_types.max(
values=tensor([4., 5., 6.]),
indices=tensor([1, 1, 1]))
类别：tensor([1, 0])
```

代码解读如下所示：

- 第 2~5 行：生成一个 2×3 的二维张量 t1；
- 第 6、7 行：求出并打印全部元素的和，每行元素的和，每列元素的和；
- 第 8、9 行：求出并打印全部元素的均值，每行元素的均值，每列元素的均值；

- 第 10、11 行：求出并打印全部元素的方差，每行元素的方差，每列元素的方差；
- 第 12、13 行：求出并打印全部元素的最大值，求出每行元素的最大值，返回结果 values 为每行的最大值，indices 为每行最大元素所在的索引号。求出每列元素的最大值，返回结果 values 为每列的最大值，indices 为每列最大元素所在的索引号；
- 第 12~17 行：定义一个三分类问题，两个样本的输出层（采用 softmax 激活函数）输出值；
- 第 18 行：用 argmax 求出每一行最大元素所在的索引号；

1.4.4 索引和切片

PyTorch 中的张量既支持 Python 标准的切片方法，也支持一些特有的方法，如下所示：

```
    def run(self):
        # cifar-10 个样本为一个批次16
        X = torch.rand(16, 3, 32, 32)
        # from zero start_idx(inclusive); end_idx(exclusive); 步长step
        # 当为负数时由开始，表示是最后一个元素-1
        print('前两张图：{0}'.format(X[:2, :, :, :].shape))
        print('最后一张图：{0}'.format(X[-1:, :, :, :].shape))
        print('取偶数图片：{0}'.format(X[1::2, :, :, :].shape))
        print('选第、、张图：189{0}'.format(X.index_select(0,
                    torch.tensor([0, 7, 8])).shape))
        print('取和两通道：RB{0}'.format(X.index_select(1,
                    torch.tensor([0, 2])).shape))
        print('取每张图片区域：8*8{0}'.format( X.index_select(2,
                    torch.arange(8)).index_select(3,
                    torch.arange(6)).shape ))
        print('语法糖：第张图：1{0}; 颜色通道：G{1}'.format(
            X[0, ...].shape, X[:, 1, ...].shape)
        )
        mask = X.ge(0.5)
        print('大于的元素：0.5{0}'.format(
            torch.masked_select(X, mask))
        )
        t2 = torch.tensor([[3.0, 3.1, 3.2], [3.3, 3.4, 3.5]])
        print('按维索引取值：1{0}'.format(torch.take(t2,
                    torch.tensor([0, 1, 5]))))
```

Listing 8: 张量索引和切片（app.pytorch.book.chp001.chp001_c007.py）

运行结果如下所示：

Figure 8: 张量索引和切片运算运行结果

```
前两张图：torch.Size([2, 3, 32, 32])
最后一张图：torch.Size([1, 3, 32, 32])
取奇数图片：torch.Size([8, 3, 32, 32])
选第1、8、9张图：torch.Size([3, 3, 32, 32])
取R和B两通道：torch.Size([16, 2, 32, 32])
取每张图片8*8区域：torch.Size([16, 3, 8, 6])
语法糖：第1张图：torch.Size([3, 32, 32])；颜色G通道：torch.Size([16, 32, 32])
大于0.5的元素：tensor([0.5057, 0.8278, 0.7129,  ..., 0.8630, 0.6127, 0.6092])
按1维索引取值：tensor([3.0000, 3.1000, 3.5000])
```

代码解读如下所示：

- 第 3 行：定义一个 CIFAR-10 数据集用于卷积神经网络输入的张量；
- 第 6 行：X[:2, :, :, :] 代表对第 1 维从索引号 0 开始，直到索引号 2 为止，但不包括索引号 2，即表示批次中的第 1、2 个样本；
- 第 7 行：代表从第 1 维最后一个元素开始，因此就是最后一张图片；
- 第 8 行：对第 1 维从索引号为 1 开始，隔一个取一个，取偶数编号的图片；
- 第 9、10 行：这里展示的是 index_select 函数的用法，函数的第 1 个参数代表对哪一维进行操作，第二个参数用张量表示对该维哪些索引进行选取，这里第 1 个参数为零，代表对批次序号维进行操作，选索引号为 0、7、8 的元素，正好是第 1、8、9 张图片；
- 第 11、12 行：index_select 的第 1 个参数为 1，代表对颜色通道维进行操作，第 2 个参数为 0、2 则代表取 RGB 中的 R 和 B 通道数据；
- 第 13∼15 行：第一次 index_select 调用中，第 1 个参数为 2，代表从行数维取索引号在 0∼7 之间的数据，第二次 index_select 调用中，第 1 个参数为 3，代表从列数维取索引号在 0∼5 之间的数据，即取出所有图像左上角 8×6 的子图像；
- 第 16∼18 行：这里展示的是一种语法糖，用... 可以省略后面维度上取所有元素的: 号；
- 第 19 行：mask 的元素为：如果对应位置 X 的元素大于 0.5 则其值为 1，否则值为 0，即其是由 0 或 1 组成的张量，形状与 X 相同，表示 X 的每个元素是否大于 0.5；
- 第 20∼22 行：调用 masked_select 函数，将 X 中值大于 0.5 的元素以 1 维张量形式返回；
- 第 23 行：定义一个 2×3 的二维张量；
- 第 24、25 行：调用 take 方法，将张量 t2，在指定索引号处的元素以 1 维张量形式返回；

1.5 张量底层原理

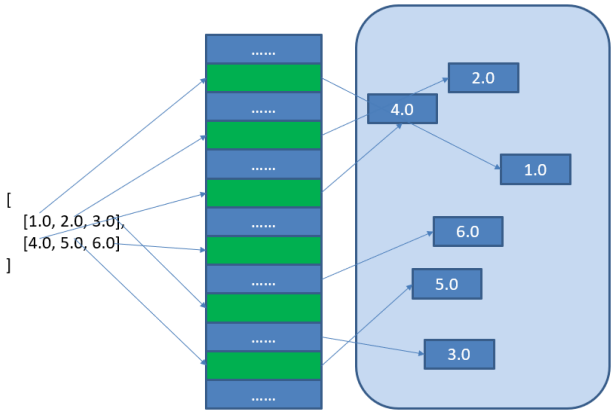
不知道大家有没有一个疑问，我们为什么不直接用 Python 中的列表，而是将其转换为张量在进行深度学习的运算呢？在本节中，我们简单介绍一下张量在内存中的表示，向大家展示，为什么张量的效率比 Python 的列表要高。
假设我们定义如下的列表：

```
t1 = [
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
]
```

Listing 9: Python 列表

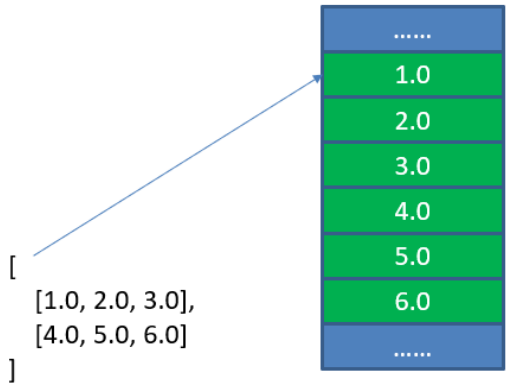
其在内存中的存储如下所示：

Figure 9: Python 列表内存存储



由上图可知，在 Python 中，数字也是以对象形式存储的。对于列表而言，每个元素对应于内存中的一个对象，对应着中间一列的内存地址，在该地址中，保存着实际值的地址，实际的数值保存在该地址中，所以对元素的获取，需要先找到元素对象地址，然后以该地址的内容作为内存地址，这样才能找到元素的具体值，需要两次寻址操作。
我们再来看 PyTorch 下张量的内存存储：

Figure 10: PyTorch 张量存储



张量在内存中以连续的内存单元存储，而且直接保存数值，当我们获取某个元素时，假设该张量保存在以 1000 开头的内存地址中，我们要获取 $t1[1][2]$ 元素，程序直接计算该元素的地址为：首地址 + idx1 * 阶 2 + idx2 = 1000 + 1 * 3 + 2 = 1005，这个位置的元素值为 6.0。由此可见，张量只需一步就可以获取到元素的值，而且改变张量形状后，获取元素时只需修改上面计算公式的参数即可，非常高效，而 Python 内置的列表对象，与此相比就要低效很多，所以我们在深度学习和其他科学计算任务中，很少使用 Python 内置的列表来表示数组，而是使用 PyTorch 的张量或 numpy 的 ndarray 来表示。
我们在这里只是简单的介绍了一下张量存储的底层原理，如果想要了解更多内容，请大家参考Eli Stevens [2019]。

1.6 总结

在本章中，我们向大家详细介绍了 PyTorch 中的张量（Tensor），介绍了张量的创建、基本属性、与 numpy 的互操作和基本操作，其中基本操作中介绍了改变形状、元素操作、Reduction、索引和切片操作，并且以深度学习中常用的方式进行了介绍，最后我们简要介绍了 PyTorch 中张量在内存中的存储，介绍了为什么张量比 Pyghon 内置列表要高效的原因。

在下一章中，我们将在此基础上，讲解自动微分技术，并且以张量和自动微分技术为基础，实现最简单的线性回归算法。

第 2 章自动微分

Abstract

在本章中我们将自动微分技术，并且将直接使用自动微分技术，来解决线性回归问题。

2 自动微分概述

记得深度学习三巨头之一 Yann LeCun 曾经说过：“深度学习已死，可微编程永生”，就是说深度学习只是一种计算范式，而背后的可微分编程，具有更广阔的应用前景。在这里我们将探索 PyTorch 中的自动微分技术。

2.1 数学原理

在本节中，我们将讲述自动微分的数学原理，我们在这里将以深度学习的典型应用场景为例，来讲解自动微分的数学原理。

2.1.1 函数以张量为参数

PyTorch 中的自动微分是基于 Tensor 的，以 Tensor 作为参数的函数，实际上是将 Tensor 中的每个元素应用该函数。只讲概念比较抽象，我们来看一个具体的例子。我们定义 $\boldsymbol{x} \in R^5$，定义如下函数：

$$y = f(\boldsymbol{x}) = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ f(x_5) \end{bmatrix} \tag{3}$$

下面的程序我们以 x^2 为例：

```
import torch
x = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0])
y = x ** 2
print(y)
# 输出结果为：tensor([ 1.,  4.,  9., 16., 25.])
```

Listing 10: 求 x^2 函数以张量为参数

在深度学习中，有三种最常见的情况：标量对向量的微分、向量对向量的微分、向量对矩阵的微分，其实还有张量对张量的微分，因为在实际中比较少见，所以这里就不再讨论了。

2.1.2 标量对向量微分

通常神经网络的代价函数会是一个标量，对于多分类问题，神经网络的输出为一个向量，这时就需要求标量对向量的微分。我们定义如下函数：

$$y = \sum_{i=1}^{5} x_i^2 \tag{4}$$

我们知道求微分可以采用解析法、数值法和自动微分法，由于这个问题比较简单，我们可以直接应用解析法：

$$\frac{\partial y}{\partial x_i} = 2x_i, \quad i = 1, 2, 3, 4, 5 \tag{5}$$

我们定义标量对向量的微分为：

$$\frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \frac{\partial y}{\partial x_3} & \frac{\partial y}{\partial x_4} & \frac{\partial y}{\partial x_5} \end{bmatrix} \in R^{1\times n}, \quad \boldsymbol{x} \in R^n \tag{6}$$

我们可以通过 PyTorch 中的自动微分来完成这一任务：

```
1 x = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0], requires_grad=True, dtype=
    torch.float32)
2 y = torch.sum(x ** 2)
3 print(y.item()) # 打印计算值
4 y.backward() # 利用自动微分计算微分
5 print('shape:{0}; {1}'.format(x.grad.size(), x.grad)) # 打印对的微分值yx
6 # 打印结果：shape:torch.Size([5]); tensor([ 2.,  4.,  6.,  8., 10.])
```

Listing 11: 标量对向量微分实现

代码解读如下：

- 第 1 行：我们在定义 Tensor 时，如果加入 requires_grad，表明我们需要求对其的微分；
- 第 2 行：计算函数值，在实际应用中就是神经网络的前向传播过程；
- 第 4 行：调用自动微分，就是实际应用中的神经网络的反向传播过程；

我们可以看到打印的结果与我们用解析法求出的内容一致。

2.1.3 向量对向量微分

实际上向量对向量微分叫做 Jacobian 矩阵，向量 $\boldsymbol{y} \in R^m$ 对向量 $\boldsymbol{x} \in R^n$ 的微分定义为：

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \in R^{m\times n} \tag{7}$$

我们假设有如下向量，例如是神经网络某层的输入向量：

$$\boldsymbol{z}^l = \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \\ 5.0 \end{bmatrix} \tag{8}$$

经过该层的激活函数（sigmoid 函数）之后，得到本层的输出向量：

$$\boldsymbol{a}^l = \begin{bmatrix} z_1^2 \\ z_2^2 \\ z_1^3 \\ z_1^4 \\ z_1^5 \end{bmatrix} \tag{9}$$

因为本神经元的输入只会影响本神经元的输出，因此我们的 $\boldsymbol{a}^l$ 如上所示。根据解析法可得其 Jacobian 矩阵为：

$$\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} = \begin{bmatrix} \frac{\partial a_1^l}{\partial z_1^l} & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial a_2^l}{\partial z_2^l} & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial a_3^l}{\partial z_3^l} & 0 & 0 \\ 0 & 0 & 0 & \frac{\partial a_3^l}{\partial z_3^l} & 0 \\ 0 & 0 & 0 & 0 & \frac{\partial a_3^l}{\partial z_3^l} \end{bmatrix} \tag{10}$$

其实际上只有对角线上有值，因此在 PyTorch 中，为了更高效的处理问题，我们通常不需要原始的 Jacobian 矩阵，而是将其乘以一个与 $\boldsymbol{a}^l$ 同维且元素均为 1 的向量，得到的结果向量的元素为对角线上的元素。

$$\frac{\partial \boldsymbol{a}^l}{\partial \boldsymbol{z}^l} \cdot \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} \frac{\partial a_1^l}{\partial z_1^l} & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial a_2^l}{\partial z_2^l} & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial a_3^l}{\partial z_3^l} & 0 & 0 \\ 0 & 0 & 0 & \frac{\partial a_3^l}{\partial z_3^l} & 0 \\ 0 & 0 & 0 & 0 & \frac{\partial a_3^l}{\partial z_3^l} \end{bmatrix} \cdot \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix} = \cdot \begin{bmatrix} \frac{\partial a_1^l}{\partial z_1^l} \\ \frac{\partial a_2^l}{\partial z_2^l} \\ \frac{\partial a_2^l}{\partial z_3^l} \\ \frac{\partial a_4^l}{\partial z_4^l} \\ \frac{\partial a_5^l}{\partial z_5^l} \end{bmatrix} \tag{11}$$

在 PyTorch 中代码如下所示：

```
x = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0], requires_grad=True)
y = x**2
y.backward(torch.ones_like(y))
print('y_x:{0}'.format(x.grad))
# 输出：y_x:tensor([ 2.,  4.,  6.,  8., 10.])
```

Listing 12: 向量对向量微分实现

2.1.4 向量对矩阵微分

在深度学习中，经常会出现上一层的输入向量 $\boldsymbol{z}^l \in R^{N_l}$ 对连接权值 $W^l \in R^{N_l \times N_{l-1}}$ 的微分，公式为：

$$\boldsymbol{z}^l = W^l \cdot \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \tag{12}$$

其中 $\boldsymbol{a}^{l-1} \in R^{N_{l-1}}$，$\boldsymbol{b}^l \in R^{N_l}$。
通过解析法，我们可以得到其微分公式为：

$$\frac{\partial \boldsymbol{z}^l}{\partial W^l} = \begin{bmatrix} a_1^{l-1} & a_2^{l-1} & \dots & a_{l-1}^{l-1} \\ a_1^{l-1} & a_2^{l-1} & \dots & a_{l-1}^{l-1} \\ \dots & \dots & \dots & \dots \\ a_1^{l-1} & a_2^{l-1} & \dots & a_{l-1}^{l-1} \end{bmatrix} = \begin{bmatrix} (\boldsymbol{a}^{l-1})^T \\ (\boldsymbol{a}^{l-1})^T \\ \dots \\ (\boldsymbol{a}^{l-1})^T \end{bmatrix} \in R^{N_l \times N_{l-1}} \tag{13}$$

PyTorch 代码如下所示：

```
a_l_1 = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0], requires_grad=True)
W_l = torch.tensor([
    [101.0, 102.0, 103.0, 104.0, 105.0],
    [201.0, 202.0, 203.0, 204.0, 205.0],
    [301.0, 302.0, 303.0, 304.0, 305.0]
], requires_grad=True)
b_l = torch.tensor([1001.0, 1002.0, 1003.0], requires_grad=True)
```

```
z_l = torch.matmul(W_l, a_l_1) + b_l
z_l.backward(torch.ones_like(z_l))
print('y_x:{0}'.format(W_l.grad))
''' 打印输出
y_x:tensor([[1., 2., 3., 4., 5.],
    [1., 2., 3., 4., 5.],
    [1., 2., 3., 4., 5.]])
'''
```

Listing 13: 向量对矩阵微分实现

由上面程序的运行结果可以看出，上述程序的打印结果与理论分析的结果一致，证明我们的求法是正确的。

有了这些知识之后，我们就可以利用这些知识，写出复杂的神经网络了，在本章中，我们将使用这些底层的技术，来解决简单的线性回归问题。但是我们在实际应用中，通常我们会用 PyTorch 包装的高阶 API 来实现，因此在“线性回归”章节，我们将采用 PyTorch 通常的方式来求解线性回归问题。线性回归的数学原理，也将在“线性回归”章节详细讲解。

2.2 自动微分求解线性回归

在这里我们只以一个简单的线性回归为例，分别采用 numpy 和 PyTorch 自动微分来求解。

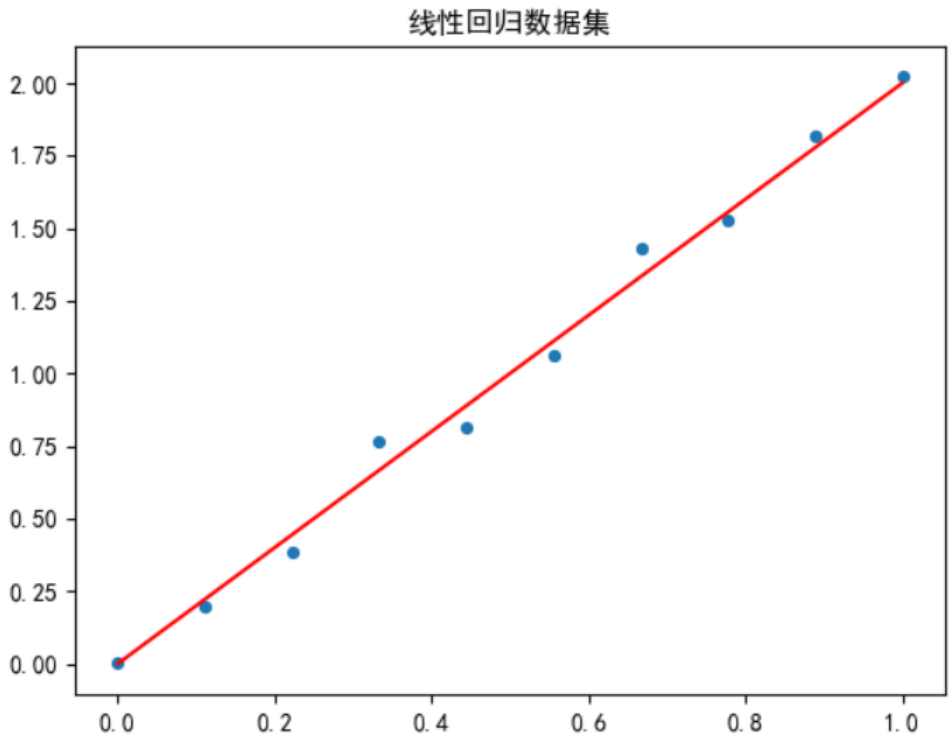
我们采用如下的程序生成一个数据集，如下所示：

```
    def load_dataset(self):
        delta = np.random.randn(10)*0.07
        X = np.linspace(0, 1, 10)
        y = 2*X + delta
        plt.rcParams['font.sans-serif'] = ['SimHei']
        plt.rcParams['axes.unicode_minus'] = False
        plt.title('线性回归数据集')
        plt.scatter(X, y, s=18)
        plt.plot(X, 2*X, '-r')
        plt.show()
        return X, y
```

Listing 14: 生成线性回归数据集

生成的图像如下所示：

Figure 11: 线性回归数据集



代码解读如下所示：

- 第 2 行：生成一个从标准正态分布 $(N)(0,1)$ 中取 10 个随机数，并将其缩小 0.7 倍；
- 第 3 行：X 为从 0~1 之间均匀取 10 个数；
- 第 4 行：求出 y 值，为了增加问题的复杂性，我们这里加入了一个随机项，使其看起来不是一个完美的直线；
- 第 5、6 行：设置绘图库的中文显示；
- 第 7 行：设置图像标题；
- 第 8 行：绘制散点图；
- 第 9 行：绘制目标直线，我们线性回归的目标；

2.2.1 使用 numpy 库

下面我们用 numpy 来实现线性回归算法，在本章中我们只讲算法实现，在下一章“线性回归”中，我们将讲解线性回归的数学原理。代码如下所示：

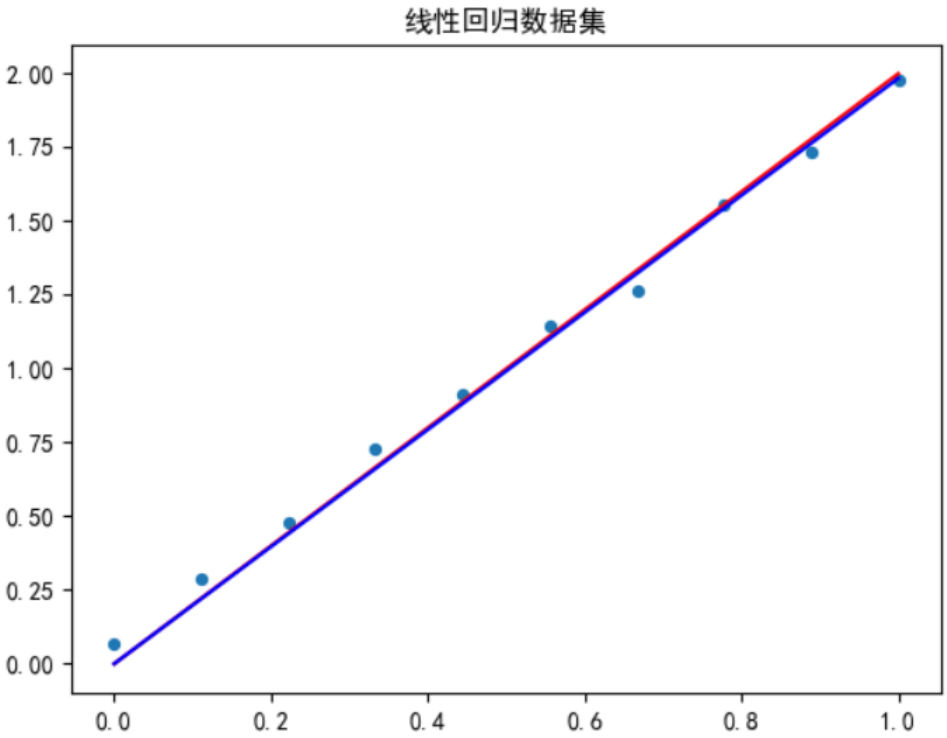
```
1     def run(self):
2         print('求解线性回归numpy')
3         X, y = self.load_dataset()
4         for epoch in range(100):
5             for xi, yi in zip(X, y):
6                 y_hat = self.forward(xi)
7                 loss = self.loss(xi, yi)
```

```
                grad = self.gradient(xi, yi, y_hat)
                self.w -= self.learning_rate * grad
                print('{0}: loss:{1}; w:{2}'.format(epoch, loss, self.w))
        print('The End ^_^')
        self.draw_dataset(X, y, self.w)

    def forward(self, x):
        return x * self.w

    def loss(self, x, y):
        y_hat = self.forward(x)
        return (y_hat - y) * (y_hat - y)

    def gradient(self, x, y, y_hat):
        return 2.0 * x * (y_hat - y)


    def load_dataset(self):
        delta = np.random.randn(10)*0.06
        X = np.linspace(0, 1, 10)
        y = 2*X + delta
        return X, y

    def draw_dataset(self, X, y, w):
        plt.rcParams['font.sans-serif'] = ['SimHei']
        plt.rcParams['axes.unicode_minus'] = False
        plt.title('线性回归数据集')
        plt.scatter(X, y, s=18)
        plt.plot(X, 2*X, '-r')
        plt.plot(X, w*X, '-b')
        plt.show()
```

Listing 15: numpy 求解线性回归问题代码

运行结果如下所示：

Figure 12: numpy 求解线性回归问题运行结果



代码解读如下所示：

- 第 3 行：载入上面所讲的线性回归数据集；
- 第 4 行：共进行 100 次迭代；
- 第 5 行：对每个样本执行 6~10 行；
- 第 6 行：调用 forward 方法（在后面讲解），求出该样本的输出值 $\hat{y}$；
- 第 7 行：以输入 $x^{(i)}$ 和正确值 $y^{(i)}$ 为参数，调用 loss 方法（在后面讲解），求出代价函数值；
- 第 8 行：以输入 $x^{(i)}$ 和正确值 y^{i} 和计算值 $\hat{y}^{(i)}$ 为参数，调用 gradient 方法（在后面讲解），求出对参数的微分；
- 第 9 行：由于我们要求的是代价函数的最小值，根据梯度下降算法调整参数：$w \leftarrow w - \alpha \cdot \nabla_w \mathcal{L}$；
- 第 10 行：打印训练过程；
- 第 12 行：绘制学习结果；
- 第 14、15 行：定义前向传播方法 forward，其计算公式为：$\hat{y}^{(i)} = w \cdot x^{(i)}$；
- 第 17~19 行：定义代价函数计算方法，公式为：$\mathcal{L}(w) = (\hat{y}^{(i)} - y^{(i)})^2$；

- 第 21~22 行：定义微分计算方法，公式推导如下：

$$\begin{aligned}\frac{\partial \mathcal{L}(w)}{\partial w} &= \frac{\partial(\hat{y}^{(i)} - y^{(i)})^2}{\partial w} \\ &= \frac{\partial(w \cdot x^{(i)} - y^{(i)})^2}{\partial w} \\ &= \frac{\partial(w \cdot x^{(i)} - y^{(i)})^2}{\partial(w \cdot x^{(i)} - y^{(i)})} \cdot \frac{\partial(w \cdot x^{(i)} - y^{(i)})}{\partial w} \\ &= 2(w \cdot x^{(i)} - y^{(i)}) \cdot \frac{\partial(w \cdot x^{(i)} - y^{(i)})}{\partial w} \\ &= 2x^{(i)}(w \cdot x^{(i)} - y^{(i)})\end{aligned} \tag{14}$$

我们来看最后我们学习得到的蓝色直线，与我们目标的红色直线非常接近，所以说明我们的算法可以很好的完成任务。

2.2.2 使用 PyTorch

下面我们利用 PyTorch 的张量和自动微分技术来求解这个线性回归问题，注意我们在实际应用中，我们通常不会采用这种方式，而是采用集成度更高的 PyTorch 风格来解决这个问题，我们将在下一章中进行详细讲解。代码如下所示：下面我们用 numpy 来实现线性回归算法，在本章中我们只讲算法实现，在下一章“线性回归”中，我们将讲解线性回归的数学原理。代码如下所示：

```
1     def __init__(self):
2         self.name = ''
3         self.w = Variable(torch.tensor([1.0]), requires_grad=True)
4         self.learning_rate = 0.01
5
6     def run(self):
7         X, y = self.load_dataset()
8         for epoch in range(100):
9             for xi, yi in zip(X, y):
10                y_hat = self.forward(xi)
11                loss = self.loss(xi, yi)
12                loss.backward()
13                #grad = self.gradient(x, y, y_hat)
14                self.w.data -= self.learning_rate * self.w.grad.data
15                self.w.grad.data.zero_()
16                print('{0}: loss:{1}; w:{2}'.format(epoch, loss.data[0],
    self.w))
17        self.draw_dataset(X.numpy(), y.numpy(), self.w.item())
18        print('The End ^_^')
19
20    def forward(self, x):
21        return x * self.w
22
23    def loss(self, x, y):
24        y_hat = self.forward(x)
25        return (y_hat - y) * (y_hat - y)
26
27    def gradient(self, x, y, y_hat):
```

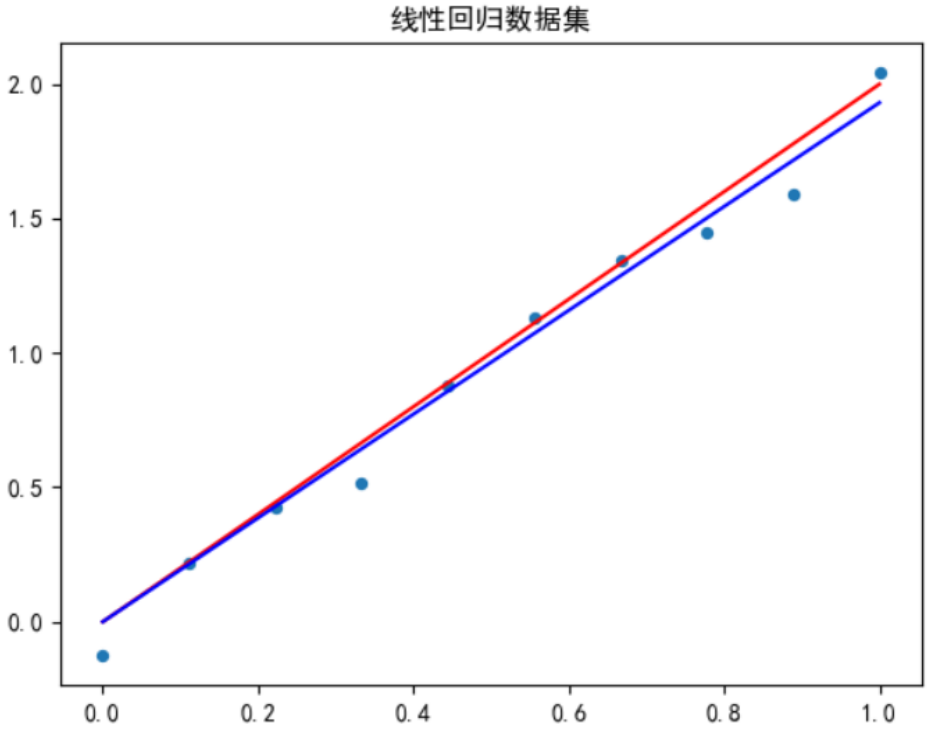
```
28          return 2.0 * x * (y_hat - y)
29
30      def load_dataset(self):
31          delta = np.random.randn(10)*0.06
32          X = np.linspace(0, 1, 10)
33          y = 2*X + delta
34          return torch.from_numpy(X), torch.from_numpy(y)
35
36      def draw_dataset(self, X, y, w):
37          plt.rcParams['font.sans-serif'] = ['SimHei']
38          plt.rcParams['axes.unicode_minus'] = False
39          plt.title('线性回归数据集')
40          plt.scatter(X, y, s=18)
41          plt.plot(X, 2*X, '-r')
42          plt.plot(X, w*X, '-b')
43          plt.show()
```

Listing 16: PyTorch 求解线性回归问题代码

代码解读如下所示：

- 第 3 行：在构造函数中定义权值变量，在 PyTorch 中，需要学习的参数，统一定义为 Variable；
- 第 4 行：定义的学习率；
- 第 7 行：载入我们前面介绍的数据集；
- 第 9 行：对于每个样本执行 10~16 行；
- 第 10 行：以输入样本 $x^{(i)}$ 为参数调用 forward 方法（在后面讲解），求出计算值 $\hat{y}^{(i)}$；
- 第 11 行：以输入信号 $x^{(i)}$ 和正确值 $y^{(i)}$ 为参数，调用 loss 方法（在后面讲解）求代价函数值；
- 第 12 行：调用 PyTorch 的反向传播方法，求对代价函数对网络参数的微分；
- 第 14 行：由于我们要求的是代价函数的最小值，根据梯度下降算法调整参数：$w \leftarrow w - \alpha \cdot \nabla_w \mathcal{L}$；
- 第 15 行：将参数的微分值清零，为下一个样本处理做准备；
- 第 16 行：打印训练进度信息；
- 第 17 行：打印训练后的结果图；

Figure 13: PyTorch 求解线性回归问题运行结果



如上图所示，学习到的直线如蓝色直线所示，与真实的红色直线的差别并不大，证明我们算法是正确的，如果增加迭代次数，可以使学习到的直线更加逼近真实的直线。

- 第 20、21 行：定义前向传播方法 forward，计算值为：$\hat{y}^{(i)} = w \cdot x^{(i)}$；
- 第 23~25 行：定义代价函数计算方法，公式为：$\mathcal{L}(w) = (\hat{y}^{(i)} - y^{(i)})^2$；
- 第 27、28 行：定义微分计算方法，公式推导如下：

$$\begin{aligned} \frac{\partial \mathcal{L}(w)}{\partial w} &= \frac{\partial (\hat{y}^{(i)} - y^{(i)})^2}{\partial w} \\ &= \frac{\partial (w \cdot x^{(i)} - y^{(i)})^2}{\partial w} \\ &= \frac{\partial (w \cdot x^{(i)} - y^{(i)})^2}{\partial (w \cdot x^{(i)} - y^{(i)})} \cdot \frac{\partial (w \cdot x^{(i)} - y^{(i)})}{\partial w} \\ &= 2(w \cdot x^{(i)} - y^{(i)}) \cdot \frac{\partial (w \cdot x^{(i)} - y^{(i)})}{\partial w} \\ &= 2x^{(i)}(w \cdot x^{(i)} - y^{(i)}) \end{aligned} \tag{15}$$

2.3 总结

在本章中，我们结合深度学习的常用场景，讲解了怎样在 PyTorch 中使用自动微分技术来实现这些功能。接着我们分别展示了怎样使用 numpy 库实现线性回归问题求解，然后展示仅使用 PyTorch 中的张量和自动微分求解线性回归问题。有了这些基础知识，我们实际上可以构造出各种复杂的神经网络。但是实际应用中，我们通常采用 PyTorch 的高阶 API

来实现这些功能。在下一章中，我们将向大家讲解怎样用 PyTorch 的标准方法来求解线性回归问题，同时我们也将详细讲解线性回归的数学原理。

第 3 章线性回归

Abstract

在本章中我们线性回归的数学原理，并利用 PyTorch 来求解线性回归问题。同时，我们会向大家进一步展示多项式回归，并讲解深度学习中的过拟合（over-fitting）问题。

3 线性回归概述

线性回归是最简单的一类机器学习算法，由于所要学习的函数是线性，而且是通过输入信号预测输出值，通常为一个标量，因此称之为线性回归。
本节的应用实例，我们选择了一个初创公司估值模型。为了形象化理解这一问题，假设我们要拟合的数据由如下公式产生：$y = 2.0 \times x_1 + 3.0 \times x_2 + 2.6$。训练数据如下所示：

Table 1: 线性回归问题数据集

x_1	x_2	y
0.54340494	0.2783694	4.521918
0.4245176	0.84477615	5.9833636
0.00471886	0.12156912	2.974145
0.67074907	0.82585275	6.419056
0.13670659	0.5750933	4.598693
0.89132196	0.20920213	5.01025
0.18532822	0.10837689	3.2957869
0.21969749	0.9786238	5.9752665
0.8116832	0.17194101	4.739189
0.81622475	0.27407375	5.054671

3.1 数学原理

我们这里采用闫涛，周琦 [2017] 书中的符号表示法，假设我们用 $\boldsymbol{x}^{(i)}$ 来表示第 i 个训练样本，训练样本共有 m 个：$\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(m)}\}$，对于每个训练样本有 n 维，如下所示：

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ ... \\ x_n^{(i)} \end{bmatrix} \tag{16}$$

训练数据集的格式为：$(\boldsymbol{x}^{(i)}, y^{(i)})$，即为表1中的一行。
因为我们现在假设我们研究的是一个线性回归问题，即 $\boldsymbol{x}^{(i)}$ 可由一个线性函数得到 $y^{(i)}$，我们可以假设这个线性函数为如下形式：

$$y^{(i)} = \boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b, \quad \boldsymbol{w} \in R^n, \quad b \in R \tag{17}$$

上式中 $\boldsymbol{w}$ 和 b 为参数，是需要我们从数据集中学习出来的模型参数。
上式用向量运算的形式表示，我们可以采用分量形式表示得更直观一些，以我们要研究的问题为例：

$$y^{(i)} = w_1 x_1^{(i)} + w_1 x_1^{(i)} + b, \quad \boldsymbol{w} \in R^n, \quad b \in R \tag{18}$$

为了衡量我们的模型好坏，对于单个样本，我们引入代价函数的概念：

$$l(\boldsymbol{\theta}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2 \tag{19}$$

我们将所有样本的代价函数定义为：

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 \tag{20}$$

我们在今后的章节中，还会引入代价函数 $\mathcal{J}(\boldsymbol{\theta})$，其定义为：$\mathcal{J}(\boldsymbol{\theta}) = \mathcal{J}(\boldsymbol{\theta}) + regualation$，即在式20上再加入避免过拟合（Over-Fitting）的调整项后的结果。
我们的目标是使总体代价函数越小越好，根据高等数学的相关知识，求一个函数的最小值，只需求该函数求对自变量的微分，令该微分为零，即可求出函数极值点时自动量的取值。在线性回归问题中，代价函数的自变量为 $\boldsymbol{w}$ 和 b，我们需要求对其的偏微分。求解这个问题有两种方法，一种是解析法，另一种是迭代法，下面我们分别来进行讲解。

3.1.1 迭代法求解

我们首先来看第 i 个样本，我们先来求代价函数对 j 个参数的导数：

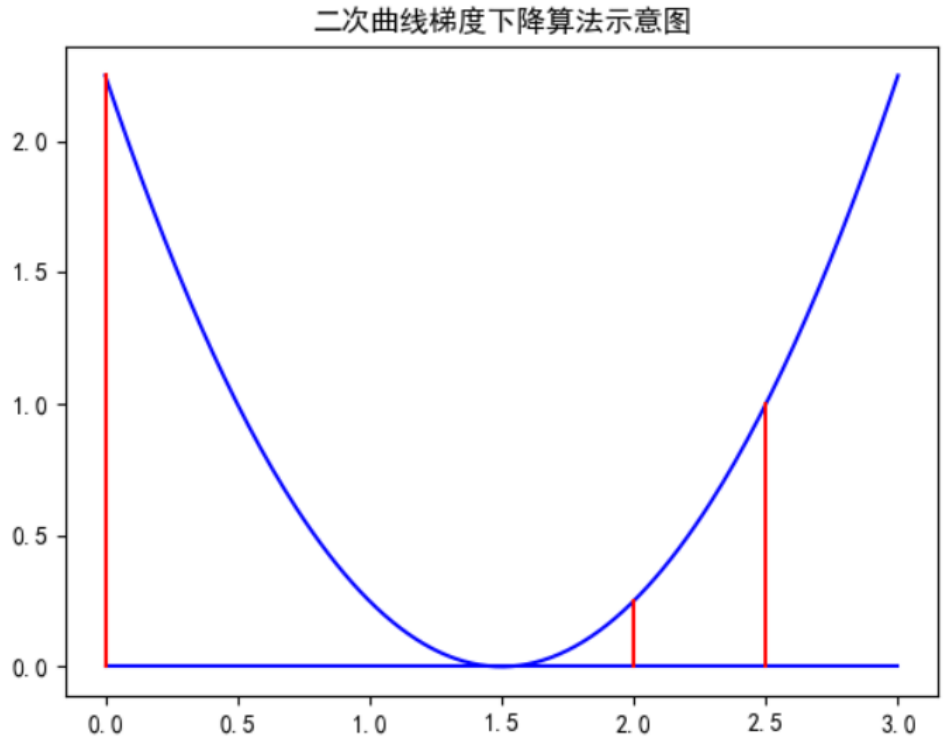
$$\begin{aligned}
\frac{\partial l^{(i)}(\boldsymbol{\theta})}{\partial \theta_j} &= \frac{\partial \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2}{\partial \theta_j} \\
&= \frac{1}{2}\frac{\partial(\hat{y}^{(i)} - y^{(i)})^2}{\partial(\hat{y}^{(i)} - y^{(i)})}\frac{\partial(\hat{y}^{(i)} - y^{(i)})}{\partial \theta_j} \\
&= (\hat{y}^{(i)} - y^{(i)})\frac{\partial(\theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + ... + \theta_j x_j^{(i)} + ... + \theta_n x_n^{(i)})}{\partial \theta_j} \\
&= (\hat{y}^{(i)} - y^{(i)})x_j^{(i)}
\end{aligned} \tag{21}$$

注意：在上式中 θ_0 即为公式中的 b。而对一个批次中所有样本，微分定义为：

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\theta_j} = \sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})x_j^{(i)} \tag{22}$$

下面我们简单讲一下梯度下降算法，我们以求一个二次曲线极小值为例来讲解梯度下降算法。假设我们要求的二次曲线为：$f(x) = (x - 1.5)^2$ 的图形为：

Figure 14: 二次曲线梯度下降算法示意图



图形绘制代码见 (resources/book/chp002/e1/chp002_e1_c002.py) 我们求出该函数微分：$f'(x) = 2(x - 1.5)$，求当 $x_0 = 2.5$ 时，$f(x_0) = 1$，微分值为 $f'(x_0) = 2$，表明在该邻域内是单调增加的，想要求极小值，就要减少 $x_0 - \delta x$ 的值，而此时梯度的值为正值，就是需要减去梯度，例如，当 $\delta x = 0.5$ 时，此时函数值为 $f(x_0 - \delta x) = f(2) = 0.25$。但是我们也不能减小的太多，例如我们取 $\delta x = 2.5$，此时 $x_0 - \delta x = 0$，此时函数的值为 $f(x_0 - \delta x) = f(0) = 2.25$，这时值反而会增大了，因此我们需要在微分前乘以一个小的正数，我们称之为学习率，在公式中用 α 来表示。
我们可以得到参数调整公式为：

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\theta_j} \tag{23}$$

3.1.2 解析法求解

迭代法比较适合复杂的应用场景，对于简单的应用场景，例如我们现在所研究的线性回归算法而言，解析法是更高效的解决方案。所以，在这里向读者介绍一下解析法的推导过程，也帮助读者熟悉一下深度学习中所用到的数学知识。首先介绍向量求导符号：

$$\nabla_{\boldsymbol{\theta}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \left[\frac{\partial \mathcal{L}}{\partial \theta_0} \quad \frac{\partial \mathcal{L}}{\partial \theta_1} \quad \frac{\partial \mathcal{L}}{\partial \theta_2} \quad \dots \quad \frac{\partial \mathcal{L}}{\partial \theta_n} \in R^{1 \times n+1} \right] \tag{24}$$

则参数更新公式可以表示为：

$$\boldsymbol{\theta}^T \leftarrow \boldsymbol{\theta}^T - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L} \tag{25}$$

我们定义以矩阵为参数的函数 $f: R^{m\times n} \to R^{m\times n}$：

$$f(A) = \begin{bmatrix} f(A_{1,1}) & f(A_{1,2}) & ... & f(A_{1,n}) \\ f(A_{2,1}) & f(A_{2,2}) & ... & f(A_{2,n}) \\ ... & & & \\ f(A_{m,1}) & f(A_{m,2}) & ... & f(A_{m,n}) \end{bmatrix}, A \in R^{m\times n} \tag{26}$$

我们可以定义对矩阵的微分为：

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{1,1}} & \frac{\partial f(A)}{\partial A_{1,2}} & \cdots & \frac{\partial f(A)}{\partial A_{1,n}} \\ \frac{\partial f(A)}{\partial A_{2,1}} & \frac{\partial f(A)}{\partial A_{2,2}} & \cdots & \frac{\partial f(A)}{\partial A_{2,n}} \\ ... & & & \\ \frac{\partial f(A)}{\partial A_{m,1}} & \frac{\partial f(A)}{\partial A_{m,2}} & \cdots & \frac{\partial f(A)}{\partial A_{m,n}} \end{bmatrix} \tag{27}$$

对于矩阵 $A \in R^{m\times n}$ 的迹定义为：

$$trA = \sum_{i=1}^{n} A_{i,i} \tag{28}$$

矩阵的迹具有如下性质：

- $trAB = trBA$
- $trABC = trCAB = trBCA$
- $\nabla_A trAB = B^T$
- $trA = trA^T$
- 对 $a \in R$，则 $tr(a) = a$
- $\nabla_A trABA^TC = CAB + C^TAB^T$

这些关于矩阵迹的性质将在后续的公式推导中用到，先在这里罗列出来，其实这些性质都是可以证明的，但是因为本书重点不是讲解数学知识，因此对证明过程感兴趣的读者可以查看其他相关资料。

下面把 m 个 n 维的训练样本向量组成设计矩阵（Design Matrix）：

$$X = \begin{bmatrix} \left(\boldsymbol{x}^{(1)}\right)^T \\ \left(\boldsymbol{x}^{(2)}\right)^T \\ ... \\ \left(\boldsymbol{x}^{(m)}\right)^T \end{bmatrix} \tag{29}$$

则线性回归的公式可以表示为如下所示的形式：

$$\hat{\boldsymbol{y}} = X \cdot \boldsymbol{\theta} = \begin{bmatrix} \left(\boldsymbol{x}^{(1)}\right)^T \\ \left(\boldsymbol{x}^{(2)}\right)^T \\ ... \\ \left(\boldsymbol{x}^{(m)}\right)^T \end{bmatrix} \cdot \boldsymbol{\theta} = \begin{bmatrix} \left(\boldsymbol{x}^{(1)}\right)^T \cdot \boldsymbol{\theta} \\ \left(\boldsymbol{x}^{(2)}\right)^T \cdot \boldsymbol{\theta} \\ ... \\ \left(\boldsymbol{x}^{(m)}\right)^T \cdot \boldsymbol{\theta} \end{bmatrix} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ ... \\ \hat{y}^{(m)} \end{bmatrix} \tag{30}$$

而正确结果可以表示为：

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ ... \\ y^{(m)} \end{bmatrix} \tag{31}$$

向量转置与向量的乘积为一个标量，公式为：

$$\boldsymbol{y}^T\boldsymbol{a} = \sum_{i=1}^{n} a_i^2 \tag{32}$$

我们根据这个公式推导出线性回归的代价函数公式：

$$\frac{1}{2}(\hat{\boldsymbol{y}} - \boldsymbol{y})^T(\hat{\boldsymbol{y}} - \boldsymbol{y}) = \frac{1}{2}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 \tag{33}$$

上式就是我们前面定义的线性回归代价函数公式。因此线性回归的公式可以变为：

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}(X\cdot\boldsymbol{\theta} - \boldsymbol{y})^T(X\cdot\boldsymbol{\theta} - \boldsymbol{y}) \tag{34}$$

我们现在的任务是求代价函数的最小值，根据高等数学的知识可知，只需要对其求微分，使其值为零即可。

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}}\frac{1}{2}(X\cdot\boldsymbol{\theta} - \boldsymbol{y})^T(X\cdot\boldsymbol{\theta} - \boldsymbol{y}) && (1)\\
&= \frac{1}{2}\nabla_{\boldsymbol{\theta}} tr\Big((X\cdot\boldsymbol{\theta} - \boldsymbol{y})^T(X\cdot\boldsymbol{\theta} - \boldsymbol{y})\Big) && (2)\\
&= \frac{1}{2}\nabla_{\boldsymbol{\theta}} tr\Big((\boldsymbol{\theta}^T X^T - \boldsymbol{y}^T)(X\cdot\boldsymbol{\theta} - \boldsymbol{y})\Big) && (3)\\
&= \frac{1}{2}\nabla_{\boldsymbol{\theta}} tr(\boldsymbol{\theta}^T X^T X\cdot\boldsymbol{\theta} - \boldsymbol{\theta}^T X^T\boldsymbol{y} - \boldsymbol{y}^T X\cdot\boldsymbol{\theta} + \boldsymbol{y}^T\boldsymbol{y}) && (4)\\
&= \frac{1}{2}\nabla_{\boldsymbol{\theta}}\Big(tr(\boldsymbol{\theta}^T X^T X\cdot\boldsymbol{\theta}) - tr(\boldsymbol{\theta}^T X^T\boldsymbol{y}) - tr(\boldsymbol{y}^T X\cdot\boldsymbol{\theta})\Big) && (5)\\
&= \frac{1}{2}\nabla_{\boldsymbol{\theta}}\Big(tr(\boldsymbol{\theta}\boldsymbol{\theta}^T X^T X) - tr(\boldsymbol{\theta}^T X^T\boldsymbol{y}) - tr(\boldsymbol{y}^T X\cdot\boldsymbol{\theta})\Big) && (6)\\
&= \frac{1}{2}\Big(\nabla_{\boldsymbol{\theta}} tr(\boldsymbol{\theta}\boldsymbol{\theta}^T X^T X) - 2\nabla_{\boldsymbol{\theta}} tr(\boldsymbol{y}^T X\boldsymbol{\theta})\Big) && (7)\\
&= \frac{1}{2}\Big((X^T X\boldsymbol{\theta} I + X^T X\boldsymbol{\theta} I^T) - 2\nabla_{\boldsymbol{\theta}} tr(\boldsymbol{y}^T X\boldsymbol{\theta})\Big) && (8)\\
&= \frac{1}{2}\Big(2X^T X\boldsymbol{\theta} - 2\nabla_{\boldsymbol{\theta}} tr(\boldsymbol{y}^T X\boldsymbol{\theta})\Big) && (9)\\
&= X^T X\boldsymbol{\theta} - \nabla_{\boldsymbol{\theta}} tr(\boldsymbol{y}^T X\boldsymbol{\theta}) && (10)\\
&= X^T X\boldsymbol{\theta} - X^T\boldsymbol{y} = 0
\end{aligned} \tag{35}$$

所以可以得到取得极小值时的参数为：

$$\boldsymbol{\theta} = (X^T X)^{-1} X^T\boldsymbol{y} \tag{36}$$

上面的公式推导有些复杂，下面我们来做一下简单的解释：

- 第 1 行：直接使用式34；
- 第 2 行：由于代价函数是一个标量，而标量的迹等于标量，因此可以对其取迹运算而值不变；
- 第 3 行：把整体的转置分解到各项，差的转置等于转置的差，另外用到转置的一个性质：$(AB)^T = B^T A^T$；
- 第 4 行：将多项式乘积展开；
- 第 5 行：由于我们是对 $\boldsymbol{\theta}$ 求微分，最后一项 $\boldsymbol{y}^T\boldsymbol{y}$ 为常量，因此微分为零被消去；另外迹的和差等于和差的迹；
- 第 6 行：用到迹的性质：$trABC = trCAB$；

- 第 7 行：根据迹的性质：$trA^T = trA$，可以合并后面两项；
- 第 8 行：根据公式 $\nabla_A trABA^TC = CAB + C^TAB^T$，我们设 $A = \boldsymbol{\theta}$ 和 $B = I$，其中 I 为单位阵，直接应用公式即可得到此式；
- 第 9 行：单位阵乘以矩阵，结果还是该矩阵；
- 第 10 行：将 2 与 $\frac{1}{2}$ 约掉；
- 第 11 行：根据公式 $\nabla_A trAB = B^T$，直接应用公式即可得到本式；

上百就是用最小二乘法求出的参数向量的解析解的推导过程，但是要求 X^TX 可逆，在一般情况下，X^TX 确实是可逆的。但是如果特征存在相互依赖的关系，此时 X^TX 就不可逆了，例如 $x_1 = 2x_2$，此时表明只有一个特征，而不是两个特征，因此就会出现不可逆情况。我们在实际应用中，很少用到解析法，所以在这里大家仅需要了解一下即可。

3.2 线性回归求解

我们首先通过程序生成前一节的数据集，然合利用 PyTorch 的 Linear 模型，来求解这个问题，求解完毕后，我们看一下求出的参数是否与真实参数吻合，同时给出一个例子，由模型进行预测，看预测的结果是否正确。
我们先来看模型的定义：

```
1  class LinearRegressionModel(torch.nn.Module):
2      def __init__(self, in_features=1, out_features=1):
3          super(LinearRegressionModel, self).__init__()
4          self.linear = torch.nn.Linear(in_features=in_features,
5                        out_features=out_features, bias=True)
6
7      def get_weights(self):
8          return self.linear.weight
9
10     def get_biases(self):
11         return self.linear.bias
12
13     def forward(self, x):
14         return self.linear(x)
```

Listing 17: 线性模型定义（resources/book/chp002/e1/linear_regression_model.py）

代码解读如下：

- 第 1 行：定义 LinearRegressionModel 类，并继承 torch.nn.Module 类；
- 第 4、5 行：初始化线性模型，指定输入信号维度 in_features，输出信号维度 out_features，指定采用偏置值，即公式 $\hat{y}^{(i)} = \boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b$ 中的 b 不为零；
- 第 7、8 行：获取模型的权值，其值为张量；
- 第 10、11 行：获取模型的偏置值，其值为张量；
- 第 13、14 行：定义前向传播算法，在 PyTorch 中只需要定义前向传播算法，反向传输通过上一章的自动微分技术自动求出；

接下来我们来看怎样使用这个类：

```
1  class LinearRegressionApp(object):
2      def __init__(self):
3          self.name = ''
4
5      def load_dataset(self):
6          np.random.seed(100)
7          X0 = np.random.rand(10, 2)
8          X = np.array(X0, dtype=np.float32)
9          w = np.array([2.0, 3.0], dtype=np.float32)
10         b = 2.6
11         y = np.matmul(X, w) + b
12         y = y.reshape(10, 1)
13         print('X:{0}; \r\ny:{1}'.format(X, y))
14         return Variable(torch.from_numpy(X)), \
15                     Variable(torch.from_numpy(y))
16
17     def run(self):
18         X_train, y_train = self.load_dataset()
19         model = LinearRegressionModel(dim_in=2, dim_out=1)
20         criterion = torch.nn.MSELoss(size_average=False)
21         optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
22         #optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
23         for epoch in range(1000):
24             y_hat = model(X_train)
25             loss = criterion(y_hat, y_train)
26             if epoch % 50 == 0:
27                 print('{0}: {1}'.format(epoch, loss.data.item()))
28             optimizer.zero_grad()
29             loss.backward()
30             optimizer.step()
31         w = model.get_weights().data
32         bias = model.get_biases().data
33         print('w:{0}; bias:{1}'.format(w, bias))
34
35
36         x1 = Variable(torch.tensor([[4.0, 5.0]]))
37         y1_hat = model.forward(x1).data[0][0]
38         print('y={0}'.format(y1_hat))
```

Listing 18: 线性回归示例（resources/book/chp002/e1/linear_regression_app.py）

代码解读如下所示：

- 第 5 行：定义生成数据集函数，生成表1所示数据；
- 第 6 行：让 numpy 每次生成的随机数相同，便于进行程序调试；
- 第 7 行：从 0~1 的均匀分布中采样随机数，生成共有 10 个样本组成，每个样本为 2 维向量的 ndarray；
- 第 8 行：将数据类型由 np.float64 变为 np.float32；
- 第 9 行：设置权值参数的真值；

- 第 10 行：设置偏置值的真值；
- 第 11 行：根据公式：$\boldsymbol{y} = X \cdot \boldsymbol{w} + b$，因为 $X \in R^{10\times 2}$，$\boldsymbol{w} \in R^{2\times 1}$，$b \in R$，所以 $\boldsymbol{y} \in R^{10}$；
- 第 12 行：将真值的维度由 R^{10} 变为 $R^{10\times 1}$；
- 第 14、15 行：将 X 和 y 转换为 PyTorch 的变量类型并返回；
- 第 18 行：载入数据集；
- 第 19 行：生成线性回归模型对象，设置输入信号维度为 2，输出信号维度为 1；
- 第 20 行：设置代价函数为 MSE，即 $\mathcal{L} = \sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$；
- 第 21 行：指定优化器为随机梯度下降算法；
- 第 23 行：循环执行 23∼30 行 1000 次；
- 第 24 行：通过前向传播求出 $\hat{Y}^{(i)}$；
- 第 25 行：求出代价函数值；
- 第 26、27 行：每隔 50 遍打印一次训练进展；
- 第 28 行：清空网络参数的微分值；
- 第 29 行：调用反向传播方法，利用 PyTorch 自动微分技术求出各参数的微分；
- 第 30 行：利用优化器更新参数值；
- 第 31 行：训练完成之后，取出模型的权值参数 $\boldsymbol{w}$；
- 第 32 行：取出偏置值参数 b；
- 第 33 行：打印权值和偏置值；
- 第 36 行：生成一个测试用新的输入信号；
- 第 37 行：利用前向传播求出网络的输出值；
- 第 38 行：打印出该输出值；

程序运行结果如下所示：

Figure 15: 线性回归运行结果

```
0: 291.85650634765625
50: 0.40368902683258057
100: 0.11347219347953796
150: 0.034133970737457275
200: 0.01053057424724102
250: 0.0032775306608527 9
300: 0.0010231807827949524
350: 0.0003197435289621353
400: 9.994146967073902e-05
450: 3.124777867924422e-05
500: 9.76708633970702e-06
550: 3.0548148970410693e-06
600: 9.546247383696027e-07
650: 2.9859216965633095e-07
700: 9.34920194595179e-08
750: 2.952839395220508e-08
800: 9.403322565049166e-09
850: 3.0477167456410825e-09
900: 1.0114717952092178e-09
950: 2.639808371895924e-10
w:tensor([[2.0000, 3.0000]]); bias:tensor([2.6000])
y=25.599933624267578
```

由运行结果可以看出，我们模型成功学出了所有的权值和偏置值，也可以预测新的样本值，证明我们线性回归应用是正确的。

3.3 多项式回归

在前面的章节的学习中，我们知道，线性回归实际上拟合的直线，而实际应用中，如果不是直线，如果是曲线的话要怎么处理呢？这个问题有两种处理思路：第一种是将其划分为足够多的段，而每段都可以用直线来近似，这样就将问题转化为多个线性回归的问题，就可以用我们前面章节介绍的技术来处理；第二种方法就是本节将介绍的多项式回归。在讲解具体的多项式回归算法之前，我们先来研究一下为什么多项式回归可以拟合任意的曲线。

我们知道在高等数学中利用泰勒公式，可以将任意函数表示为多项式形式：

$$\begin{aligned} f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \\ \frac{f^{(3)}(x_0)}{3!}(x - x_0)^3 + ... + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + ... \\ = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \end{aligned} \tag{37}$$

如果将 $x_0 = 0$，上式就变为泰勒级数：

$$\begin{aligned} f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \\ \frac{f^{(3)}(0)}{3!}x^3 + ... + \frac{f^{(n)}(0)}{n!}x^n + ... \\ = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!}x^n \end{aligned} \tag{38}$$

这就是我们可以使用多项式回归来拟合任意函数的理论依据，你要所拟合的函数无限可微，我们就可以以任意精度来拟合的这个函数。

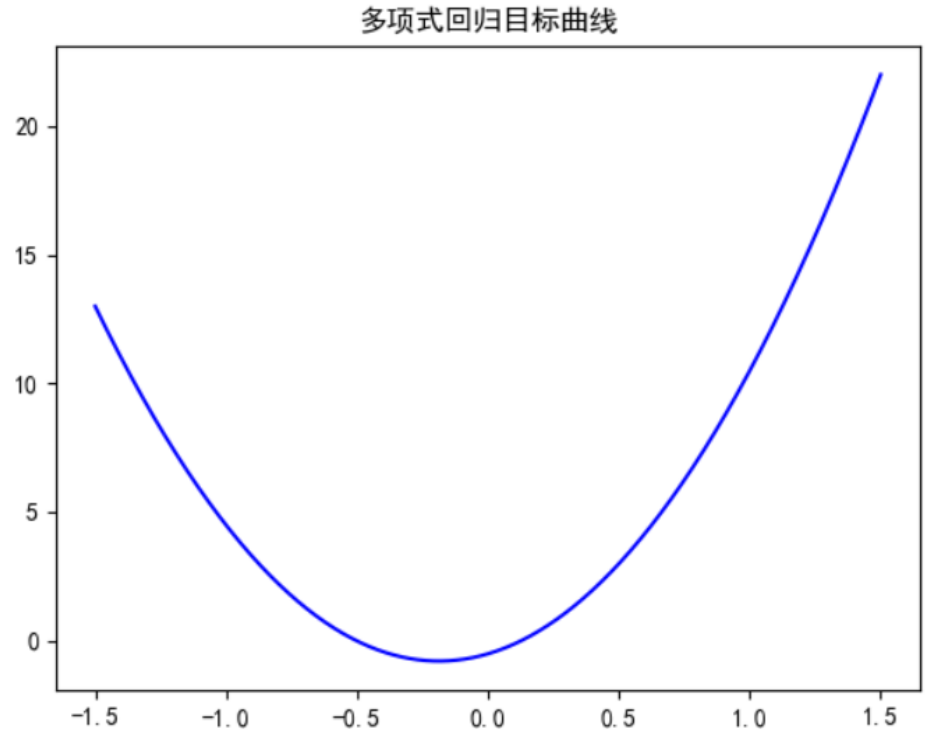
下面我们来看我们要拟合的曲线：

```
    def draw_curve(self):
        x = np.linspace(-1.5, 1.5, 1000)
        y = self.target_func(x)
        plt.rcParams['font.sans-serif'] = ['SimHei']
        plt.rcParams['axes.unicode_minus'] = False
        plt.title('二次曲线梯度下降算法示意图')
        plt.plot(x, y, '-b')
        plt.show()

    def target_func(self, x):
        return (x + 0.5)*(8.0*x - 1.0)
```

Listing 19: 多项式回归示例（resources/book/chp002/e2/polynomial_regression_app.py）

由于程序比较简单，我们就不讲解代码本身了，其定义的图形如下所示：

Figure 16: 多项式回归目标曲线图



代码如下所示：

```
class PolynomialRegressionApp(object):
    def __init__(self):
        self.name = ''
        self.low_limit = -0.5
        self.high_limit = 0.5
        plt.rcParams['font.sans-serif'] = ['SimHei']
        plt.rcParams['axes.unicode_minus'] = False
        plt.title('多项式回归目标曲线')

    def run(self):
        rank = 9
        X_train, y_train = self.load_dataset(rank=rank)
        model = PolynomialRegressionModel(in_features=rank, out_features
    =1)
        criterion = torch.nn.MSELoss(size_average=False)
        optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
        #optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
        for epoch in range(8000):
            y_hat = model(X_train)
            loss = criterion(y_hat, y_train)
            if epoch % 50 == 0:
                print('{0}: {1}'.format(epoch, loss.data.item()))
            optimizer.zero_grad()
```

```
            loss.backward()
            optimizer.step()
        w = model.get_weights().data
        bias = model.get_biases().data
        print('w:{0}; bias:{1}'.format(w, bias))
        # 绘制学习的曲线
        test_data_x, test_np_x = self.generate_test_data(rank)
        test_input_x = Variable(torch.tensor([test_np_x]))
        test_y_hat = model.forward(test_input_x)
        y_raw = test_y_hat.detach().numpy()
        y_hat = y_raw.reshape(y_raw.shape[1])
        plt.plot(test_data_x, y_hat, '-r')
        # 绘制数据散点图
        data_x = X_train[:, :1].flatten()
        data_y = self.target_func(data_x)
        plt.scatter(data_x, data_y)
        plt.show()

    def generate_test_data(self, rank):
        raw_data = np.linspace(self.low_limit,
                               self.high_limit, 100)
        Xt1 = raw_data.reshape(raw_data.shape[0], 1)
        Xt2 = Xt1 * Xt1
        Xt3 = Xt2 * Xt1
        Xt4 = Xt3 * Xt1
        Xt5 = Xt4 * Xt1
        Xt6 = Xt5 * Xt1
        Xt7 = Xt6 * Xt1
        Xt8 = Xt7 * Xt1
        Xt9 = Xt8 * Xt1
        if 1 == rank:
            Xt = Xt1
        elif 2 == rank:
            Xt = np.hstack((Xt1, Xt2))
        else:
            Xt = np.hstack((Xt1, Xt2, Xt3, Xt4, Xt5,
                            Xt6, Xt7, Xt8, Xt9))
        return raw_data, np.array(Xt, dtype=np.float32)

    def load_dataset(self, rank=1):
        np.random.seed(100)
        X1 = np.random.rand(10, 1) * (self.high_limit - \
                               self.low_limit) - (self.high_limit - \
                               self.low_limit) / 2.0
        X2 = X1 * X1
        X3 = X2 * X1
        X4 = X3 * X1
        X5 = X4 * X1
        X6 = X5 * X1
        X7 = X6 * X1
```

```
73          X8 = X7 * X1
74          X9 = X8 * X1
75          if 1 == rank:
76              X_raw = X1
77          elif 2 == rank:
78              X_raw = np.hstack((X1, X2))
79          else:
80              X_raw = np.hstack((X1, X2, X3, X4, X5,
81                                 X6, X7, X8, X9))
82          X = np.array(X_raw, dtype=np.float32)
83          b = 0.0
84          y0 = self.target_func(X1)
85          y = np.array(y0, dtype=np.float32)
86          return Variable(torch.from_numpy(X)), \
87                      Variable(torch.from_numpy(y))
```

Listing 20: 多项式回归示例（resources/book/chp002/e2/polynomial_regression_app.py）

代码解读如下所示：

- 第 4、5 行：规定数据集的下限和上限；
- 第 6~8 行：设置 matplotlib 库可以显示中文和负号；
- 第 11 行：设置多项式回归的最高项数，我们试验了 1、2、9 次方；
- 第 12 行：载入数据集（方法将在后面讲解）；
- 第 13 行：创建多项式回归模型，对于 $rank = 1$ 时，$in_features = 1$，$rank = 2$ 时 $in_features = 2$，当 $rank = 9$ 时 $in_features = 9$；
- 第 14 行：定义平方误差函数为代价函数；
- 第 15 行：采用随机梯度下降算法作为优化方法；
- 第 17 行：循环执行 18~24 行代码 8000 次；
- 第 18 行：调用模型的 forward 方法求出网络输出值 $\hat{Y}$；
- 第 19 行：计算代价函数值；
- 第 20、21 行：每隔 50 次迭代打印一次训练进度；
- 第 22 行：清空网络参数的微分值；
- 第 23 行：调用 PyTorch 自动微分反向传播算法网络参数微分；
- 第 24 行：调用优化器根据参数微分更新参数值；
- 第 25 行：获取训练好模型的权值参数 $\boldsymbol{w}$；
- 第 26 行：获取训练好模型的偏置值 b；
- 第 27 行：打印权值和偏置值；
- 第 29 行：调用 generate_test_data 方法（将在后面介绍）生成原始的数据和按照多项式最高阶人工生成的数据；
- 第 30 行：将阶数对应的数据转化为 PyTorch 的张量；
- 第 31 行：调用模型 forward 方法求出网络输出值；
- 第 32、33 行：取出结果张量将基转换为 numpy 数据且将形状变为 R^m；

- 第 34 行：绘制学习出的曲线；
- 第 36 行：取出训练数据集第一列即一次项对应的数据；
- 第 37 行：调用目标函数求出正确值；
- 第 38 行：将训练数据集以散点图形式绘制出来；

下面我们来看生成训练数据集方法：

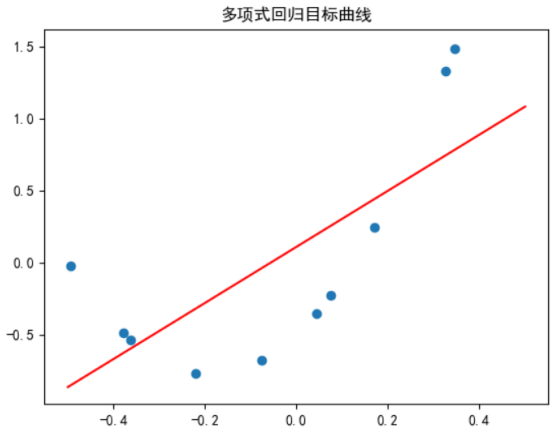
- 第 62 行：定义生成训练数据集方法；
- 第 63 行：设置随机数生成种子，使每次生成的随机数固定，便于进行代码调试；
- 第 64∼66 行：从 0 到 1 的均匀分布中抽样随机数形成形状为 $R^{10\times1}$ 的数组，并将其范围转换为 self.low_limit 到 self.high_limit 之间的数；
- 第 67∼74 行：生成 $x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9$ 次方的数组；
- 第 75、76 行：如果多项式最高次为 1 次时，只取 x 为训练样本集；
- 第 77、78 行：如果多项式最高次为 2 次时，取 x, x^2 两项，组成形状为 $R^{10\times2}$ 的数组；
- 第 79、81 行：如果多项式最高次为 9 次时，取 $x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9$ 等项，形成形状为 $R^{10\times9}$ 的数组；
- 第 82 行：将数组改为 float32 类型；
- 第 83 行：设置偏置值；
- 第 84 行：根据我们的目标曲线 $y = (x + 0.5)(8x - 1) = 3x + 8x^2 - 0.5$，计算样本点的目标值；
- 第 85 行：将目标值转换为 float32 类型；
- 第 86、87 行：将样本值和目标值数组转换为 PyTorch 变量类型并返回；

下面我们来看测试数据的生成：

- 第 41 行：定义生成测试数据的方法；
- 第 42、43 行：生成由 self.low_limit 到 self.high_limit 之间均匀分布的 100 个数据点；
- 第 44 行：将数组的形状改为 $R^{100\times1}$，作为 1 次项特征数值；
- 第 45∼52 行：生成 $x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9$ 次方项的特征数值；
- 第 53、54 行：如果多项式最高次为 1 次，取 1 次项特征值；
- 第 55、56 行：如果多项式最高次为 2 次，取 1 次和 2 次项特征值；
- 第 57、58 行：如果多项式最高次为 9 次，取 1 次、2 次、3 次、4 次、5 次、6 次、7 次、8 次、9 次项特征值；
- 第 59 行：返回原始采样点数据和考虑最高次后的数据；

当多项式最高次为 1 次时运行结果如下所示：

Figure 17: 多项式回归最高次为 1 次运行结果图



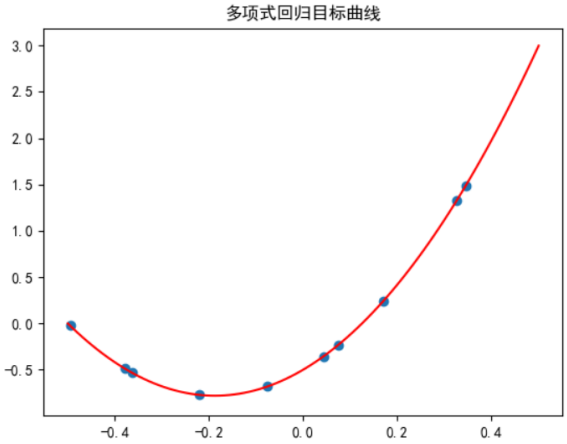
参数值为：

Figure 18: 多项式回归最高次为 1 次学习的参数值

```
7950: 2.7421655654907227
w:tensor([[1.9471]]); bias:tensor([0.1092])
```

由上图可以看出，只用一条直线，不可能拟合出图中的数据点。这就是深度学习中的欠拟合（Under-Fitting）问题，当模型的表现能力低于数据集的实际分布函数时，就会出现这种情况。例如在本例中，拟合的函数为二次，我们硬要使用一次函数来拟合，结果就是不能很好的拟合。
当最高次为 2 次时，运行结果如下所示：

Figure 19: 多项式回归最高次为 2 次运行结果图



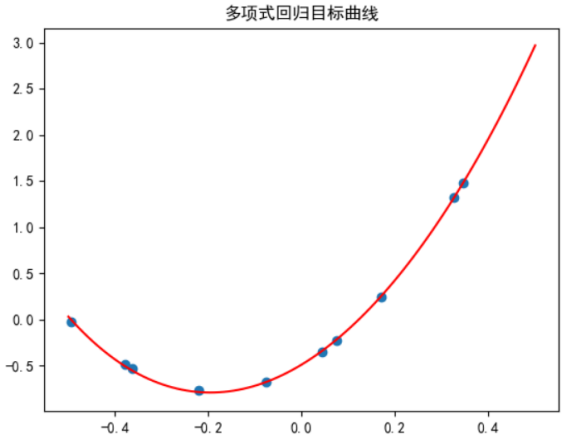
学习到的参数为：

Figure 20: 多项式回归最高次为 2 次学习的参数值

```
7900: 5.692639206245076e-06
7950: 5.234836407908006e-06
w:tensor([[2.9985, 7.9894]]); bias:tensor([-0.4992])
```

由上面可以看出，由于实际数据是二次函数，而我们的模型也是二次函数，所以无论从图形上看，还是从学习到的参数上看，都非常好的拟合了原曲线。当最高次为 9 次时，运行结果如下所示：

Figure 21: 多项式回归最高次为 9 次运行结果图



学习到的参数为：

Figure 22: 多项式回归最高次为 9 次学习的参数值

```
7900: 0.0024289193097501993
7950: 0.002417712239548564
w:tensor([[ 3.0672,  7.6114, -0.3955,  1.2342, -0.3613,  0.5319, -0.3699,  0.2009,
         -0.0150]]); bias:tensor([-0.4883])
```

从上面的结果来看，我们 9 次多项式在图像上看，也可以较好的拟合数据点，但是所学到的参数，就和目标值比起来有些差距了，而且运算量也比二次时要大得多。并且我们为了演示效果，我们只选取的 $[-0.5, 0.5]$ 这一个小的区域，如果将区域放大，那么拟合的效果就会更差。
综上所述，我们选择与实际问题相匹配的模型复杂度，对于解决实际问题是非常重要的。

3.4 总结

在本章中，我们向大家详细介绍了线性回归的数学原理，同时还介绍了怎样用解析法求解线性回归问题。接着我们利用标准的 PyTorch 方式，求解了线性回归问题。我们同时讨论了所拟合的函数不是线性回归的情况，我们以二次函数为例，并分别用 1 次、2 次、9 次多项式来拟合，发现 2 次多项式的效果最好，因此得出需要找到与实际问题相匹配的模型，对于解决实际问题非常重要。

第 4 章逻辑回归

Abstract

在本章中我们将讨论分类问题，介绍二分类和多分类问题，也就是逻辑回归和 softmax 回归。

4 逻辑回归概述

从本章开始，将正式进入深度学习领域。逻辑回归算法是最简单的一种模式识别算法，虽然其仅能解决线性可分的问题，但是由于其较简单，故当前还在大量使用。例如在医学诊断中，每个疾病都是由一系列症状和综合特征组成的，对于每一位患者，只要准确地收集其症状和综合特征，利用逻辑回归算法就能做出医学专家水准的诊断。由此可见，逻辑回归算法的威力还是相当强大的。在本章中，将首先研究逻辑回归算法的数学基础，然后以 MNIST 手写数字识别为例，介绍这个数据的格式，以及怎样用逻辑回归算法进行 MNIST 手写数字识别，并且训练模型，使其达到 1% 左右的误差率。

4.1 数学基础

4.1.1 直观解释

逻辑回归算法虽然是非常简单的机器学习算法，但是其数学基础还是比较复杂的。如果刚开始就陷入算法的数学细节中，则很难理解逻辑回归问题的本质，最后只是一堆数学公式，而不知道具体该怎么应用到实际问题中。这种现象是我们应该着力避免的。
所以先不考虑数学理论，而是讲一下逻辑回归算法的物理意义，使读者对逻辑回归算法有一个直观的了解，为对后续数学理论的理解打下基础。
下面通过一个简单的例子来说明什么是逻辑回归算法。假设在三维空间中有一组待分类的点，同时有一系列平面代表这些点应该属于的类别，我们将通过这些点到代表类别的平面的距离，来判断点属于的类别。也就是说，对于一个点来说，我们找到与其距离最近的平面，那么就说这个点属于这个类别。上面的讨论是在三维空间下进行的，如果推广到多维空间，那么这里的平面就变成了超平面，但是概念是类似的。
将上述描述转换成数学语言：假设输入向量为 $\boldsymbol{x}$，其维数为 D，输出类别为 $\boldsymbol{y}$，共有 N 个类别。对于上面的分类问题，如果把问题简化，就变为在二维平面上的点及一系列代表类别的线，求距离该点最近的直线的问题，而直线在二维情况下可以表示为 $y = wx + b$，其中 w 为权重，b 为偏移量。如果将上式推广到高维空间，则权值将变为一个矩阵，偏移量将变为一个向量，可以表示为 $W\boldsymbol{x} + \boldsymbol{b}$，我们将权值矩阵和偏移量向量称为模型的参数集。

4.1.2 数学推导

有了对逻辑回归问题的直观理解之后，开始推导逻辑回归算法。首先讲解单类别逻辑回归问题，如判断患者是否患有某种疾病。然后讨论多类别逻辑回归问题，因为在 MNIST 手写数字识别的例子中，需要判断给定图片是 0 9 中的哪个数字，就是一个多类别模式识别问题。
与线性回归算法类似，下面除了讲述普通迭代法求解，还会讲解利用牛顿法解决逻辑回归问题。通常牛顿法收敛速度更快。
在本章理论部分的最后，会简单讨论一下通用学习模型，因为线性回归和逻辑回归算法都是这种通用学习模型的特例，而且利用通用学习模型，还可以推导出更多机器学习算法。

对于模式分类（Classification）问题，训练样本中 $y \in \{0,1\}$，假设就变为：

$$\hat{y} = h_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \{0,1\} \tag{39}$$

其中 $\boldsymbol{x} \in R^n$ 为特征向量。

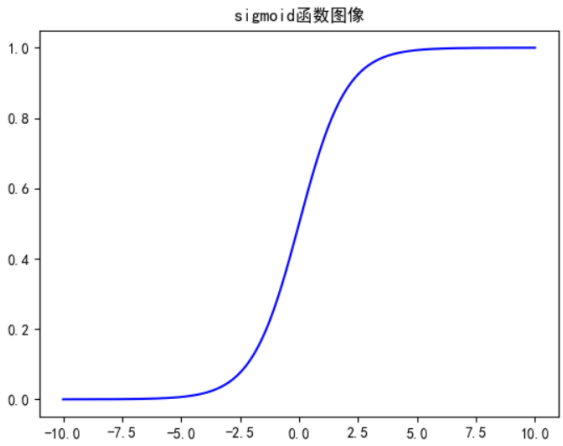
我们使用 sigmoid 函数来表示 $\hat{y} = h_{\boldsymbol{\theta}}(\boldsymbol{x})$，选择 Sigmoid 函数并不是随意的，而是有理论基础的。在本节中，先假定是为了方便而选择 Sigmoid 函数。在后面通用机器学习部分，大家将看到这样做的原因。

sigmoid 函数定义为：

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad z \in R \tag{40}$$

式中的 z 我们称之为线性输入和。Sigmoid 函数曲线为：

Figure 23: sigmoid 函数图像



由于 Sigmoid 函数在深度学习中比较常用，并且经常用到其导数，下面我们来推导其导数形式：

$$\begin{aligned}
\sigma'(z) &= \frac{d(\frac{1}{1+e^{-z}})}{dz} = \frac{d(\frac{1}{1+e^{-z}})}{d(1+e^{-z})}\frac{d(1+e^{-z})}{dz} \\
&= \Big(-\frac{1}{(1+e^{-z})^2}\Big)(-e^{-z}) = \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} = \frac{1}{1+e^{-z}}\Big(1-\frac{1}{1+e^{-z}}\Big) \\
&= \sigma(z)\Big(1-\sigma(z)\Big)
\end{aligned} \tag{41}$$

在上式中，线性输入和定义为：

$$z = \boldsymbol{w}^T \cdot \boldsymbol{x} + b, \quad z \in R \tag{42}$$

其中权值参数 $\boldsymbol{w} \in R^n$，输入信号 $\boldsymbol{x} \in R^n$，$b \in R$ 为偏置值。与线性回归时的表示形式相同。

我们可以将假设函数表示为：

$$\hat{y} = h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{w}^T \cdot \boldsymbol{x} - b}} \tag{43}$$

由于我们研究的问题是一个二分类问题，目标值 y 只能取 0 或者 1，下面我们来表示出 $y=0$ 和 $y=1$ 事件：

$$\begin{aligned}
P(y=1|\boldsymbol{x};\boldsymbol{\theta}) &= \hat{y} \\
P(y=0|\boldsymbol{x};\boldsymbol{\theta}) &= 1-\hat{y}
\end{aligned} \tag{44}$$

在实际应用中，我们经常将上面的两个式子简化为一个式子：

$$P(y|\boldsymbol{x};\boldsymbol{\theta}) = \hat{y}^{y}(1-\hat{y})^{1-y} \tag{45}$$

对上式可以这样理解，对于 $y=1$，等式右边第二项的指数值为 0，其值为 1，所以就剩下第一项；而 $y=0$ 时，等式右边第一项的指数值为 0，其值为 1，则只剩下第二项，所以与用两个等式表达的结果相同。
假设我们连续进行 m 次实验，每次实验均可能取 0 或者 1，我们称这些实验结果出现的概率为似然函数：

$$\mathcal{L}(\boldsymbol{\theta}) = \prod_{i=1}^{m} P(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \tag{46}$$

这时我们的任务就是调整参数 $\boldsymbol{\theta}$ 使得出现上面结果的概率最大，这就是最大似然算法。
在上式中是连乘形式，在实际中很难处理，由于我们知道对一个正数取对数，其相对值的大小不变，所以我们可以对上式取对数，得到对数似然函数：

$$l(\boldsymbol{\theta}) = \log \mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{m} \log P(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \tag{47}$$

将式45代入式47可得：

$$\begin{aligned} l(\boldsymbol{\theta}) &= \sum_{i=1}^{m} \log P(y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \\ &= \sum_{i=1}^{m} \log\left((\hat{y}^{(i)})^{y^{(i)}}(1-\hat{y}^{(i)})^{1-y^{(i)}}\right) \\ &= \sum_{i=1}^{m}\left(y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right) \end{aligned} \tag{48}$$

在实际应用中，我们通常喜欢求最小值，所以我们常用负数似然函数：

$$l(\boldsymbol{\theta}) = \sum_{i=1}^{m}\Big(-y^{(i)}\log\hat{y^{(i)}} - (1-y^{(i)})\log(1-\hat{y}^{(i)})\Big) \tag{49}$$

我们定义：

$$\nabla_{\boldsymbol{\theta}} l(\boldsymbol{\theta}) = \frac{\partial l(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{50}$$

网络操作的更新公式为：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}} l(\boldsymbol{\theta}) \tag{51}$$

式中 $0<\alpha<1$，是学习率。
代价函数值 $\nabla_{\boldsymbol{\theta}} l(\boldsymbol{\theta}) \in R$，网络参数为权值参数 $\boldsymbol{w} \in R^n$ 和偏置值 b。
代价函数值对权值参数导数为：

$$\frac{\partial l(\boldsymbol{\theta})}{\boldsymbol{w}} = \left[\frac{\partial l(\boldsymbol{\theta})}{w_1} \quad \frac{\partial l(\boldsymbol{\theta})}{w_2} \quad \cdots \quad \frac{\partial l(\boldsymbol{\theta})}{w_n}\right] \tag{52}$$

上式中第 j 项为：

$$\frac{\partial l(\boldsymbol{\theta})}{\partial w_j} = \frac{\partial \sum_{i=1}^{m} \Big(- y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \Big)}{\partial w_j} \tag{1}$$

$$= \sum_{i=1}^{m} - \frac{\partial y^{(i)} \log \hat{y}^{(i)}}{\partial w_j} - \frac{\partial (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})}{\partial w_j} \tag{2}$$

$$= \sum_{i=1}^{m} \Big(- \frac{\partial y^{(i)} \log \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial w_j} - \frac{\partial (1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)) \log(1 - \hat{y}^{(i)})}{\partial w_j} \Big) \tag{3}$$

$$= \sum_{i=1}^{m} \Big(- \frac{\partial y^{(i)} \log \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial w_j}$$

$$- \frac{\partial (1 - y) \log(1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b))}{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial w_j} \Big) \tag{4}$$

$$= \sum_{i=1}^{m} \Big(- y^{(i)} \frac{1}{\sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)(1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)) x_j^{(i)}$$

$$- (1 - y^{(i)}) \frac{-1}{1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)(1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)) x_j^{(i)} \Big) \tag{5}$$

$$= \sum_{i=1}^{m} \Big(- y^{(i)} (1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)) x_j^{(i)}$$

$$+ (1 - y^{(i)}) \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b) x_j^{(i)} \Big) \tag{6}$$

$$= \sum_{i=1}^{m} \Big(\sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b) - y^{(i)} \Big) x_j^{(i)} \tag{53}$$

接下来我们来求代价函数对偏置值的微分：

$$\frac{\partial l(\boldsymbol{\theta})}{\partial b} = \frac{\partial \sum_{i=1}^{m} \Big(- y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \Big)}{\partial b} \tag{1}$$

$$= \sum_{i=1}^{m} - \frac{\partial y^{(i)} \log \hat{y}^{(i)}}{\partial b} - \frac{\partial (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})}{\partial b} \tag{2}$$

$$= \sum_{i=1}^{m} \Big(- \frac{\partial y^{(i)} \log \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial b} - \frac{\partial (1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)) \log(1 - \hat{y}^{(i)})}{\partial b} \Big) \tag{3}$$

$$= \sum_{i=1}^{m} \Big(- \frac{\partial y^{(i)} \log \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial b}$$

$$- \frac{\partial (1 - y) \log(1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b))}{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \frac{\partial (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)}{\partial b} \Big) \tag{4}$$

$$= \sum_{i=1}^{m} \Big(- y^{(i)} \frac{1}{\sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)(1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b))$$

$$- (1 - y^{(i)}) \frac{-1}{1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)} \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)(1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b)) \Big) \tag{5}$$

$$= \sum_{i=1}^{m} \Big(- y^{(i)} (1 - \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b))$$

$$+ (1 - y^{(i)}) \sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b) \Big) \tag{6}$$

$$= \sum_{i=1}^{m} \Big(\sigma(\boldsymbol{w}^T \cdot \boldsymbol{x}^{(i)} + b) - y^{(i)} \Big) \tag{54}$$

对权值参数的更新公式为：

$$w_j \leftarrow w_j - \alpha \frac{\partial l(\boldsymbol{\theta})}{\partial w_j}, \quad j \in \{1, 2, ..., n\} \tag{55}$$

对偏置值的更新公司为：

$$b \leftarrow b - \alpha \frac{\partial l(\boldsymbol{\theta})}{\partial b} \tag{56}$$

如果我们用 $\boldsymbol{\theta}$ 来表示网络参数，则可以有如下参数更新公式：

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial l(\boldsymbol{\theta})}{\partial \theta_j} = \theta_j - \alpha(\hat{y}^{(i)} - y^{(i)})x_j, \quad j \in \{1, 2, ..., n\} \tag{57}$$

上式与线性回归算法中的随机梯度下降算法的学习公式非常相似，那么它们是一类函数吗？答案是否定的。因为在上式中，$\hat{y}^{(i)}$ 是非线性的 Sigmoid 函数，所以两个公式虽然外形上有点儿相似，但却是两种不同的算法。

4.1.3 牛顿法

在线性回归算法中，除了使用迭代算法，还可以使用解析法，即最小二乘法求出参数解。但是在逻辑回归中，假设函数使用的是指数函数，所以很难用解析法求出解。但是还有更好的算法，使收敛速度更快，这就是下面要介绍的牛顿法。
先来介绍一下标准的牛顿法，然后再介绍牛顿法在逻辑回归算法中的应用。假设给定一个函数映射 $f : R \to R$，我们的任务是发现一点 x 使 $f(x) = 0$，用牛顿法求解这个问题的公式：

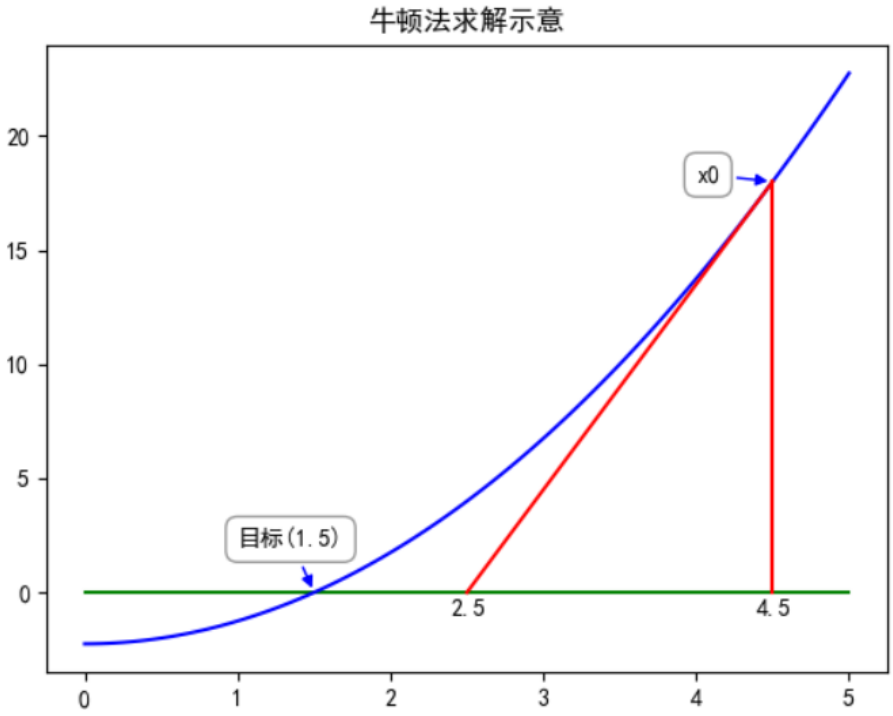
$$x \leftarrow x - \frac{f(x)}{f'(x)} \tag{58}$$

只介绍定义会令人感觉很抽象。下面用一个具体的例子来讲解牛顿法的具体算法，使读者对牛顿法有一个直观的认识。
我们假设有一个函数：

$$f(x) = x^2 - 2.25 \tag{59}$$

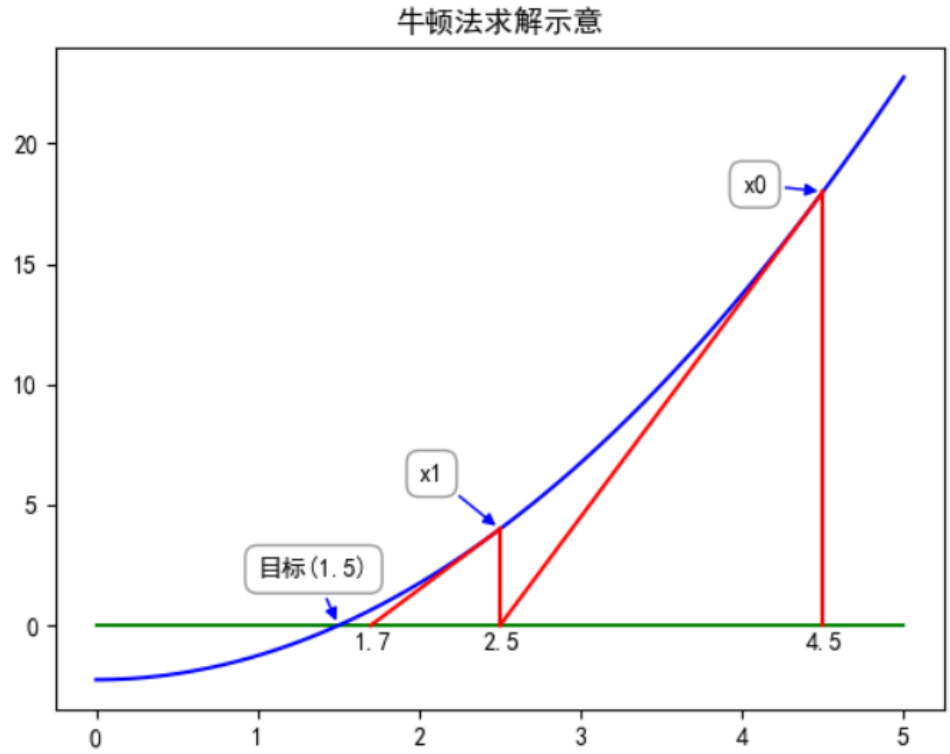
我们知道，其与 x 轴交点为 $x = 1.5$，假设现在的任务就是给定 $f(x)$，以及任意初始值点 $x_0 = 4.5$，求出 $f(x) = 0$ 点的坐标，这就是牛顿法要解决的问题。
根据牛顿法，从 $x_0 = 4.5$ 处做切线，如下图所示：

Figure 24: 过 x_0 点切线示意图



我们在 x_0 点处作曲线的切线，与 x 轴相交于 x_1 点，做曲线过 x_1 点切线，与 x 轴相交于 x_2，如下图所示：

Figure 25: 过 x_1 点切线示意图



此时得到的值 x_2 即为 1.7，而这个函数与 X 轴交点的真实坐标为 1.5，我们仅通过两次迭代，就已经非常接近真实值了。我们相信，如果再连续应用一两次牛顿法，就可以得到 1.5 这个正确值。

从上面的例子可以看出，用牛顿法迭代求值时，可以证明其是以平方的速度向真值点逼近的，如果在离真值点很近的地方，如本次迭代点离真值点的距离为 0.1，那么经过一次迭代之后，其距离将变为 0.01，再经过一次迭代，其距离将进一步缩小为 0.0001。由此可见，其收敛速度是相当快的。

上述图形的生成代码：

```
class Chp003C002(object):
    def __init__(self):
        self.name = ''

    def run(self):
        plt.rcParams['font.sans-serif'] = ['SimHei']
        plt.rcParams['axes.unicode_minus'] = False
        plt.title('牛顿法求解示意')
        self.draw_curve()
        self.draw_x0()
        self.draw_x1()
        plt.show()

    def draw_curve(self):
        # 绘制轴x
        xx = np.array([0.0, 5.0])
```

```
        xy = np.array([0.0, 0.0])
        plt.plot(xx, xy, '-g')
        # 绘制曲线
        x = np.linspace(0, 5, 100)
        y = x*x - 2.25
        plt.plot(x, y, '-b')
        plt.annotate(s=r'目标(1.5)',xy=(1.5, 0.0),\
            xytext=(1.0,2.0),weight='bold',color='black',\
            arrowprops=dict(arrowstyle='->',\
            connectionstyle='arc3',color='blue'),\
            bbox=dict(boxstyle='round,pad=0.5', fc='white', \
            ec='k',lw=1 ,alpha=0.4))

    def draw_x0(self):
        # 点垂线x0
        x00x = np.array([4.5, 4.5])
        x00y = np.array([0.0, 4.5*4.5-2.25])
        plt.plot(x00x, x00y, '-r')
        plt.annotate(s=r'x0',xy=(4.5, 4.5*4.5-2.25),
            xytext=(4.0,18.0),weight='bold',color='black',\
            arrowprops=dict(arrowstyle='->',\
            connectionstyle='arc3',color='blue'),\
            bbox=dict(boxstyle='round,pad=0.5', fc='white', \
            ec='k',lw=1 ,alpha=0.4))
        plt.text(4.4, -1.0, '4.5')
        # 点处切线并与轴相交x0x
        x01x = np.array([4.5, 2.5])
        x01y = np.array([4.5*4.5-2.25, 0])
        plt.plot(x01x, x01y, '-r')
        plt.text(2.4, -1.0, '2.5')

    def draw_x1(self):
        # 点垂线x1
        x10x = np.array([2.5, 2.5])
        x10y = np.array([0.0, 2.5*2.5-2.25])
        plt.plot(x10x, x10y, '-r')
        plt.annotate(s=r'x1',xy=(2.5, 2.5*2.5-2.25),\
            xytext=(2.0,6.0),weight='bold',color='black',\
            arrowprops=dict(arrowstyle='->',\
            connectionstyle='arc3',color='blue'),\
            bbox=dict(boxstyle='round,pad=0.5', fc='white', \
            ec='k',lw=1 ,alpha=0.4))
        # 点切线x1
        x11x = np.array([2.5, 1.7])
        x11y = np.array([2.5*2.5-2.25, 0])
        plt.plot(x11x, x11y, '-r')
        plt.text(1.6, -1.0, '1.7')
```

Listing 21: 牛顿法求解示意图（app/pytorch/book/chp003/chp003_c002.py）

在上述代码中，我们略去了切线方程的求解，以及求出切线方程与 x 轴的交点，这些部分的计算大家可以自己完成并验证，由于这部分代码与深度学习无关，这里就不进行过多解释了。

下面来讨论，怎样将牛顿法应用于逻辑回归问题的求解过程。

在逻辑回归算法中，需要求出负对数似然函数 $l(\boldsymbol{\theta})$ 的最大值，根据高等数学知识可以知道，求极值点就等于求对数似然函数导数为 0 的点，如果把 $l'(\boldsymbol{\theta})$ 视为上面讨论中的 $f(x)$，则就可以针对 $l'(\boldsymbol{\theta})$ 应用牛顿法，公式为：

$$\theta \leftarrow \theta - \frac{l'(\boldsymbol{\theta})}{l''(\boldsymbol{\theta})} \tag{60}$$

应用牛顿法可以更快地求出 l 的极值。

在前述公式推导中，并没有强调实际上我们的特征值是 n 维的，所以输入信号为 n 维的，所以输入信号为 n 维向量 $\boldsymbol{x} \in R^n$，参数同样是 n 维向量 $\boldsymbol{\theta} \in R^n$，因此我们需要牛顿法的向量形式。

我们首先将偏置 b 作为 θ_0，如下所示：

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \cdots \\ \theta_n \end{bmatrix} = \begin{bmatrix} b \\ \theta_1 \\ \theta_2 \\ \cdots \\ \theta_n \end{bmatrix} \in R^{n+1} \tag{61}$$

同理我们在特征向量加入 $x_0 = 1$ 项，如下所示：

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \in R^{n+1} \tag{62}$$

则牛顿法的向量形式可以表示为：

$$\theta^T \leftarrow \theta - H^{-1}\Big(\nabla_{\boldsymbol{\theta}} l(\boldsymbol{\theta})\Big)^T \tag{63}$$

其中代价函数对参数的微分定义为：

$$\nabla_{\boldsymbol{\theta}} l(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial l(\boldsymbol{\theta})}{\partial \theta_0} & \frac{\partial l(\boldsymbol{\theta})}{\partial \theta_1} & \frac{\partial l(\boldsymbol{\theta})}{\partial \theta_2} & \cdots & \frac{\partial l(\boldsymbol{\theta})}{\partial \theta_n} \end{bmatrix} \in R^{1\times(n+1)} \tag{64}$$

式中 H^{-1} 为海森矩阵求逆，海森矩阵定义如下所示：

$$H = \begin{bmatrix} \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_0 \theta_0} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_0 \theta_1} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_0 \theta_2} & \cdots & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_0 \theta_n} \\ \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_1 \theta_0} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_1 \theta_1} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_1 \theta_2} & \cdots & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_1 \theta_n} \\ \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_2 \theta_0} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_2 \theta_1} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_2 \theta_2} & \cdots & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_2 \theta_n} \\ \cdots & & & & \\ \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_n \theta_0} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_n \theta_1} & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_n \theta_2} & \cdots & \frac{\partial l^2(\boldsymbol{\theta})}{\partial \theta_n \theta_n} \end{bmatrix} \in R^{(n+1)\times(n+1)} \tag{65}$$

通常，即在逻辑回归算法中，在特征向量的维数不太高的情况下，例如在几百个特征之内，利用牛顿法求解的速度还是相当快的。但是如果特征数量太多，有成千上万维，因为算法需要对海森矩阵求逆，所以运算量会比较大，使用牛顿法也就不太适合了，迭代算法可能具有更好的性能。

4.1.4 通用学习模型

到目前为止，我们学习了两种学习算法，分别是线性回归算法和逻辑回归算法，其中线性回归算法用于解决数值预测问题，而逻辑回归算法用于模式分类问题。其实这两种算法都是由所谓的通用线性模型（GLM）派生出来的，而且通用线性模型不仅可以派生出线性回归算法和逻辑回归算法，还可以派生出很多其他的主流算法。
对于线性回归问题，可以将其表示为：

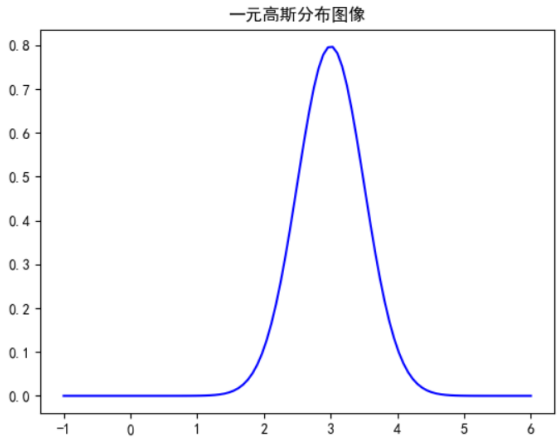
$$P(y|\boldsymbol{x};\boldsymbol{\theta}), \quad y \in R, \quad (y|\boldsymbol{x};\boldsymbol{\theta}) \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma) \tag{66}$$

上式第一部分表示给定样本 $\boldsymbol{x}$ 并以 $\boldsymbol{\theta}$ 为参数情况下某个 y 值出现的概率，我们可以认为，由于研究的问题具有随机性，即使相同的 $\boldsymbol{x}$ 也会出现不同的 y 值，而不同 y 值出现的概率不同。第二部分表示以 $\boldsymbol{\theta}$ 为参数，y 的分布符合均值为 $\boldsymbol{\mu}$ 方差为 Σ 的高斯分布。
由概率论的知识可以知道，一元高斯分布（Gaussian）的概率密度函数为：

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{67}$$

其图像如下所示：

Figure 26: 一元高斯分布函数图像



????????????????? 加入二元高斯分布图像？？？？？？？？？？？？？？？？ 根据以上假设，可以得到线性回归算法，同时可以推导出最小二乘法，这在线性回归算法部分已经详细推导过，这里就不再赘述了。
对于逻辑回归问题，我们定义：

$$P(y|\boldsymbol{x};\boldsymbol{\theta}) = \phi \tag{68}$$

表示给定样本 $\boldsymbol{x}$ 并以 θ 为参数情况下，$y \in \{0, 1\}$ 出现的概率。其中 y 符合伯努利分布 $y \sim Bernulli(\phi)$。在这里我们也认为所研究的问题具有随机性，对于相同的输入样本 $\boldsymbol{x}$，y 可以取 0 或者 1，但是符合伯努利分布。
下面我们来证明，这两种情况实际可以统一于通用机器学习（GML）模型。
下面引入通用线性模型，可以推导出上述两个函数均是这个函数的特殊情况，并且以这个模型为基础，还可以推导出其他有用的模型。
首先定义一个指数族函数：

$$P(y;\boldsymbol{\eta}) = b(y)exp(\boldsymbol{\eta}^T T(y) - \alpha(\boldsymbol{\eta})) \tag{69}$$

在上式中：

- $\boldsymbol{\eta}$：自然参数；
- T(y)：充分统计量，在通常情况下 $T(y)=y$；

通过指定不同的 α、b 和 T，我们可以得到一族函数，这些函数以 $\boldsymbol{\eta}$ 为参数，为指数族函数。

逻辑回归推导 下面我们来推导逻辑回归过程。对于伯努利分布：

$$P(y|\boldsymbol{x};\boldsymbol{\theta}),\quad y\sim Bernulli(\phi) \tag{70}$$

y 的概率质量函数可以表示为：

$$\begin{aligned} P(y|\boldsymbol{x};\boldsymbol{\theta}) &= \phi^y(1-\phi)^{1-y} = \exp\Big(\log\phi^y(1-\phi)^{1-y}\Big) \\ &= \exp\Big(y\log\phi+(1-y)\log(1-\phi)\Big) \\ &= \exp\Big(y(\log\phi-\log(1-\phi))+\log(1-\phi)\Big) \\ &= \exp\Big(\log\frac{\phi}{1-\phi}y+\log(1-\phi)\Big) \end{aligned} \tag{71}$$

我们设定：

- $b(y)=1$；
- $\eta=\log\frac{\phi}{1-\phi}$，此时 η 为标量；
- $T(y)=y$；
- $\alpha(\eta)=-\log(1-\phi)$

将上述假设条件代入式71，我们就可以看到其将变为通用机器学习模型的公式。
我们来研究一下自然参数 η，在伯努利分布下其值定义为：

$$\eta=\log\frac{\phi}{1-\phi} \tag{72}$$

我们可以解出伯努利分布的参数 ϕ 的值：

$$\phi=\frac{1}{1+e^{-\eta}} \tag{73}$$

如果我们定义 $\eta=\boldsymbol{w}^T\boldsymbol{x}+b$，上式就变为我们讲逻辑回归时模型的公式。由此可见，逻辑回归是通用机器学习算法的一个特例。

线性回归 在线性回归问题中，我们认为 $y\sim\mathcal{N}(\mu,\sigma)$ 符合高斯分布，我们设 $\mu=\boldsymbol{w}^T\boldsymbol{x}+b$ 作为其均值，其概率密度函数如下所示：

$$\begin{aligned} P(y|\boldsymbol{x};\boldsymbol{\theta}) &= \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(y-\mu)^2}{2\sigma^2}} \\ &= \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{y^2-2y\mu+\mu^2}{2\sigma^2}} \\ &= \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{y^2}{2\sigma^2}}e^{\frac{2y\mu-\mu^2}{2\sigma^2}} \\ &= \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{y^2}{2\sigma^2}}e^{(\frac{2y\mu}{2\sigma^2}-\frac{\mu^2}{2\sigma^2})} \\ &= \frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{y^2}{2\sigma^2})\exp(\frac{2y\mu}{2\sigma^2}-\frac{\mu^2}{2\sigma^2}) \end{aligned} \tag{74}$$

我们设定：

- $\eta = \mu$;
- $T(y) = y$;
- $a(\eta) = \frac{\mu^2}{2\sigma^2} == \frac{\eta^2}{2\sigma^2}$;
- $b(y) = \frac{1}{\sqrt{2\pi}\sigma} exp(-\frac{y^2}{2\sigma^2})$;

通过指定 η、T、α 和 b 之后，式74就变为了通用机器学习模型（GML）的形式了，由此可见，线性回归也是通用机器学习模型的一个特例。通过这种指数族分布，实际上可以得到很多分布。

- 多项式（Multinomial）分布：例如在手写数字识别中，分为 0 9 共 10 种类别，其中只有一个类别为 1，其余类别均为 0；
- 泊松分布（Poisson）：通常用来估算数量，如估算网站访问量和点击率等，也属于指数族分布；
- 伽码分布（Gamma）：通常用于对时间间隔的建模，也是指数族分布的一种；

在讲述了通用线性模型之后，我们自然要问，怎样使用它呢？假设我们有一个微信公众号，想估算文章主题、更新频率、大 V 数量等参数对粉丝数增长的影响。对于粉丝数增长的问题，可以选用泊松分布（Poisson），因此下面的任务就是设计一种泊松回归算法，来解粉丝数增长估算问题。如果以通用线性模型为指导，通过选择恰当的参数，就可以很方便地得出相应的回归算法。因为在本章的例子中，在使用 MNIST 手写数字识别的实例中，需要用到多项式分布情况，即输出类别为 0 9 共 10 个类别，采用 softmax 函数形式，所以下面就来推导一下多项式分布下的 softmax 函数。

通用机器学习模型总结 在根据通用线性模型做具体的算法设计前，需要有以下 3 个前提条件：

1. $(y|\boldsymbol{x};\boldsymbol{\theta}) \sim \exp^*(\boldsymbol{\eta})$：对于变量 y 在给定输入样本 $\boldsymbol{x}$ 和模型参数 $\boldsymbol{\theta}$ 情况下符合参数为 $\boldsymbol{\eta}$ 的指数族分布；
2. 我们的目标是在给定输入信号 $\boldsymbol{x}$ 的情况下，求出 $T(y)$ 的希望值作为模型函数，即：

$$\hat{y} = E\Big[T(y)|\boldsymbol{x}\Big] = E\Big[y|\boldsymbol{x}\Big] \tag{75}$$

   因为在通常情况下我们都取 $T(y) = y$；
3. 当输出值 y 为标量时,$\eta = \boldsymbol{\theta}\boldsymbol{x}$；如果输出值 $\boldsymbol{y} \in R^m$ 时，$\boldsymbol{\eta}$ 为向量形式，此时参数 $\Theta \in R^{m\times n}$ 为一个矩阵，如下所示：

$$\boldsymbol{\eta} = \Theta\boldsymbol{x} = \begin{bmatrix} \boldsymbol{\Theta}_{1,}\boldsymbol{x} \\ \boldsymbol{\Theta}_{2,}\boldsymbol{x} \\ \cdots \\ \boldsymbol{\Theta}_{m,}\boldsymbol{x} \end{bmatrix}, \Theta_{i,} \in R^{1\times n} \tag{76}$$

线性回归 对于线性回归问题，我们假设 $(y|\boldsymbol{x};\boldsymbol{\theta}) \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ 的高斯分布（Gaussian），根据前面的推导，如果把高斯分布问题视为通用线性模型的指数族函数，则其参数如下：

$$
\begin{aligned}
P(y|\boldsymbol{x};\boldsymbol{\theta}) &= b(y)\exp\Big(\eta^T T(y) - \alpha(\eta)\Big) && (1)\\
\eta &= \mu && (2)\\
T(y) &= y && (3)\\
\alpha(\boldsymbol{\eta}) &= \frac{\mu^2}{2\sigma^2} = \frac{\eta^2}{2\sigma^2} && (4)\\
b(y) &= \frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{y^2}{2\sigma^2}) && (5)
\end{aligned}
\tag{77}
$$

由上式中的第 (2) 行可以得到：

$$
\hat{y} = E[\mu|\boldsymbol{x};\boldsymbol{\theta}] = \eta = \boldsymbol{\theta}^T\boldsymbol{x} \tag{78}
$$

在上式中，我们将偏置值定义为 $\theta_0 = b$，有：由上式中的第 (2) 行可以得到：

$$
\boldsymbol{\theta} = \begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\\ \cdots\\ \theta_n\end{bmatrix} = \begin{bmatrix}b\\ \theta_1\\ \theta_2\\ \cdots\\ \theta_n\end{bmatrix} \in R^{n+1} \tag{79}
$$

将 $x_0 = 1$，有：

$$
\boldsymbol{x} = \begin{bmatrix}x_0\\ x_1\\ x_2\\ \cdots\\ x_n\end{bmatrix} = \begin{bmatrix}1\\ x_1\\ x_2\\ \cdots\\ x_n\end{bmatrix} \in R^{n+1} \tag{80}
$$

逻辑回归 逻辑回归是一个二分类问题，其中 $y \in \{0, 1\}$ 只能取 0 或 1 两个值，其服从的是伯努利分布：

$$
\begin{aligned}
P(y|\boldsymbol{x};\boldsymbol{\theta}) &= b(y)\exp\Big(\eta^T T(y) - \alpha(\eta)\Big) && (1)\\
P(y|\boldsymbol{x};\boldsymbol{\theta}) &= \phi^y(1-\phi)^{1-y} && (2)\\
b(y) &= 1 && (3)\\
\eta &= \log\frac{\phi}{1-\phi} && (4)\\
T(y) &= y && (5)\\
\alpha(\eta) &= -\log(1-\phi) && (6)
\end{aligned}
\tag{81}
$$

根据75可得：

$$
\begin{aligned}
\hat{y} = E\Big[T(y)|\boldsymbol{x};\boldsymbol{\theta}\Big] &= E[y|\boldsymbol{x};\boldsymbol{\theta}] && (1)\\
= 0\cdot P(y=0|\boldsymbol{x};\boldsymbol{\theta}) + 1\cdot P(y=1|\boldsymbol{x};\boldsymbol{\theta}) && (2)\\
&= P(y=1|\boldsymbol{x};\boldsymbol{\theta}) && (3)\\
&= \phi = \frac{1}{1+e^{-\eta}} && (4)\\
&= \frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x}}} && (5)
\end{aligned}
\tag{82}
$$

公式解读如下所示：

- 第 (2) 行：由均值定义得到；
- 第 (4) 行：根据81得到；
- 第 (5) 行：由我们设定 $\mu = \eta = \boldsymbol{\theta}^T \boldsymbol{x}$ 的前提条件得到；

softmax 回归 我们之前研究的逻辑回归问题，分类为 0 或 1，即为二分类问题。在实际应用中，还有很多多分类问题，这就是 softmax 回归要研究的问题，我们在这一节中，将要详细讲解 softmax 回归问题。
在本章的例子中，将要解决 MNIST 手写数字识别问题，其有 0 9 共 10 类，而上面对逻辑回归问题的讨论只涉及了两个类别的分类问题，所以在本部分我们将讨论多个类别的模式分类问题。
在多类别模式分类问题中：$y \in \{1, 2, ..., K\}$，对于要研究的手写数字识别问题，这里 $K = 10$，设定每个类别发生的概率为 ϕ_k，将得到一组概率 $\phi_1, \phi_2, ..., \phi_K$。因为我们处理的是分类问题，所以所有类别出现的概率加在一起应该为 1，即：

$$\sum_{k=1}^{K} \phi_k = 1 \tag{83}$$

所以 $\phi_1, \phi_2, ..., \phi_K$，并不是互相独立的，而是有冗余的。原因很简单，我们以第 k 个类别为例，因为假设：

$$P(y = k|\boldsymbol{x}; \boldsymbol{\theta}) = \phi_k, \quad k \in \{1, 2, ..., K\} \tag{84}$$

则 ϕ_K 可以由前面 $K - 1$ 个概率表示出来：

$$\phi_K = 1 - \sum_{k=1} K - 1\phi_k \tag{85}$$

为了能够从通用线性模型推导出多项式分布，定义 $T(y)$ 为 K 维向量，如下所示：

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ ... \\ 0 \\ ... \\ 0 \end{bmatrix} \quad T(2) = \begin{bmatrix} 0 \\ 1 \\ ... \\ 0 \\ ... \\ 0 \end{bmatrix} \quad ... \quad T(k) = \begin{bmatrix} 0 \\ 0 \\ ... \\ 1 \\ ... \\ 0 \end{bmatrix} \quad ... \quad T(K) = \begin{bmatrix} 0 \\ 0 \\ ... \\ 0 \\ ... \\ 1 \end{bmatrix} \tag{86}$$

下面我们来定义指示函数：

$$1x = \Big\{1, \quad x = True0, \quad x = False \tag{87}$$

利用指示函数，我们可以把多项式分布的 $T(y)$ 表示为：

$$\Big(T(y)\Big)_k = 1\{y = k\} \tag{88}$$

下面求当类别为 k 时，多项式分布的期望：

$$E\left[\Big(T(y)\Big)_k\right] = 1\{y = k\} = \sum_{k=1}^{K} y_k P(y = k) = P(y = k) = \phi_k \tag{89}$$

根据多项式分布的定义，其概率密度函数为：

$$
\begin{aligned}
P(y|\boldsymbol{x};\boldsymbol{\theta}) &= \phi_1^{1\{y=1\}}\phi_2^{1\{y=2\}}...\phi_K^{1\{y=K\}} \quad (1)\\
&= \phi_1^{(T(y))_1}\phi_2^{(T(y))_2}...\phi_{K-1}^{(T(y))_{K-1}}\phi_K^{1-\sum_{k=1}^{K-1}(T(y))_k} \quad (2)\\
&= \exp\Big((T(y))_1\log\phi_1 + (T(y))_2\log\phi_2 + ...\\
&+(T(y))_{K-1}\log\phi_{K-1} + (1-\sum_{k=1}^{K}(T(y))_k\log\phi_k)\Big) \quad (3)\\
&= \exp\Big((T(y))_1\log\frac{\phi_1}{\phi_K} + (T(y))_2\log\frac{\phi_2}{\phi_K} + ...\\
&(T(y))_{K-1}\log\frac{\phi_{K-1}}{\phi_K} + \log\phi_K\Big) \quad (4)\\
&= b(y)\exp(\boldsymbol{\eta}^T T(y) - \alpha(\boldsymbol{\eta})) \quad (5)
\end{aligned}
\tag{90}
$$

公式解析如下所示：

- 第 (1) 行：根据概率定义得出，只得正确类别 k 项有值，其余项均为 1，等于没乘；
- 第 (2) 行：将 $T(y)$ 的定义代入；
- 第 (3) 行：对式子先取对数再取指数值不变；
- 第 (4) 行：将第 (3) 行最后一项展开，剩下第 (4) 行最后一项，将其余减的 $K-1$ 项与对应的 $(T(y))_k$ 项合并，将对数相减改写为对数内相除；
- 第 (5) 行：将其规整为通用机器学习模型；

参数定义为：

$$
\begin{aligned}
b(y) &= 1 \quad (1)\\
\boldsymbol{\eta} &= \begin{bmatrix} \log\frac{\phi_1}{\phi_K} \\ \log\frac{\phi_2}{\phi_K} \\ ... \\ \log\frac{\phi_{K-1}}{\phi_K} \\ 0 \end{bmatrix} \in R^{K-1} \quad (2)\\
\alpha(\boldsymbol{\eta}) &= -\log\phi_K \quad (3)
\end{aligned}
\tag{91}
$$

在上式第 (2) 行中，第 K 项为 $\log\frac{\phi_K}{\phi_K} = \log 1 = 0$。将参数取上面的值之后，就可以从通用线性模型的指数族函数中得到多项式分布的表示形式了。
我们来研究自然参数 $\boldsymbol{\eta}$：

$$
\begin{cases}
\log\frac{\phi_k}{\phi_K}, & k = 1,2,...,K-1\\
0, & k = K
\end{cases}
\tag{92}
$$

对上式两边取指数：

$$
\begin{aligned}
\eta_k &= \log\frac{\phi_k}{\phi_K} \quad (1)\\
e^{\eta_k} &= \frac{\phi_k}{\phi_K} \quad (2)\\
\phi_k &= e^{\eta_k}\phi_K \quad (3)
\end{aligned}
\tag{93}
$$

在第 (2) 行是对第 (1) 行等式两边同时取指数，第 (3) 行是移项后的结果。因为 K 个类别出现的概率加在一起为 1，则有：

$$
\sum_{k=1}^{K}\phi_k = \phi_K\sum_{k=1}^{K}e^{\eta_k} = 1 \tag{94}
$$

由上式可以求 ϕ_K 如下所示：

$$\phi_K = \frac{1}{\sum_{k=1}^{K} e^{\eta_k}} \quad \rightarrow \quad \phi_K = \frac{1}{\sum_{k'=1}^{K} e^{\eta_{k'}}} \tag{95}$$

我们将上式中分母处的累加变量从 k 变为 k' 上式不变。则第 k 类的概率可以表求为：

$$\phi_k = \frac{e^{\eta_k}}{\sum_{k'=1}^{K} e^{\eta_{k'}}} \tag{96}$$

上式就是多类别模式分类问题中输出层经常使用的 softmax 函数。在本章及之后各章的 MNIST 手写数字识别实例中，输出层均采用这种形式。

有了上述这些准备工作之后，就可以推导出 softmax 回归模型了，其实这个模型是逻辑回归模型的泛化形式。

因为此时 $\boldsymbol{\eta}$ 为向量形式，所以网络参数为 $\Theta \in R^{K \times n}$，其中 K 为类别数，n 为输入向量 $\boldsymbol{x}$ 的维度。假设网络参数第 i 行 $\Theta_{i,:}$ 用向量 $\boldsymbol{w}_i \in R^{n \times 1}$ 表示，同时我们令 $\boldsymbol{w}_K = \boldsymbol{0}$，则自然参数 $\boldsymbol{\eta}$ 的分量可以表示为：

$$\eta_k = \boldsymbol{w}^T \boldsymbol{x} + b_k \tag{97}$$

在上式中，当 $k = K$ 时，$\boldsymbol{w}_K = \boldsymbol{0}$ 且 $b_K = 0$，也前面的分析一致。此时 softmax 回归的概率密度函数表示为：

$$P(y = k | \boldsymbol{x}; \Theta) = \phi_k = \frac{e^{\eta_k}}{\sum_{k'=1}^{K} e^{\eta_{k'}}} \tag{98}$$

模型函数可以表示为：

$$\hat{y} = E\Big[T(y)|\boldsymbol{x};\Theta\Big] = \begin{bmatrix} 1\{y=1\} \\ 1\{y=2\} \\ \cdots \\ 1\{y=k\} \\ \cdots \\ 1\{y=K-1\} \\ 1\{y=K\} \end{bmatrix} = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \cdots \\ \phi_k \\ \cdots \\ \phi_{K-1} \\ \phi_K \end{bmatrix} = \begin{bmatrix} \frac{e^{\boldsymbol{w}_1^T \boldsymbol{x}+b_1}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}} \\ \frac{e^{\boldsymbol{w}_2^T \boldsymbol{x}+b_2}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}} \\ \cdots \\ \frac{e^{\boldsymbol{w}_k^T \boldsymbol{x}+b_k}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}} \\ \cdots \\ \frac{e^{\boldsymbol{w}_K^T \boldsymbol{x}+b_K}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}} \end{bmatrix} \tag{99}$$

在式中的第 K 项推导过程如下所示：

$$\begin{aligned} P(y = K|\boldsymbol{x};\Theta) = \phi_K &= 1 - \sum_{k=1}^{K-1} \\ &= \frac{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}} - \frac{\sum_{k^*} e^{\boldsymbol{w}_{k^*}^T \boldsymbol{x}+b_{k^*}}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}} \\ &= \frac{e^{\boldsymbol{w}_K^T \boldsymbol{x}+b_K}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x}+b_{k'}}} \end{aligned} \tag{100}$$

此时 softmax 回归算法可以表述为：给定训练样本集 $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(m)}\}$，求网络参数 $\boldsymbol{\theta}$ 使得代价函数值最小。我们目前已经讲过负对数似然函数，在实际应用中，还有交叉熵函数更加常用，我们将在下一节中进行讲解。我们先来看负对数似然函数。我们定义训练样本集出现的概率为：

$$P(X;\boldsymbol{\theta}) = \prod_{i=1}^{m} P(y^{(i)} = k|\boldsymbol{x}^{(i)};\boldsymbol{\theta}) \tag{101}$$

为了便于计算，我们将止式取对数并加上负号，变为负对数似然函数：

$$\begin{aligned} l(\theta) &= -\log \prod_{i=1}^{m} P(y^{(i)} = k|\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \\ &= -\sum_{i=1}^{m} \log P(y^{(i)} = k|\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \\ &= -\sum_{i=1}^{m} \log \frac{e^{\boldsymbol{w}_k^T \boldsymbol{x} + b_k}}{\sum_{k'=1}^{K} e^{\boldsymbol{w}_{k'}^T \boldsymbol{x} + b_{k'}}} \end{aligned} \tag{102}$$

4.1.5 交叉熵函数

多分类问题 我们在深度学习中遇到的最常见的问题就多分类问题，这时我们神经网络的输出层的激活函数 softmax 函数，代价函数取的是 Cross Entropy 函数。比较遗憾的是，关于这部数学原理的详细描述非常少，大部分都直接给出 Cross Entropy 用于 softmax 情形下的公式，直接拿来用就可以了。所以在这一节中，我们将详细介绍 softmax 函数和 Cross Entropy 函数应用于多分类问题的物理意义和数学推导过程。

多分类问题的表示 我们在这里仍然以 MNIST 手写数字识别任务为例，我们要识别的类别为 10 类，分别为数字 0∼9，我们通常用 one-hot 向量形式来表示：

$$\boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & y_3 & \dots & y_k & \dots & y_{10} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \dots & 1 & \dots & 0 \end{bmatrix} \tag{103}$$

其中等于 1 的维所对应的数字就是我们希望的识别结果。
通常我们神经网络的输出层为 softmax 函数，代表 0∼9 这 10 个数字出现的概率，并且这些概率之和为 1。

$$\hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \hat{y}_3 & \dots & \hat{y}_k & \dots & \hat{y}_{10} \end{bmatrix} \tag{104}$$

在数学上，我们可以把这个问题看成是有 10 个随机变量，分别对应 0∼9 的出现事件，而对于某个数字对应的事件，其值只可能是出现和不出现两种，其符合我们数学上的 Bernoulli 分布，整个问题也就由这 10 个 Bernoulli 分布组成。我们首先来研究某个数字的 Bernoulli 分布。
我们需要明确一点，我们现在研究的是识别任务的目标，也就是对应的 $\boldsymbol{y}$,Bernoulli 分布的定义如下所示：

$$\begin{aligned} P(y = 1) &= \phi \\ P(y = 0) &= 1 - \phi \\ P(Y = y) &= \phi^y \cdot (1 - \phi)^{1-y} \end{aligned} \tag{105}$$

式105中，ϕ 代表该事件出现的概率。

信息论简介 信息论本来是研究在不可靠信道中，例如移动通信，怎样以最小的码长来可靠的传递信息的学科，理论体系非常庞大和复杂，对深度学习来讲，我们只需要知道在信息论中，如果知道最可能发生的事情发生了，那么我们得到的信息量非常小，而如果我们知道不可能发生的事件发生了，则我们得到的信息量将非常大。例如，如果我们知道今天太阳从东方升起，那么我们从中得不到任何有价值的信息，而如果我们知道今天早晨下了雪，路上湿滑，那么我们就知道应该注意出行安全，得到较多的有用信息。
对于一个随机变量 x，取每个可能的值（在这里我们只计论离散值情况）就是一个事件 $x = x$，我们可以定义 self information：

$$I(x) = -\log \big(P(x)\big) \tag{106}$$

在这里我们取的是以 e 为底的对数，单位是 nats。
如果我们把该随机变量所有出现的事件都集中起来，就得到信息论中的香农熵：

$$H(x) = E_{x\sim P}(I(x)) = -E_{x\sim P}\big(\log P(x)\big) \tag{107}$$

从式107可以看出，香农熵可以定义为当 x 符合 P 分布时，x 的 self information 的希望值。
我们可以将在 MNIST 数据集上手写数字识别任务中，正确答案 $\boldsymbol{y}$ 对某个数字的识别视为一个 Bernoulli 分布，将我们神经网络输出层 softmax 函数值输出中某个数字的识别，视为另一个 Bernoulli 分布，我们的目标就是让这两个分布尽可能接近。在信息论，对这个问题可以用 KL 散度来表示，我们假设正确答案所对应的分布为 P，我们神经网络输出的分布为 Q，则 KL 散度定义为：

$$D_{KL}(P\|Q) = E_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = E_{x\sim P}\Big[\log P(x) - \log Q(x)\Big] \tag{108}$$

KL 散度的值永远为正，而且当 P 和 Q 越接近其值越小，当 P 和 Q 是同一分布时，其值为零。但是 KL 散度有一个问题，就是 $D_{KL}(P\|Q)$ 与 $D_{KL}(Q\|P)$ 不相等，不完全等同于 P 和 Q 的距离。这就引出了我们的交叉熵 Cross Entropy 的定义：

$$H(P,Q) = H(P) + D_{KL}(P\|Q) = -E_{x\sim P}\Big[\log Q(x)\Big] \tag{109}$$

多分类问题交叉熵函数定义 回到我们 MNIST 手写数字识别任务，我们假设正确答案对某个数字识别的分布为 P，神经网络输出对某个数字的概率的分布为 Q，我们假设要研究的是第 k 维，即 $y_k = 1$，则有：

$$CrossEntropy = H(P) + D_{KL}(P\|Q) = -E_{x\sim P}\Big[\log Q(x)\Big] = -\frac{1}{N_k}\sum_{i=1}^{N_k}\log \hat{y}_k^{(i)} \tag{110}$$

其中 N_k 表示训练样本集中第 k 维等于 1 的样本数，我们对这些样本求出均值就是交叉熵函数了。为了便于处理，我们根据正确答案的特点，只有第 k 维是 1，其余全为零，我们可以得到如下的计算公式：

$$\begin{aligned} CrossEntropy = H(P) + D_{KL}(P\|Q) &= -E_{x\sim P}\Big[\log Q(x)\Big] = -\frac{1}{N_k}\sum_{i=1}^{N_k}\log \hat{y}_k^{(i)} \\ &= -\frac{1}{N_k}\sum_{i=1}^{N_k}\sum_{k=1}^{K=10} y_k \cdot \log\left(\hat{y}_k^{(i)}\right) \end{aligned} \tag{111}$$

这个公式就是其他文档或教程里给出的 Cross Entropy 的公式，希望大家可以通过我们上面的推导过程，对交叉熵和 softmax 函数有更加深入的理解。
以上我们讨论的是针对识别某个特定数字情况，而实际中有 10 个数字，处理方式完全相同，在训练样本集中包括所有这 10 个数字的识别样本，我们只需要将这些情况加在一起就可以了。

4.2 二分类问题应用

为了直观起见，我们首先生成一个数据集，输入向量 $\boldsymbol{x} \in R^2$，共有两个类别，这样我们可以将所有数据点绘制到二维平台，便于大家观察。代码如下所示：

```
1    def load_dataset(self):
2        '''
3        X, y = skds.make_blobs(n_samples=200, centers=2, n_features=2,
     cluster_std=[0.8, 1.2])
```

```
4         np.savetxt('ds_x.csv', X, delimiter=',')
5         np.savetxt('ds_y.csv', y, delimiter=',')
6         '''
7         X = np.loadtxt(open("./ds_x.csv","rb"), delimiter=",", skiprows
      =0)
8         y = np.loadtxt(open("./ds_y.csv","rb"), delimiter=",", skiprows
      =0)
9         # 绘制第一个类别
10        idx1 = np.where(y == 0)
11        X1 = np.array([X[idx] for idx in idx1[0]])
12        plt.scatter(X1[:, 0:1], X1[:, 1:2], c='r')
13        # 绘制第二个类别
14        idx2 = np.where(y == 1)
15        X2 = np.array([X[idx] for idx in idx2[0]])
16        plt.scatter(X2[:, 0:1], X2[:, 1:2], c='b', marker='x')
17        plt.show()
18        return X, y
```

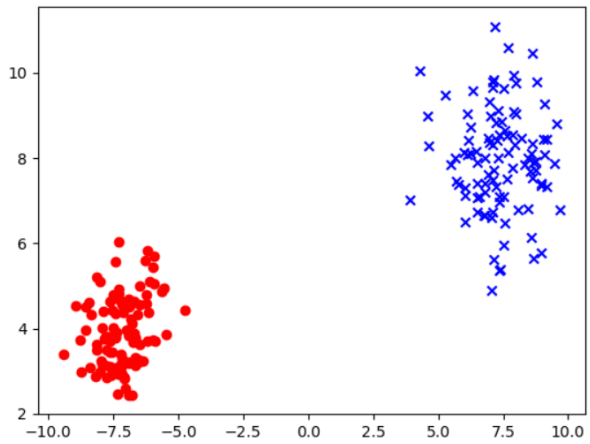
Listing 22: 生成学习数据（app/pytorch/book/chp003/e1/logistic_regression_app.py）

代码解读如下所示：

- 第 3 行：调用 sklearn.datasets.samples_generator.make_blobs 方法，生成训练数据集，centers=2 代表两个类别，n_features=2 代表 $\boldsymbol{x} \in R^2$，并且第一个类别的方差为 0.8，第二个类别的方差为 1.2；
- 第 4 行：将样本数据保存到当前目录下的 ds_x.csv 文件中；
- 第 5 行：将标签数据 y 保存到当前目录下的 ds_y.csv 文件中；
- 第 6 行：以上程序为训练数据生成代码，只需运行一次，生成训练数据保存到文件即可，之后的运行中，我们都是直接读数据文件；
- 第 7 行：从当前目录下的 ds_x.csv 文件中读出样本数据 $X \in R^{m \times 2}$；
- 第 8 行：从当前目录下的 ds_y.csv 文件中读出标签数据 $\boldsymbol{y} in R^m$；
- 第 10 行：从标签集中找出所有值为零的元素的下标值的列表 idx1；
- 第 11 行：从样本集中取出这些下标所对应的元素组成一个新的数组 X1；
- 第 12 行：以第 1 列数据作为 x，以第 2 列数据作为 y，以红色缺省图标绘制散点图；
- 第 10 行：从标签集中找出所有值为 1 的元素的下标值的列表 idx2；
- 第 11 行：从样本集中取出这些下标所对应的元素组成一个新的数组 X2；
- 第 12 行：以第 1 列数据作为 x，以第 2 列数据作为 y，以蓝色叉图标绘制散点图；

数据集图形如下所示：

Figure 27: 逻辑回归二分类数据集图



我们首先来定义逻辑回归模型类：

```
import torch
from torch.autograd import Variable
import torch.nn.functional as F

class LogisticRegressionModel(torch.nn.Module):
    def __init__(self):
        super(LogisticRegressionModel, self).__init__()
        self.l1 = torch.nn.Linear(2, 1)

    def get_weight(self):
        return self.l1.weight

    def get_bias(self):
        return self.l1.bias

    def forward(self, x):
        y_hat = F.sigmoid(self.l1(x))
        return y_hat
```

Listing 23: 逻辑回归模型类（app/pytorch/book/chp003/e1/logistic_regression_model.py）

在上面的代码中，我们的网络由一个线性层组成，激活函数为 Sigmoid 函数。
接下来我们来看模型的训练过程，如下所示：

```
class LogisticRegressionApp(object):
    def __init__(self):
        self.name = ''

    def run(self):
        print('二分类逻辑回归问题')
        X_train, y_train = self.load_dataset()
        model = LogisticRegressionModel()
        criterion = torch.nn.BCELoss(size_average=True)
```

```
10          optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
11          #optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
12          for epoch in range(1000):
13              y_hat = model(X_train)
14              loss = criterion(y_hat, y_train)
15              print('{0}: {1}'.format(epoch, loss.data.item()))
16              optimizer.zero_grad()
17              loss.backward()
18              optimizer.step()
19          print(model.parameters)
20          w = model.get_weight().data.numpy()
21          print('weight:{0}; {1}'.format(w.shape, w))
22          b = model.get_bias().data.numpy()
23          print('bias:{0}; {1}'.format(b.shape, b))
24          # 学习后曲线
25          x0 = np.linspace(-1.0, 0.0, 100)
26          y0 = -w[0][0]/w[0][1]*x0 - b[0]/w[0][1]*1.14
27          plt.plot(x0, y0, '-b')
28
29          x1 = Variable(torch.tensor([[3.31, 0.21]]))
30          y1_hat = model.forward(x1).data[0][0]
31          print('y={0}'.format(y1_hat))
32          plt.show()
```

Listing 24: 二分类逻辑回归训练（app/pytorch/book/chp003/e1/logistic_regression_app.py）

代码解读如下所示：

- 第 7 行：载入上一节所介绍的数据集；
- 第 8 行：创建逻辑回归模型；
- 第 9 行：采用交叉熵代价函数；
- 第 10 行：采用随机梯度下降算法来做优化器；
- 第 12 行：循环执行 13~18 行 1000 次；
- 第 13 行：调用模型前向传播过程求出网络输出；
- 第 14 行：计算出代价函数的值；
- 第 15 行：打印训练进展情况；
- 第 16 行：清空网络参数的微分值缓存；
- 第 17 行：利用 PyTorch 的自动微分求出对网络参数的微分值；
- 第 18 行：调用随机梯度下降优化器调整网络参数；
- 第 20 行：将连接权值 $\boldsymbol{w}$ 变为 numpy 数组；
- 第 22 行：将偏置值 b 变为 numpy 数组；
- 第 25~27 行：绘制学习到的分割超平面（直线）；
- 第 29 行：生成一个测试样本；
- 第 30 行：调用模型前向传播过程求出网络输出；

- 第 31 行：我们可以看出网络输出为 0.9548038840293884，我们可以非常肯定的是该点属于第 2 类（右上方类别）；

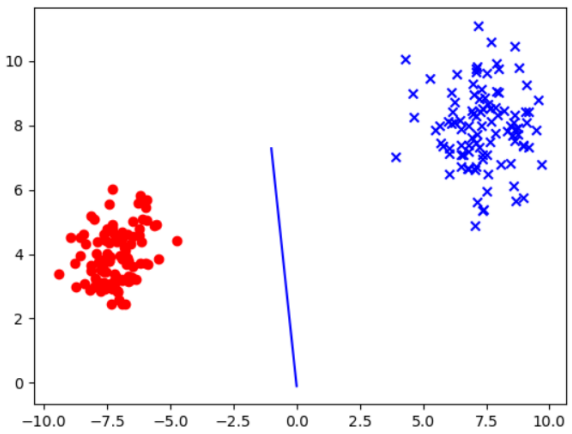
运行结果如下所示：

Figure 28: 逻辑回归二分类学习结果（参数值）图

```
<bound method Module.parameters of LogisticRegressionModel(
  (11): Linear(in_features=2, out_features=1, bias=True)
)>
weight:(1, 2): [[0.9104482  0.12319656]]
bias:(1,): [0.01103974]
y=0.9548038840293884
```

由上图可以看出，参数值为：$w_1 = 0.9104482$，$w_2 = 0.12319656$，$b = 0.01103974$，其分割超平面方程为：$w_1x_1 + w_2x_2 + b = 0$，可以将其解析为直线方程：$x_2 = -\frac{w_1}{w_2}x_1 + \frac{b}{w_2}$，如下图所示：

Figure 29: 逻辑回归二分类学习结果图



如上图所示，我们的模型可以正确区分这两个类别，实际上就是在图中找出中间的直线，将样本点划分为两类。

4.3 softmax 回归应用

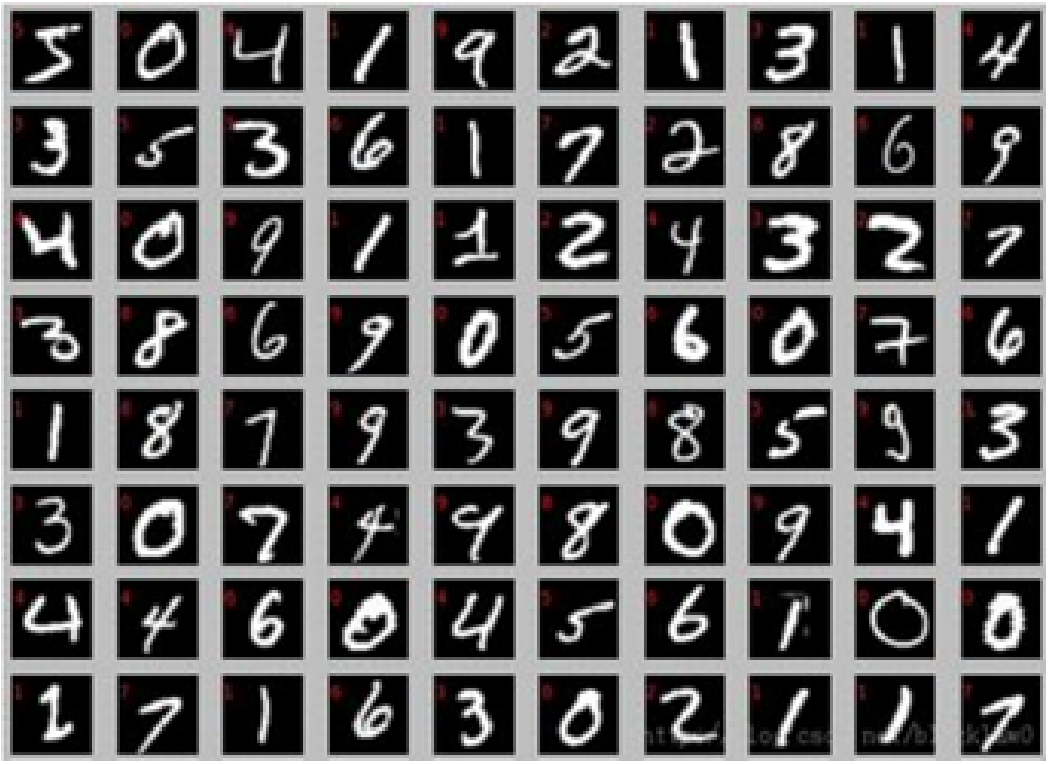
4.3.1 载入 MNIST 数据集

MNIST 是 Mixed National Institute of Standards and Technology 的简称，是国际公认的大型手写数字识别数据集，是机器学习算法验证的标准数据集，其地位有点像生物学研究中的大肠杆菌，可以方便地用于各种机器学习算法间性能的比较。实际上，MNIST 每年都会组织算法竞赛，以检验各种算法的有效性。

MNIST 数据集由纽约大学 Yann LeCun 教授整理推出，每个手写数字图片的大小均为 28×28，黑底白字，并且位于图片中央，共有 60000 个训练样本集，10000 个测试样本集，其中测试样本集是不公开的。

MNIST 手写数字图片如下图所示：

Figure 30: MNIST 手写数字识别图片示例



因为 MNIST 数据集在国外网站上，通过 sklearn.datasets.ftech_openml 下载数据集会有很慢，有时会下载不下来。因此我们采取直接从网站下载别人整理好的 CSV 格式的文件，然后载入该 CSV 文件，程序如下所示：

```
from __future__ import print_function
import csv
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import sklearn.datasets as skds
#
from util.npai_ds import NpaiDs
from util.npai_stats import NpaiStats
from ann.ml.logistic_regression import LogisticRegression
from util.npai_plot import NpaiPlot

classMlpApp(object):
    def __init__(self):
        self.name = 'app.ml.Mlp'

    def test_load_mnist_ds(self):
        X, y = self.load_mnist_ds()


    def load_dataset(self):
```

```
        # 文件下载链接：CSVhttps://www.openml.org/d/554
        # 从网络上获取数据集：
    X, y = skds.fetch_openml('mnist_784', version=1, return_X_y=True)
        with open('E:/alearn/dl/npai/data/mnist_784.csv', newline='',
    encoding='UTF-8') as fd:
            rows = csv.reader(fd, delimiter=',', quotechar='|')
            X0 = None
            y0 = None
            next(rows)
            cnt = 0
            rst = 0
            amount = 1000 # 每条记录保存一次1000
            X = None
            y = None
            for row in rows:
                x = np.array(row[:784], dtype=np.float)
                x /= 255.0
                y_ = np.array(row[784:])
                if None is X:
                    X = np.array([x])
                    y = np.zeros((1, 10))
                    y[cnt, int(y_[0])] = 1
                else:
                    X = np.append(X, x.reshape(1, 784), axis=0)
                    yi = np.zeros((1, 10))
                    yi[0, int(y_[0])] = 1
                    y = np.append(y, yi.reshape(1, 10), axis=0)
                if cnt % amount == 0 and cnt > 0:
                    if None is X0:
                        X0 = X
                        y0 = y
                    else:
                        X0 = np.append(X0, X, axis=0)
                        y0 = np.append(y0, y, axis=0)
                    X = None
                    y = None
                    cnt = 0
                    rst += amount
                    print('处理完记录{0}'.format(rst))
                else:
                    cnt += 1
            idx = 101
            plt.title('{0}th sample: {1}'.format(idx, np.argmax(y[idx])))
            plt.imshow(X[idx].reshape(28, 28), cmap='gray')
            plt.show()
        return X0, y0
```

Listing 25: 通过预先下载的 CSV 文件载入 MNIST 数据集 (app.pytorch.book.chp003.e2.logistic_regression_app.py)

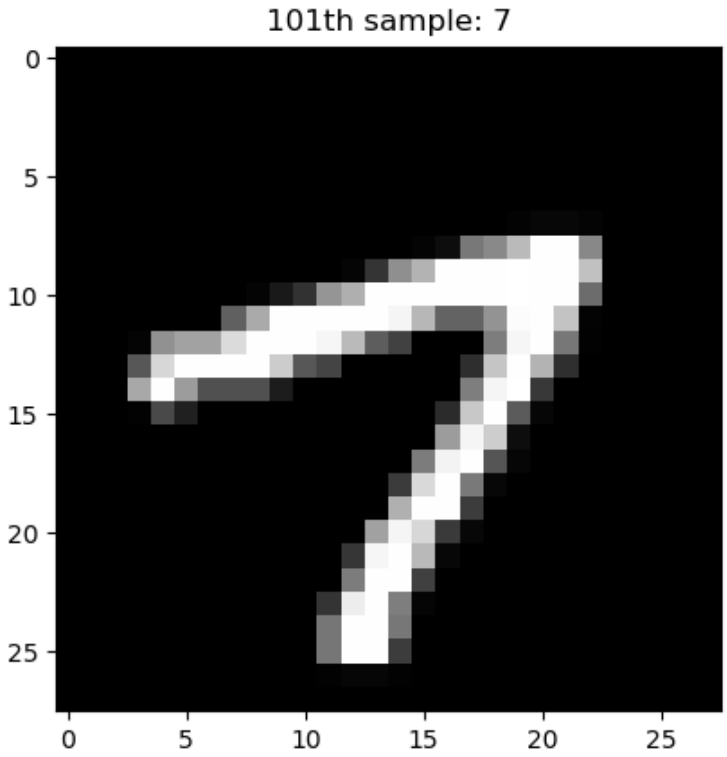
代码解读如下所示：

- 第 19 行：在启动程序中选择运行多分类逻辑回归问题；
- 第 29∼31 行：定义由 CSV 文件载入 MNIST 数据集方法，大家可以通过链接直接从网站上下载 CSV 格式文件；
- 第 32 行：读入下载的 CSV 文件；
- 第 33 行：读取文件中所有行的集合；
- 第 34、35 行：定义最终需要返回的样本集和标签集，初始值为空；
- 第 36 行：忽略 CSV 文件的第一行表头信息；
- 第 37 行：由于我们为了提高 CSV 文件读取和转换为数据集的效率，我们以 amount 条记录为单位，先形成这 1000 条记录的样本集 X 和标签集 y，然后将其添加到最终样本集 X0 和最终标签集 y0 的后面，cnt 代表这 1000 条记录的索引号；
- 第 38 行：共处理了多少条数据；
- 第 39 行：第 1000 条数据形成样本集 X 和标签集 y，然后添加到最终样本集 X0 和最终标签集 y0 的后面；
- 第 40、41 行：amount 条数据形成的样本集 X 和标签集 y；
- 第 42∼68 行：循环处理 CSV 文件的每一行；
- 第 43 行：由该行前 784 个元素形成样本，即一个 28×28 的黑白图片；
- 第 44 行：由于 CSV 文件中像素值的取值范围是 0∼255，除以 255 可以变为 0∼1 区间，方便将来利用 matplotlib 绘制图形；
- 第 45 行：由该行第 785 个元素形成标签，其值为 0∼9 的数字，代表该行样本对应的数字，我们在后面将会将该值变为 10 维的 one-hot 向量形式来表式；
- 第 46∼49 行：如果 amount 条记录形成的样本集 X 和标签集 y 为空，则生成只含有当前样本的样本集 X 和只含有当前标签的标签集 y，对于标签处理，我们先成一个全为零的 10 维向量，然后根据标签数值，将 one-hot 向量对应位置的值变为 1；
- 第 50∼54 行：如果 amount 条记录形成的样本集 X 和标签集 y 不为空，则将当前样本添加到样本集 X 后面，先成一个全为零的 10 维向量，然后根据标签数值，将 one-hot 向量对应位置的值变为 1，然后将此标签添加到标签集 y 的后面；
- 第 55∼66 行：如果正好处理了 amount 条 CSV 文件中的数据，进行如下处理：
  - 第 56∼58 行：如果最终样本集为空，则最终样本集等于当前样本集，最终标签集等于当前标签集；
  - 第 59∼61 行：如果最终样本集不为空，则将当前样本集添加到最终样本集后面，当前标签集添加到最终标签集后面；
- 第 62、63 行：将 amount 条数据对应的样本集 X 和标签集 y 置为空；
- 第 64 行：amount 对应的样本集索引号置为 0；
- 第 65 行：更新处理完成的总记录数；
- 第 67、68 行：如果没处理完 amount 条 CSV 文件中数据，增加 amount 样本集的索引号；
- 第 69 行：返回最终样本集 X0 和最终标签集 y0；

下面我们来看可视化 MNIST 数据集：

- 第 21 行：定义多分类逻辑回归算法运行程序；
- 第 22 行：调用 load_mnist_ds 方法获取样本集 X 和标签集 y；
- 第 23 行：指定要显示样本的索引号；
- 第 24 行：将样本索引号和样本最终代表的数字作为图片标题，np.argmax 可以求出数组中最大元素的索引号；
- 第 25 行：将样本数据重新组织为 28 × 28 的黑白图像格式，并利用 matplotlib 进行绘制，如下所示：

Figure 31: MNIST 手写数字识别单位样本示例



4.3.2 MNIST 数据集可视化

由于手写数字识别数据集 MNIST 为 28 × 28 的黑白图像，所以每个样本是 784 维，而我们只能直观的看到二维样本的情形。为了能够直观的显示 MNIST 数据集，我们使用 t-SNE 算法，将其变为二维形式进行展示。代码如下所示（app/pytorch/book/chp003/e2/logistic_regression_app.py）：

```
1 class MlpApp(object):
2     def __init__(self):
3         self.name = 'ann.ml.MlpApp'
4 
5     def show_mnist_in_tsne(self):
6         X, y_ = self.load_mnist_ds()
7         y = np.argmax(y_, axis=1)
```

```
8         row_embedded = skmd.TSNE(n_components=2).fit_transform(X)
9         pos = pd.DataFrame(row_embedded, columns=['X', 'Y'])
10        pos['species'] = y
11        ax = pos[pos['species']==0].plot(kind='scatter', x='X', y='Y',
    color='blue', label='0')
12        pos[pos['species']==1].plot(kind='scatter', x='X', y='Y', color='
    red', label='1', ax=ax)
13        pos[pos['species']==2].plot(kind='scatter', x='X', y='Y', color='
    green', label='2', ax=ax)
14        pos[pos['species']==3].plot(kind='scatter', x='X', y='Y', color='
    yellow', label='3', ax=ax)
15        pos[pos['species']==4].plot(kind='scatter', x='X', y='Y', color='
    brown', label='4', ax=ax)
16        pos[pos['species']==5].plot(kind='scatter', x='X', y='Y', color='
    orange', label='5', ax=ax)
17        pos[pos['species']==6].plot(kind='scatter', x='X', y='Y', color='
    black', label='6', ax=ax)
18        pos[pos['species']==7].plot(kind='scatter', x='X', y='Y', color='
    pink', label='7', ax=ax)
19        pos[pos['species']==8].plot(kind='scatter', x='X', y='Y', color='
    purple', label='8', ax=ax)
20        pos[pos['species']==9].plot(kind='scatter', x='X', y='Y', color='
    cyan', label='9', ax=ax)
21        plt.show()
```

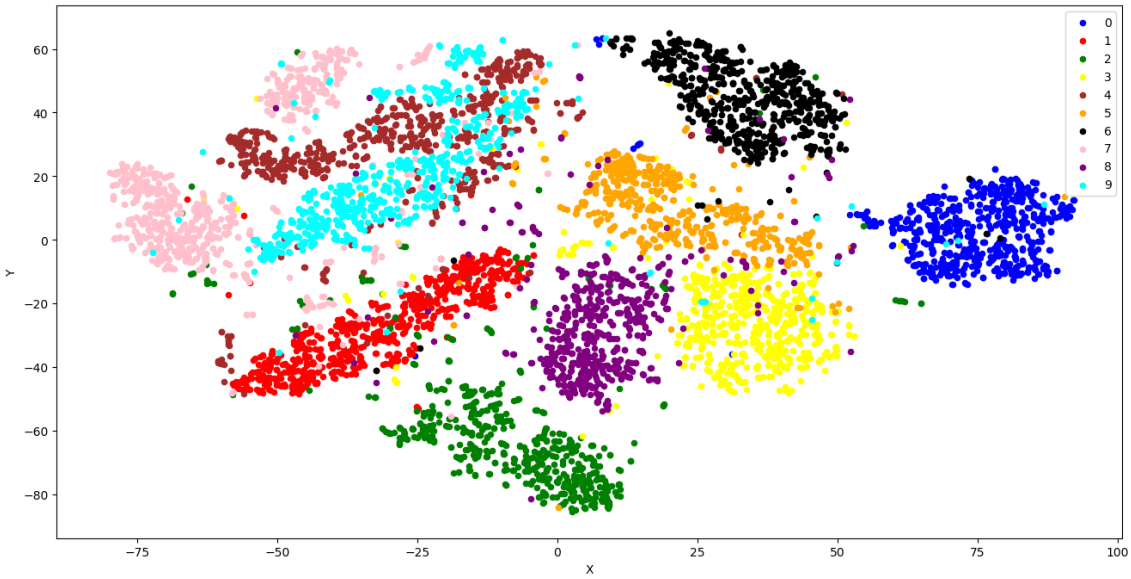
Listing 26: t-sne 可视化 MNIST 数据集

代码解读如下所示：

- 第 5 行：定义 t-sne 算法显示 MNIST 数据集方法；
- 第 6 行：读入 MNIST 数据集，$X \in R^{m \times 784}$ 为样本集，每一行为一个图片，$\boldsymbol{y} \in R^{10}$ 为 one-hot 向量，不为 0 的索引值代表对应的数据；
- 第 7 行：将原来的以 one-hot 向量形式的标签集，变为索引号形式；
- 第 8 行：调用 sklearn.manifold.TSNE 方法进行降维处理；
- 第 9 行：将其变为两列的 pandas.DataFrame；
- 第 10 行：将标签作为 DataFrame 的第三列；
- 第 11 行：将标签为 0 的记录，以蓝色散点图形式绘制到图片中；
- 第 12~19 行：将类别为 1 到 9 的样本，以不同颜色散点图形大绘制到图片中；
- 第 20 行：显示图片；

t-SNE 图片如下所示：

Figure 32: MNIST 数据集 t-SNE 图



由上图可以看出，通过选择合理的特征，我们还是可以将手写数字区分开的。

4.3.3 代码实现

在这一部分，我们将来看怎样用 PyTorch 实现 softmax 回归（即多分类逻辑回归），为了代码的完整性，我们在这里列出了完整的类代码，我们先来看多分类逻辑回归模型类，如下所示（app/pytorch/book/chp003/e2/logistic_regression_model.py）：

```
1  class LogisticRegressionModel(torch.nn.Module):
2      def __init__(self):
3          super(LogisticRegressionModel, self).__init__()
4          self.l1 = torch.nn.Linear(784, 10)
5
6      def get_weight(self):
7          return self.l1.weight
8
9      def get_bias(self):
10         return self.l1.bias
11
12     def forward(self, x):
13         return F.softmax(self.l1(x))
```

Listing 27: softmax 回归模型类

上面的代码与二分类逻辑回归模型类基本相同，所不同的是：

- 第 4 行：定义线性模型时，输出信号维度为 10，代表 10 个数字现现的概率；
- 第 13 行：将输出层激活函数由 Sigmoid 函数变为 softmax 函数；

我们接下来看多分类逻辑回归应用类，如下所示（app/pytorch/book/chp003/e2/logistic_regression_app.py）：

```
class LogisticRegressionApp(object):
    def __init__(self):
        self.name = ''
        plt.rcParams['font.sans-serif'] = ['SimHei']
        plt.rcParams['axes.unicode_minus'] = False

    def run(self):
        print('回归应用softmaxMNIST')
        X_train, y_train = self.load_dataset()
        epochs = 1000
        model = LogisticRegressionModel()
        criterion = torch.nn.BCELoss(size_average=True)
        optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
        #optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
        for epoch in range(epochs):
            y_hat = model(X_train)
            loss = criterion(y_hat, y_train)
            print('{0}: {1}'.format(epoch, loss.data.item()))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        w = model.get_weight().data.numpy()
        print('weight:{0}; {1}'.format(w.shape, w))
        b = model.get_bias().data.numpy()
        print('bias:{0}; {1}'.format(b.shape, b))
        # 测试模型
        idx = 118
        Xt = X_train[idx:idx+1, :]
        yt = y_train[idx:idx+1, :]
        yt_hat = model(Xt)
        print('Xt:{0}; yt:{1}; yt_hat:{2}'.format(Xt.shape, yt.shape,
    yt_hat.shape))
        r = torch.argmax(yt)
        r_hat = torch.argmax(yt_hat)
        print('r:{0}; r_hat:{1}'.format(r, r_hat))
        plt.title('第张图预测结果{0}: {1}'.format(idx+1, r_hat))
        X_raw = Xt[0].numpy().reshape(28, 28)
        plt.imshow(X_raw, cmap='gray')
        plt.show()

    def load_dataset(self):
        # 文件下载链接：CSVhttps://www.openml.org/d/554
        # 从网络上获取数据集：
    X, y = skds.fetch_openml('mnist_784', version=1, return_X_y=True)
        with open('E:/alearn/dl/npai/data/mnist_784.csv', newline='',
    encoding='UTF-8') as fd:
            rows = csv.reader(fd, delimiter=',', quotechar='|')
            X0 = None
            y0 = None
            next(rows)
```

```
48              cnt = 0
49              rst = 0
50              amount = 1000 # 每条记录保存一次1000
51              X = None
52              y = None
53              for row in rows:
54                  x = np.array(row[:784], dtype=np.float)
55                  x /= 255.0
56                  y_ = np.array(row[784:])
57                  if None is X:
58                      X = np.array([x])
59                      y = np.zeros((1, 10))
60                      y[cnt, int(y_[0])] = 1
61                  else:
62                      X = np.append(X, x.reshape(1, 784), axis=0)
63                      yi = np.zeros((1, 10))
64                      yi[0, int(y_[0])] = 1
65                      y = np.append(y, yi.reshape(1, 10), axis=0)
66                  if cnt % amount == 0 and cnt > 0:
67                      if None is X0:
68                          X0 = X
69                          y0 = y
70                      else:
71                          X0 = np.append(X0, X, axis=0)
72                          y0 = np.append(y0, y, axis=0)
73                      X = None
74                      y = None
75                      cnt = 0
76                      rst += amount
77                      print('处理完记录{0}'.format(rst))
78                  else:
79                      cnt += 1
80              #self.draw_dataset(X0, y0)
81              print('X0:{0} vs {2}; y0:{1} vs {3}'.format(X0.shape, y0.
    shape, X0.dtype, y0.dtype))
82              X0 = np.array(X0, dtype=np.float32)
83              y0 = np.array(y0, dtype=np.float32)
84          return Variable(torch.from_numpy(X0)), Variable(torch.from_numpy(
    y0))
```

Listing 28: softmax 回归代码实现

代码解读如下所示：

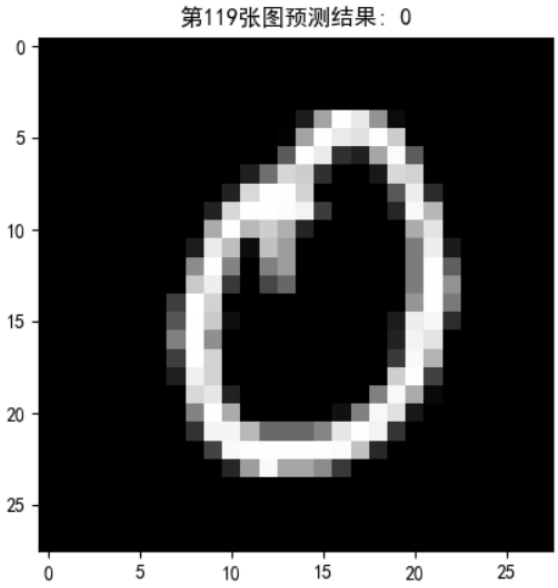
- 第行:;
- 第行:;
- 第行:;
- 第行:;
- 第行:;

运行结果如下所示：

Figure 33: 运行结果后台输出

```
996: 0.21893814206123352
997: 0.21886096894741058
998: 0.2187831997871399
999: 0.21870622038841248
weight:(10, 784); [[-0.0354643   0.00931083  0.02506286 ... -0.00495472  0.02016767
   0.01218596]
 [-0.01392718  0.0211636   0.00301772 ... -0.00716948 -0.01075491
  -0.01952665]
 [ 0.02929425 -0.01321891  0.01656839 ...  0.01932243  0.01866839
  -0.03261283]
 ...
 [ 0.0226347   0.00404936 -0.00149247 ...  0.00398033  0.01340739
  -0.02794876]
 [-0.00713882 -0.01138555 -0.01715047 ... -0.03297758  0.02124333
  -0.00026979]
 [-0.02331644 -0.00845982 -0.01630524 ...  0.03316144 -0.00503226
   0.01132584]]
bias:(10,); [-0.01401416  0.04500581 -0.02132655  0.00696151 -0.00126627  0.02756357
  0.00536281  0.04988732 -0.04470437 -0.01765244]
Xt:torch.Size([1, 784]); yt:torch.Size([1, 10]); yt_hat:torch.Size([1, 10])
r:0; r_hat:0
```

Figure 34: 运行结果示意图



由上图可见，我们模型还是可以预测出正确结果的。需要注意的是，我们这里作为示例，使用的是刚训练过的训练数据集中的样本，实际上这样验证模型的精度是有严重问题的。实际上，我们应该用训练数据集来训练模型，从独立同分布（IID）中采样的验证样本集上精度变化情况，决定何时停止训练以及选择哪些超参数，当模型训练结束后，再运行测试数据集，得到在测试数据集上的精度，作为模型在实际应用中近似的精度。

4.4 总结

在本章中，我们详细讲解了逻辑回归算法的数学原理，包括通用学习模型和牛顿法，重点讲解了二分类问题和多分类问题，对于二分类问题，我们讲解了自己生成的人工数

据的解决方案，对于多分类问题，我们使用手写数字识别 MNIST 数据集，向大家演示了应用方法。但是大家可能注意到，在本章的例子中，我们并没有采用训练数据集、验证数据集和测试数据集，也没有考虑过拟合（Over-Fitting）问题，这些问题我们将在下一章多层感知器模型中进行讲解，作为练习，大家可以在阅读完下一章后，将本章的程序改造为使用验证数据集、测试数据集，使用 Early Stopping、Dropout 等调整手段保证不过拟合（Over-Fitting）的实用算法。

第 5 章多层感知器

Abstract

在本章中我们将详细讲解多层感知器（MLP）模型，本章分为三大部分：第一部分：我们将以一个简单的例子为例，讲解怎样使用 Numpy 来做多层感知器（MLP）模型，重点在于讲解多层感知器模型的数学原理；第二部分我们将以 PyTorch 的典型方式实现多层感知器模型；第三部分是使用 PyTorch 底层技术来实现多层感知器（MLP）模型。

5 多层感知器（MLP）概述

多层感知器（MLP）模型是最老的一类神经网络模型，以前人们通常认为多层感知器（MLP）模型的能力有限，而且根据万能逼近理论，只有一层的神经网络，只要中间层有足够多的节点，就可以拟合任意复杂的函数。所以很长时间以来，研究人员通常认为浅而宽的神经网络是最好的神经网络结构，没有必要采用深层神经网络。但是在 2006 年，深度学习之父 Hinton 通过预训练方式，使得基于多层感知器（MLP）模型的深度信念网络（DBN）取得重大成功，成为当代深度学习技术崛起的开端。此后随着 Web2.0 的发展，有了越来越大的训练数据集，通过利用 GPU，使算力提到上百倍的提升。人们发现，多层感知器（MLP）模型具有巨大的应用潜力，是功能最为强大的神经网络模型。其他在深度学习领域大获成功的模型，利如卷积神经网络（CNN）、递归神经网络（RNN）都可以视为多层感知器（MLP）模型的特例。

通过研究多层感知器（MLP）的发展历史，其由上世纪 90 年代被打入冷宫，到今天在深度学习时代大行其道，在这中间有三项关键的推动因素：第一项是更加适合的代价函数，在此之前，人们普遍采用平均平方误差（MSE）来衡量，神经元激活函数通常采用 Sigmoid 函数，由于 Sigmoid 函数的特点，在偏差值较大时，反而是其曲线非常平坦的区域，所以学习速度会非常慢，所以以前的深层网络非常难以训练，到近代之后，人们逐渐采用交叉熵（CrossEntropy）作为代价函数，极大缓解了深层网络学习缓慢的问题；第二项是在原来的深层网络中，人们在隐藏层也采用 Sigmoid 函数，由于 Sigmoid 函数的特点，在输出值较大或较小时，其微分都非常接近于零，因此学习速率非常低，这种现象在深层网络中尤其明显，在当代，人们采用 ReLU 函数及其变体 LickyReLU、SeLU、ELU 等激活函数作为隐藏层神经元的激活函数，这些函数的特点是当神经元输入小于某个阈值时，神经元处于非激活状态，其微分值为零，当超过该阈值时，其微分值为某个正数，与输入的信号大小无关，正是由于这一特性，成功的解决了多层神经网络训练慢的问题；第三项与算法改进的关系不大，但是可能是最重要的原因，就是随着 Web2.0 的兴起，人们掌握了越来越大的数据集，例如 ImageNet 数据集就有 1200 万张图片，而且从 2012 年之后，GPU 在深度学习中的应用成为常态，与 CPU 相比，可以提高几十倍的计算效率，正是由于这些技术上的进步，使得多层感知器（MLP）模型这一古老的模型具有越来越大的应用前景。

5.1 数学原理

5.1.1 前向传播过程

在本节中，我们想向大家介绍多层感知器（MLP）模型的表示方式，这同时是各种神经网络的表示基础。

我们假定第 $l-1$ 层具有 N_{l-1} 个神经元，其输出信号为一个向量 $\boldsymbol{a}^{l-1}$：

$$\boldsymbol{a}^{l-1}=\begin{bmatrix}a_1^{l-1}\\a_2^{l-1}\\...\\a_{N_{l-1}}^{l-1}\end{bmatrix}\in R^{N_{l-1}} \tag{112}$$

其中第 j 个神经元的输出为 a_j^{l-1}，其上标代表是第 $l-1$ 层，下标代表是第 j 个神经元。第 l 层有 N_l 个神经元，第 $l-1$ 层和第 l 层神经元互相两两全部连接，连接权值 W 为：

$$W^l=\begin{bmatrix}W_{1,1}^l & W_{1,2}^l & ... & W_{1,N_{l-1}}^l\\W_{1,1}^l & W_{1,2}^l & ... & W_{1,N_{l-1}}^l\\... & ... & ... & ...\\W_{N_l,1}^l & W_{N_l,2}^l & ... & W_{N_l,N_{l-1}}^l\end{bmatrix} \tag{113}$$

第 $l-1$ 层第 j 个神经元指向第 l 层第 i 个神经元的连接权值为 $W_{i,j}^l$。
第 l 层的输入用向量 $\boldsymbol{z}^l$ 表示：

$$\begin{aligned}\boldsymbol{z}^l&=W^l\cdot\boldsymbol{a}^{l-1}+\boldsymbol{b}\\z_i^l&=W_{i,:}^l\cdot\boldsymbol{a}^{l-1}+b_i\\&=W_{i,1}a_1^{l-1}+W_{i,2}a_2^{l-1}+...+W_{i,N_{l-1}}a_{N_{l-1}}^{l-1}\\&=\sum_{j=1}^{N_{l-1}}W_{i,j}^l\cdot a_j^{l-1}+b_i\end{aligned} \tag{114}$$

我们假设第 l 层的激活函数为 $f(x)$，则第 l 层输出为：

$$\boldsymbol{a}^l=f(\boldsymbol{z}^l) \tag{115}$$

依次运行上述步骤，即可完成神经网络的前向传播过程，求出输出层的输出。

5.1.2 神经元激活函数

sigmoid 函数 在现代深度学习技术复兴之前，sigmoid 函数是神经元缺省的激活函数，因为 sigmoid 的函数在定义域内连续且处处可微，同时可以解释为事件出现的概率，所以成为神经元的缺省激活函数。但是近些年来，人们逐渐意识到，sigmoid 函数在其定义域内在值过大或过小时，会出现饱合情况，曲线变得非常平坦，这时求导时值就几乎为零，这使得基于求导的梯度下降法收敛很慢，这种情况在深度网络中尤其严重，会出现梯度消失的问题，在这期间传统的机器学习方法，例如支撑向量机（SVM）却没有这个问题，所以近年来，除了二元分类问题的输出层，长短时记忆网络（LSTN）的门限操作，等比较特殊的情形外，人们已经不再使用 sigmoid 函数作为神经元的激活函数了，在现代深度学习网络中，缺省的隐藏层激活函数是 ReLU 函数。ReLU 函数是一个分段线性函数，除零点外处处可导，不存在梯度消失问题，所有这些特点使得 ReLU 成为深度学习网络理想的神经元激活函数。

定义 sigmoid 函数定义如下所示：

$$\sigma(x)=\frac{1}{1+e^{-x}} \tag{116}$$

我们可以用如下代码绘制 sigmoid 函数的图形：

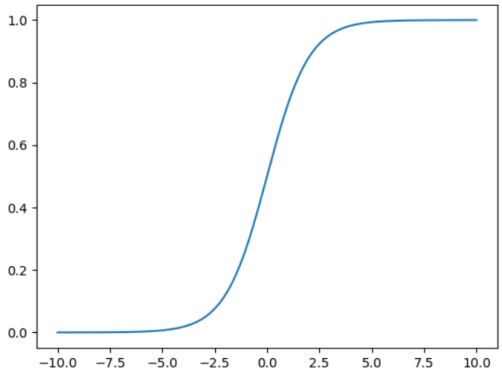
```
import numpy as np
import matplotlib.pyplot as plt
```

```
3
4  def main():
5      x = np.linspace(-10, 10, 300)
6      y = 1.0 / (1 + np.exp(-x))
7      plt.plot(x, y)
8      plt.show()
9
10 if '__main__' == __name__:
11     main()
```

Listing 29: 绘制 sigmoid 函数

上面代码绘制出来的 sigmoid 函数图形如下所示：

Figure 35: sigmoid 函数



如图35所示，当函数自变量 x 的值大于 5 或小于 -5 时，函数曲线就变得非常平坦了，这时导数将非常小，几乎接近于零，这时梯度下降算法收敛的速度就会相当慢，为了解决这一问题，研究人员提出了 ReLU 函数作为隐藏层神经元的激活函数，并逐渐成为深度学习网络中主流技术。我们将在本节稍后时间给大家讲解 ReLU 函数。

导数 对于一个可导的函数 $f(x)$，我们将其在 x_0 点的导数定义为：

$$\frac{dy}{dx} = lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} \tag{117}$$

这是导数的定义，我们验证求导函数是否正确时，就是用这种方式来进行正确性检验的。但是在深度学习中，我们不使用这个定义来求导，因为这种数值计算方式不仅运算量大，而且精度低，当前业界的主流求导技术是自动微分技术，这个技术我们会在本节最后给大家做一个简单的介绍。

回到求导问题上来，我们一般应用解析法来求导，这非常类似于我在学高等数学时的手工求导公式。高等数学中求导公式有很多，但是对于 sigmoid 函数求导来说，只需要用到如下两个公式：

$$(e^x)' = e^x \tag{118}$$

$$(\frac{u}{v})' = \frac{u' \cdot v - u \cdot v'}{v^2} \quad v \neq 0 \tag{119}$$

对于 sigmoid 函数而言，分母 $u = 1$, 式119就可以简化为：

$$(\frac{1}{v})' = \frac{-v'}{v^2} \quad v \neq 0 \tag{120}$$

根据以上公式，sigmoid 导数为：

$$\begin{aligned}\sigma(x)' &= \frac{-(1+e^{-x})'}{(1+e^{-x})^2} = -\frac{-e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\ &= \sigma(x) \cdot \left(1 - \sigma(x)\right)\end{aligned} \tag{121}$$

ReLU 函数 ReLU 函数的定义为：

$$g(\boldsymbol{z}) = \max\{0, z\} \tag{122}$$

用我们常用的数学语言来描述，ReLU 就是一个分段函数：

$$g(\boldsymbol{z}) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases} \tag{123}$$

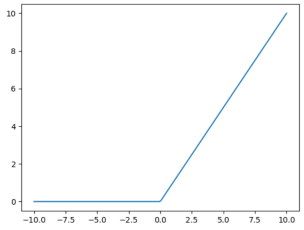
大家可以看到，在 $z < 0$ 时，函数的导数为零，当 $z > 0$ 时，函数的导数为常数 1，不存在饱合问题，同时具有线性函数的简洁性，降低了深度神经网络的训练难度。可以通过如下代码绘制 ReLU 函数：

```python
import numpy as np
import matplotlib.pyplot as plt

def relu(x):
    s = np.where(x<0, 0, x)
    return s


def relu_grad(s):
    ds = np.where(s<0, 0, 1)
    return ds

def main():
    x = np.linspace(-10, 10, 200)
    y = relu(x)
    plt.plot(x, y)
    plt.show()

if '__main__' == __name__:
    main()
```

Listing 30: 绘制 ReLU 函数

ReLU 函数的图形如下所示：

Figure 36: ReLU 函数图像



ReLU 函数与 Sigmoid 函数正好相反，虽然在 $x=0$ 时不可微，但是其他处均可微，现代深度学习理论认为，只要函数绝大多数地方可微就可以了。另外，ReLU 函数也不存在饱合性，当 $x>0$ 时，其微分值始终为 1，可以加快梯度下降算法的收敛速度。同时，ReLU 函数很好地模拟了生物神经元刺激在一定阈值下不反应，达到这个阈值后起反应，但是和刺激强度无关的特性。正是由于采用了 ReLU 函数，才使得现代的深度学习网络取得成功。

5.1.3 深度学习中的微分运算

深度学习当中，函数微分是一项非常重要的任务。关于计算机实现函数微分，总结起来有以下三种方式：第一种直接按导数定义的数值计算方法，这种方法的优点是简单，但是缺点是计算量大并且精度低，通常只用于验证其他微分方法的正确性；第二种方法是解析法，就是像我们在上一节对 sigmoid 函数那样，手工求出函数微分的表达式，这种方式的精度高，但是很多函数不容易解出微分的表达式，同时利用隐函数表示的函数，也很难求出微分表达式；第三种方法就是未来可能会流行起来的自动微分方法，深度学习三巨头之一的 Yann LeCun 在去年所说的“深度学习已死，可微编程永生”就是指的这种方法。这一部分内容可以参考我的博客**?**。

在这一节中，我们将以一个简单的逻辑回归模型，来带领大家掌握三种主流的求微分的方式，同时向大介绍自动微分技术。去年深度学习三巨头之一 Yann LeCun 曾经说：“深度学习已死，可微编程永生”，他说的可微编程就是自动微分技术，所以有必要介绍一下这个最新技术。

在逻辑回归模型中，我们的输入为向量 $\boldsymbol{x}$，与权值向量 $\boldsymbol{w}$ 相乘，再加上偏置值 b，再将得到的结果取 Sigmoid 函数，最后就得到了一个 0∼1 的判别结果，如下所示：

$$\hat{y}=\frac{1}{1+e^{-(\boldsymbol{w}\cdot\boldsymbol{x}+b)}} \tag{124}$$

数值法 数值法求微分就是利用导数的定义来求微分值，根据高等数学导数的定义：

$$f(x)'=\lim_{x\to x_0}\frac{f(x)-f(x_0)}{x-x_0} \tag{125}$$

在实际中为了有更好的效果，我们通常采用这种方式来计算微分：

$$f(x)'=\lim_{h\to 0}\frac{f(x+h)-f(x-h)}{2\cdot h} \tag{126}$$

公式法求微分 公式法就是根据高等数学公式，求出函数微分的表达式，将数值代入表达式求出微分值。实际上我们在 Sigmoid 函数一节中，就是用这种方式求的微分。具体求导过程如下所示：

$$\begin{aligned}\hat{y}=\frac{d(\frac{1}{1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b}})}{d\boldsymbol{x}}&=-\frac{(1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b})'}{(1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b})^2}=-\frac{(e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b})'}{(1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b})^2}\\&=-\frac{e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b}\cdot(-\boldsymbol{w}^T\cdot\boldsymbol{x}-b)'}{(1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b})^2}\\&=\frac{e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b}}{(1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b})^2}\cdot\boldsymbol{w}\\&=\frac{1}{1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b}}\frac{e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b}}{1+e^{-\boldsymbol{w}^T\cdot\boldsymbol{x}-b}}\cdot\boldsymbol{w}\\&=\hat{y}(1-\hat{y})\boldsymbol{w}\end{aligned} \tag{127}$$

这里用的技巧就是高等数学在复合函数求导中用的换元法，就是把复杂的部分视为一个整体先求导，然后再乘以对这个部分进行求导，这个过程一直进行下去，直到求到自变

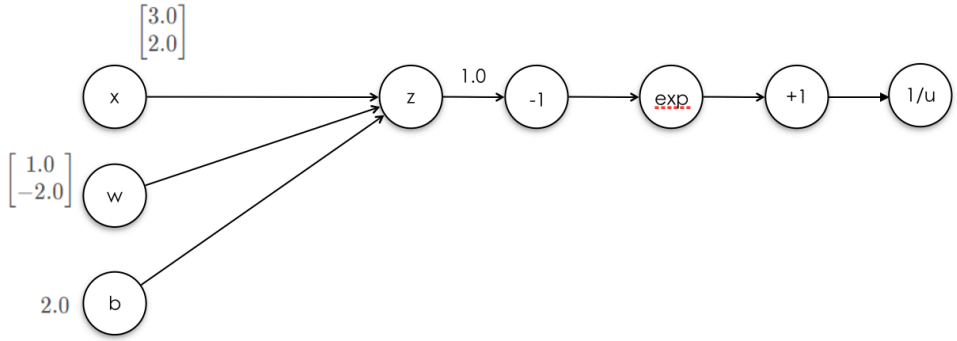
量时为止。另外，注意这里是对输入信号的微分，而不是我们在深度学习中经常使用的对连接权值的微分，但是这一技术确实是有用的，在生成对抗样本时就是使用这一技术。

计算图法 计算图是当前深度学习框架所用到的主流技术，可以在计算图上直接运行 BP 算法，得到网络参数的微分。下面我们就以逻辑回归用到的输出层激活函数为例：

$$y = \frac{1}{1 + e^{-(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)}} \tag{128}$$

我们假设我们求不出这个函数的导数，只是将这个函数拆成简单函数的组合，对应的计算图如下所示：

Figure 37: 计算图基本形式



图中第 1 个需要计算的节点就是 z 节点，计算公式为：

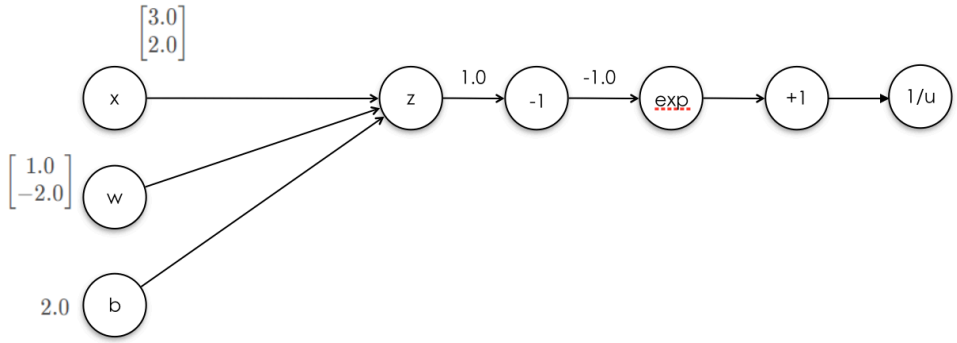
$$z = \boldsymbol{w}^T \cdot \boldsymbol{x} + b = \begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix}^T \cdot \begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} + 2.0 = 1.0 \tag{129}$$

计算出来的值我们写在 z 节点后，箭头线的上方。
接着我们来算 -1 这个节点，计算公式如下所示：

$$-u = -1.0 \tag{130}$$

将计算结果写在 -1 这个节点右侧箭头线的上方，如下图所示：

Figure 38: 计算图基本形式

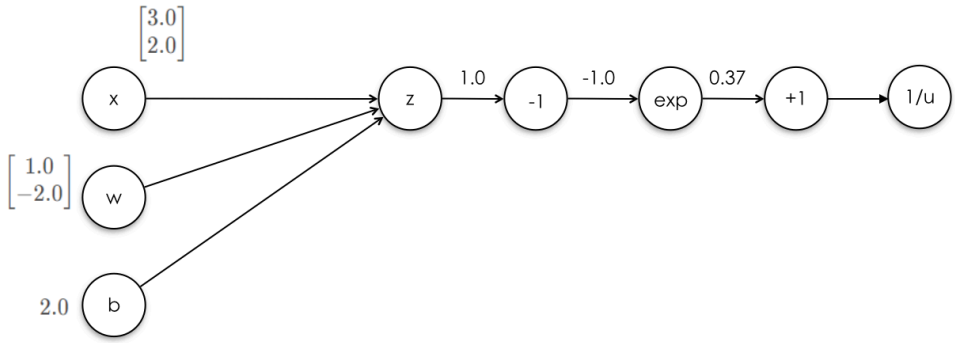


我们接下来计算 exp 节点，计算公式如下所示：

$$e^u = e^{-1} = -\frac{1}{2.718281828459} = 0.37 \tag{131}$$

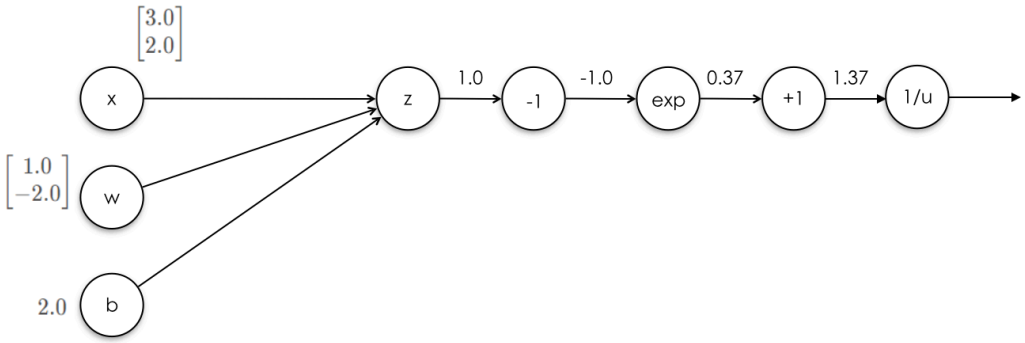
将计算结果写到 exp 节点右侧箭头线的上方，如下图所示：

Figure 39: 计算图基本形式



接下来一步是计算求 +1 节点，并把计算结果写到右侧箭头线上方，如下图所示：

Figure 40: 计算图基本形式

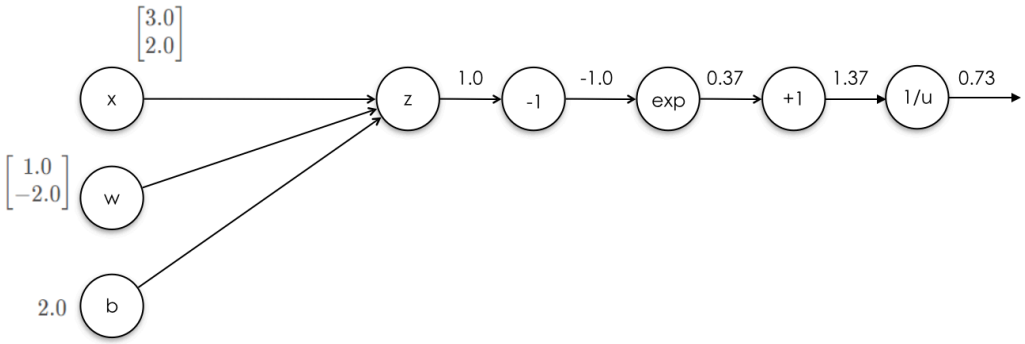


最后我们计算 1/u 节点，计算公式如下所示：

$$\frac{1}{u} = \frac{1}{1.37} = 0.73 \tag{132}$$

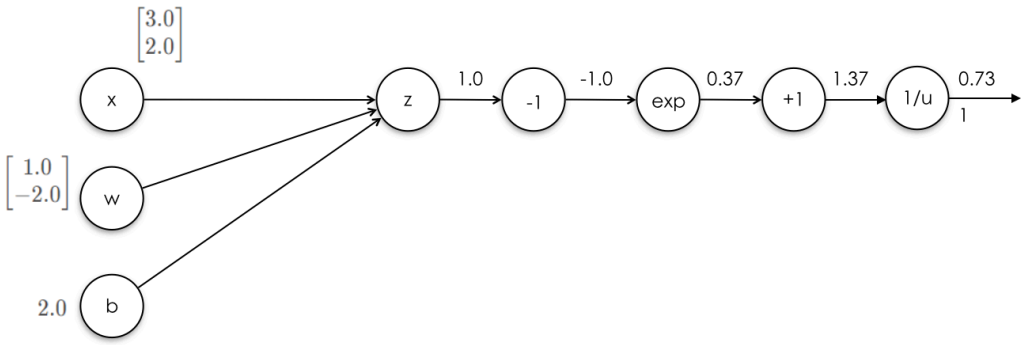
将求出的结果写到 1/u 节点的右侧箭头线的上方，如下图所示：

Figure 41: 计算图基本形式



以上就是信号正向传播的过程。接下来就是反向求微分值的过程。我们第一步是在计算图最后一步，即 1/u 节点右侧箭头线下方写上 1，代表输出信号对自身的导数，如下图所示：

Figure 42: 计算图基本形式

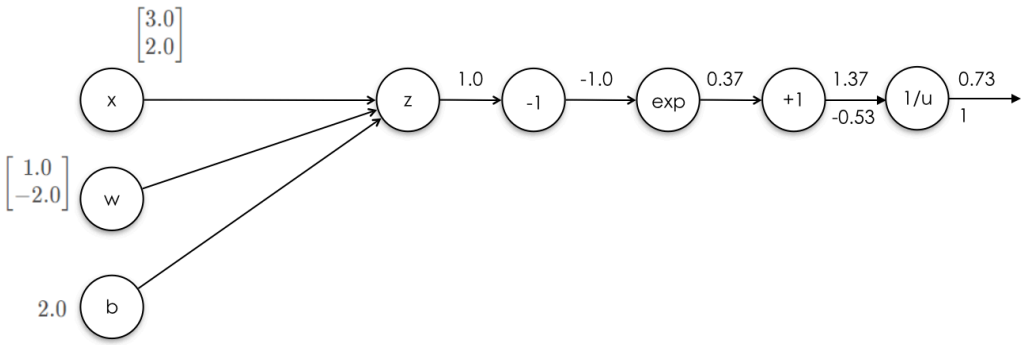


接下来我们来看反向传播的第一个节点 1/u，该节点的微分为：

$$\begin{aligned} & u = 1.37 \\ (1) \cdot \frac{d(\frac{1}{u})}{du} &= -\frac{1}{u^2} = -\frac{1}{1.37^2} = -0.53 \end{aligned} \tag{133}$$

将微分值写在 1/u 节点左侧箭头线的下方，如下图所示：

Figure 43: 计算图基本形式

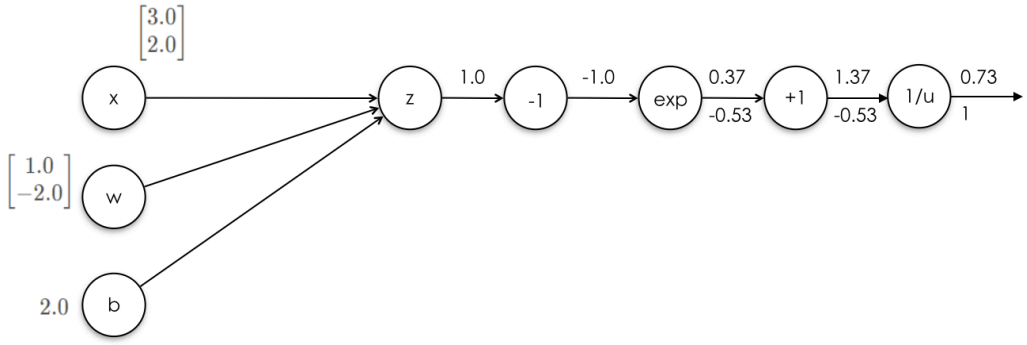


接着我们求 u+1 节点，计算公式如下所示：

$$(-0.53) \cdot \frac{d(u+1)}{du} = (-0.53) \cdot 1 = -0.53 \tag{134}$$

将计算出的微分值写在 +1 节点左侧箭头线下方，如下所示：

Figure 44: 计算图基本形式

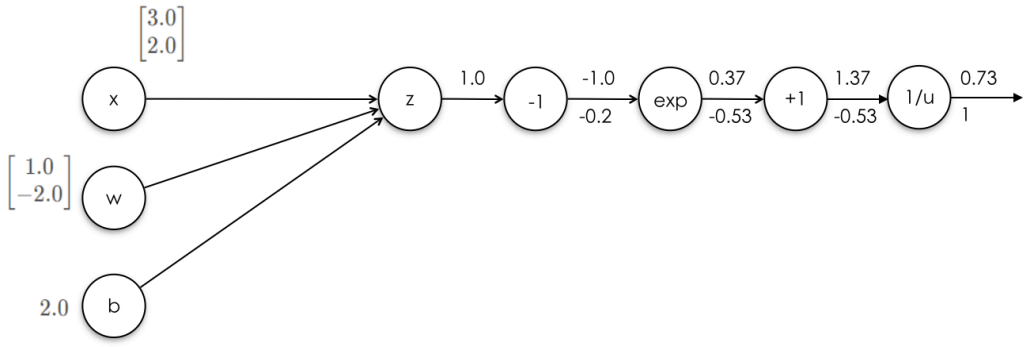


接下来我们处理 exp 节点，其微分计算公式为：

$$(-0.53) \cdot \frac{de^u}{du} = (-0.53) \cdot e^u = (-0.53) \cdot 0.37 = -0.20 \tag{135}$$

式中 e^u 可以直接取正向传播时计算出的值，不必重新计算。将计算出的微分值写在 exp 节点左侧箭头线下方，如下图所示：

Figure 45: 计算图基本形式

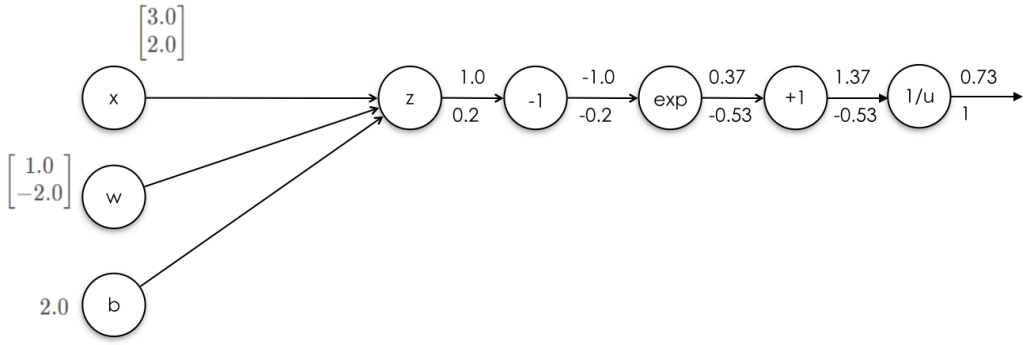


接下来我们处理取 -1 这个节点，微分计算公式为：

$$(-0.20) \cdot \frac{d(-u)}{du} = (-0.20) \cdot (-1) = 0.20 \tag{136}$$

将微分值写在取 -1 节点左侧箭头线下方，如下图所示：

Figure 46: 计算图基本形式

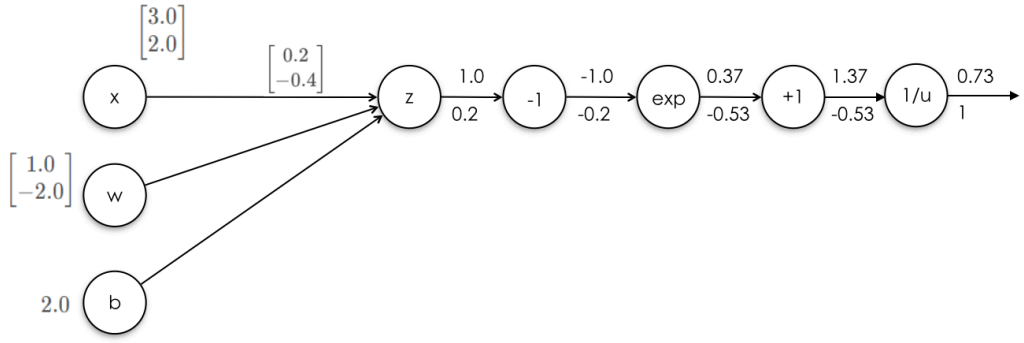


我们来处理 z 节点，为了演示概念，我们对 $\boldsymbol{x}$ 也进行了求导，但是实际上我们只需要对网络参数进行求导。我们先对 $\boldsymbol{x}$ 进行求导，微分计算公式为：

$$(0.20) \cdot \frac{d(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)}{d\boldsymbol{x}} = (0.20) \cdot \boldsymbol{w} = 0.20 \cdot \begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix} = \begin{bmatrix} 0.20 \\ -0.40 \end{bmatrix} \tag{137}$$

将求出的微分值写在 $\boldsymbol{z}$ 与 $\boldsymbol{x}$ 箭头线下方，如下图所示：

Figure 47: 计算图基本形式

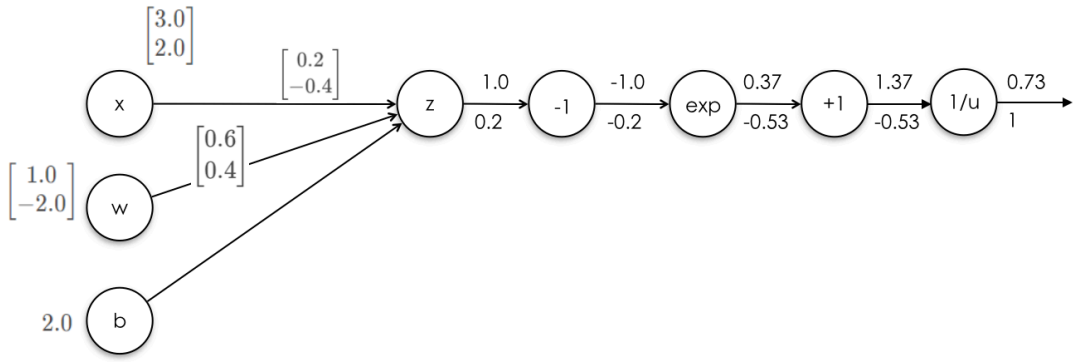


接下来我们对 $\boldsymbol{z}$ 对 $\boldsymbol{w}$ 的导数，微分计算公式为：

$$(0.20) \cdot \frac{d(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)}{d\boldsymbol{w}} = (0.20) \cdot \boldsymbol{x} = 0.20 \cdot \begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} = \begin{bmatrix} 0.60 \\ 0.40 \end{bmatrix} \tag{138}$$

将计算出的微分值写在 $\boldsymbol{z}$ 与 $\boldsymbol{w}$ 之间箭头线的下方，如下图所示：

Figure 48: 计算图基本形式

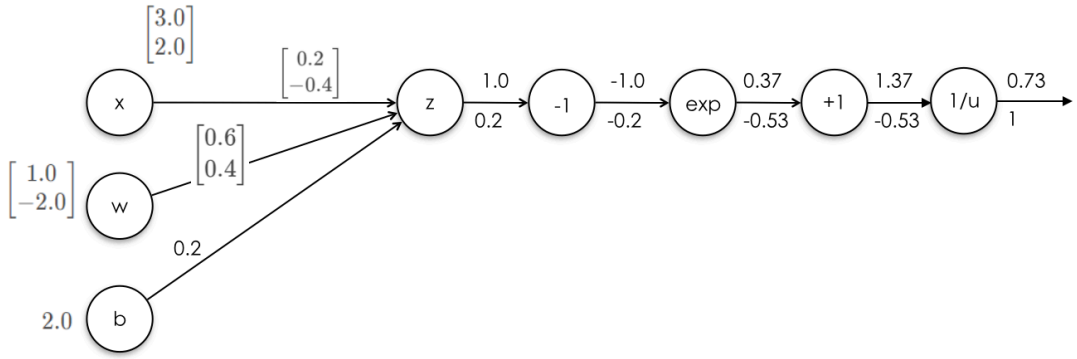


最后我们求 $\boldsymbol{z}$ 对偏置值 b 的导数，计算公式如下所示：

$$(0.20) \cdot \frac{d(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)}{d\boldsymbol{b}} = (0.20) \cdot 1 = 0.20 \tag{139}$$

将计算出的微分值写在 $\boldsymbol{z}$ 与 b 之间箭头线的下方，如下图所示：

Figure 49: 计算图基本形式



以上就是基于计算图的 BP 算法的计算过程，我们可以看出，无论多么复杂的函数，我们都能通过计算图求出该函数的导数。但是实际上我们还可以做的更好，因为基于计算图的模型中，我们为了反向传播，我们需要存储所有正向传播过程中的变量值，同时需要一步步反向运行正向过程。所以最近比较火的自动微分，就是重新建立一个用于微分的计算图，只需记录微分中用到的变量值，同时我们还可以根据函数求导的特点，设计简化的计算图。

自动微分 如果关心深度学习的人，一定记得 Yann LeCun 说过：“深度学习已死，可微分编程永生”的话，就是说深度学习技术太受限了，可微分编程由于其灵活的特性，会成为未来具有巨大发展潜力的技术。而可微分编程的关键技术就是自动微分。自动微分就是重新构建一个计算微分的计算图，只需要将计算微分需要用到的变量值，保存到微分计算图中即可，然后就可以用与前向传播相同的方法来计算微分了。下面我们还以逻辑回归为例，来讲解自动微分技术。

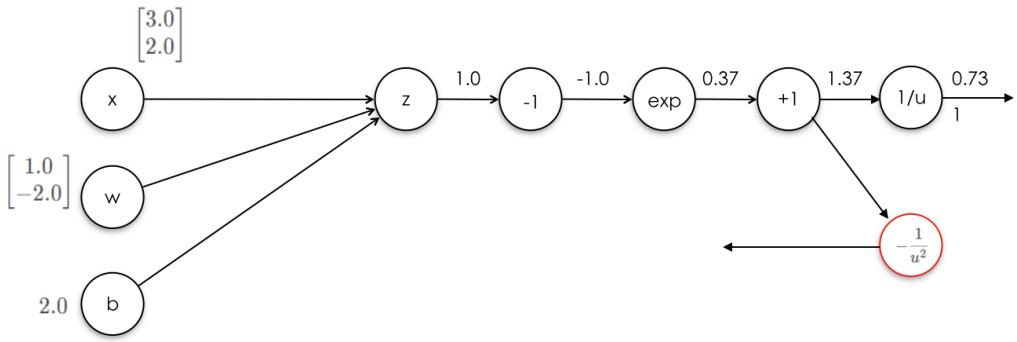
与计算图类似，我们首先完成信号的前向传播过程，接下来我们首先处理 1/u 节点，其计算

公式为：

$$\frac{d(\frac{1}{u})}{du} = -\frac{1}{u^2} \tag{140}$$

我们在正向计算图下面，画出我们微分计算图的第一个节点，如下图红色的节点：

Figure 50: 自动微分

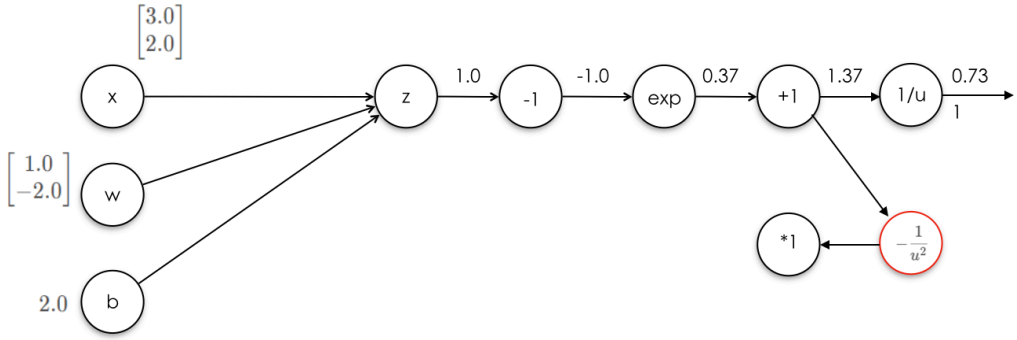


由于计算中需要 u 值，所以将正向传播时的 u 值保存到新节点中。
接下来我们处理 +1 节点，计算公式如下所示：

$$\frac{d(u+1)}{du} = 1 \tag{141}$$

所以我们需要在自动微分计算图中添加一个 *1 的节点，这时我们就不再需要保存前向传播的变量值了，这样就节省了空间，如下图所示：

Figure 51: 自动微分

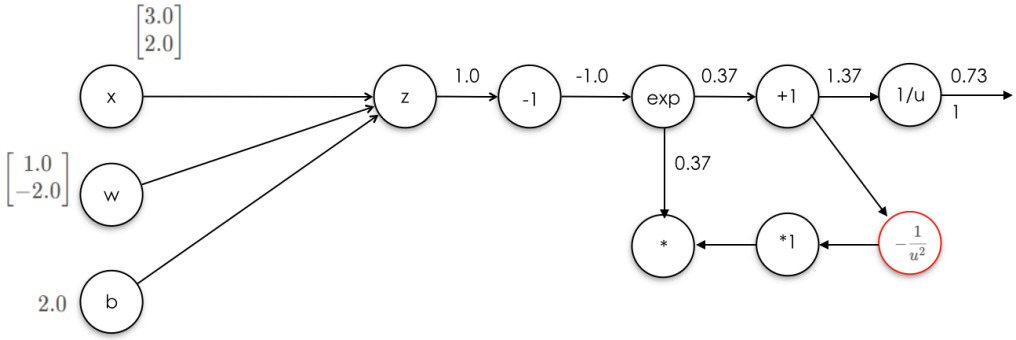


接下来我们来处理 exp 节点，计算公式如下所示：

$$\frac{de^u}{du} = e^u \tag{142}$$

由于要计算中要用到 e^u 的值，所以需要在自动微分计算图中保存这个值，如下图所示：

Figure 52: 自动微分

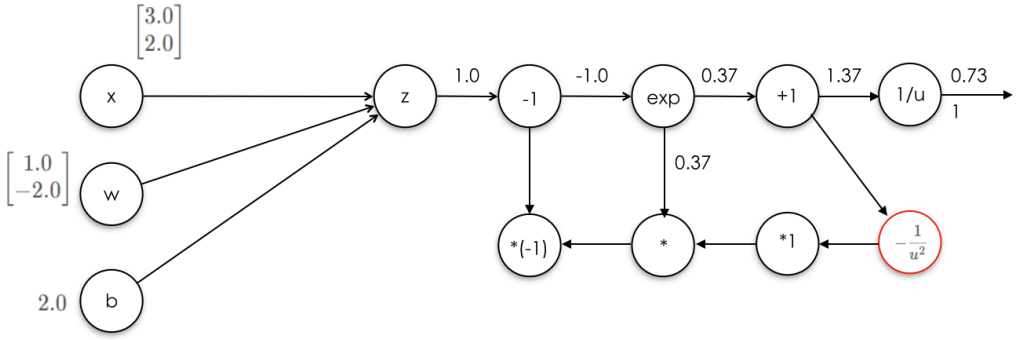


接下来我们处理取负节点，计算公式如下所示：

$$\frac{d(-u)}{du} = -1 \tag{143}$$

所以我们添加 *(-1) 节点，如下图所示：

Figure 53: 自动微分

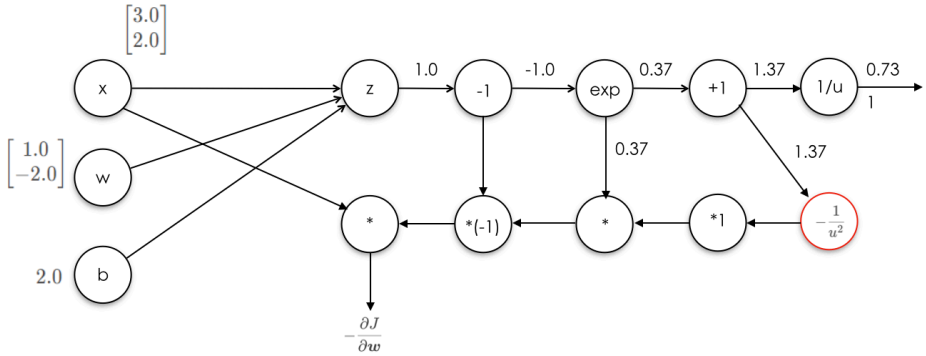


接下来我们处理 z 节点，我们这里先求 z 对 x 的微分，计算公式如下所示：

$$\frac{d(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)}{d\boldsymbol{w}} = \boldsymbol{x} \tag{144}$$

我们在自动微分计算图中添加如下节点：

Figure 54: 自动微分

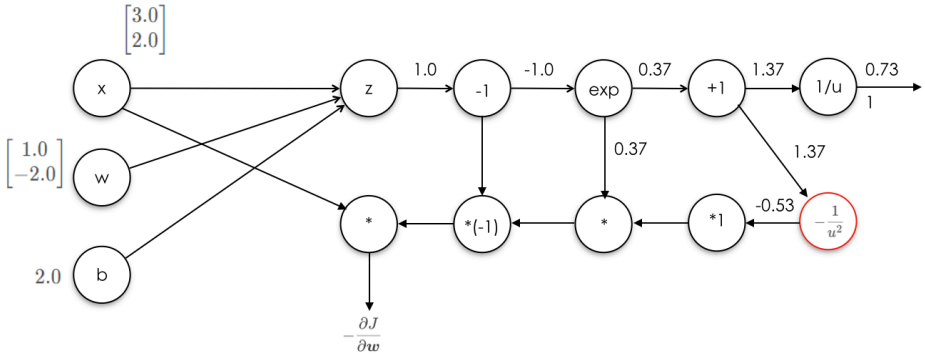


我们在这里就仅举 $\boldsymbol{z}$ 对网络参数 $\boldsymbol{w}$ 的微分为例。当有了计算图之后，我们求微分就和前向传播计算方式没有区别了。首先微分计算图的输入为 1.37，运行第 1 个节点公式可得：

$$-\frac{1}{u^2} = -\frac{1}{1.37^2} = -0.53 \tag{145}$$

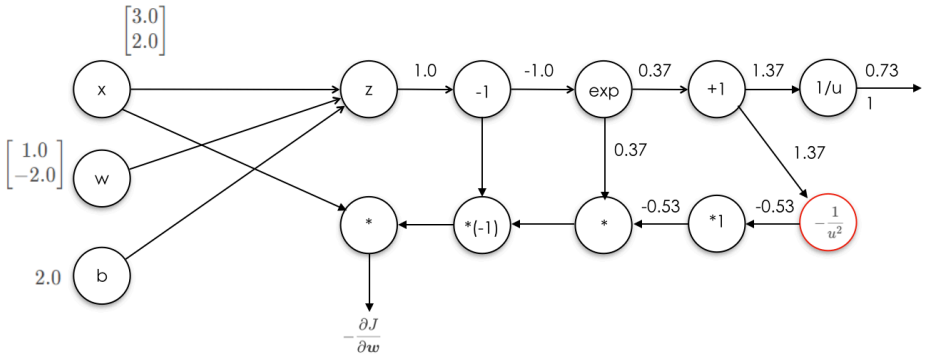
将计算结果写到图上，如下图所示：

Figure 55: 自动微分



接下来我们处理 *1 节点，该节点只需简单的将前一步的微分值乘 1 即可，得到结果如下所示：

Figure 56: 自动微分

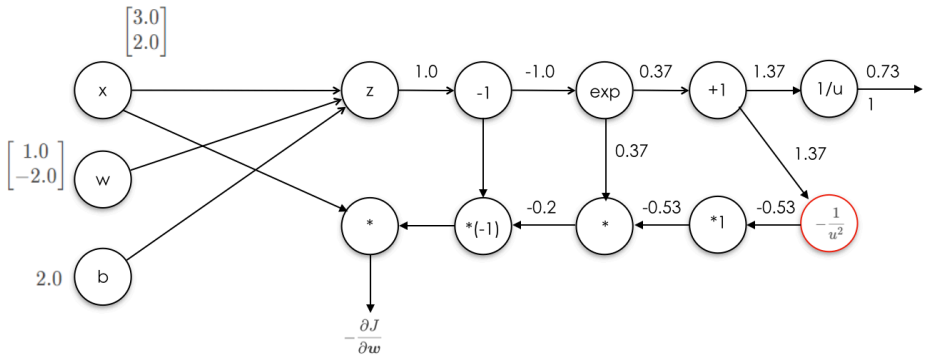


接下来我们处理 * 这个节点，计算公式为：

$$(-0.53) * e^u = (-0.53) * 0.37 = -0.20 \tag{146}$$

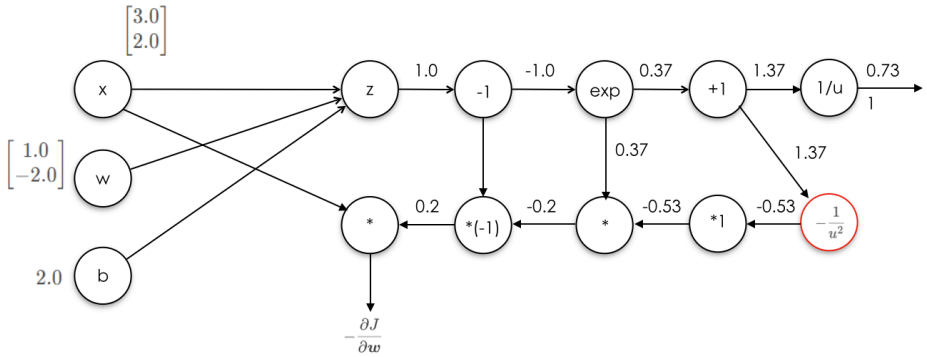
处理结果如下所示：

Figure 57: 自动微分



接下来处理 *(-1) 节点，只是在上一步微分结果上乘以 -1 即可，如下所示：

Figure 58: 自动微分

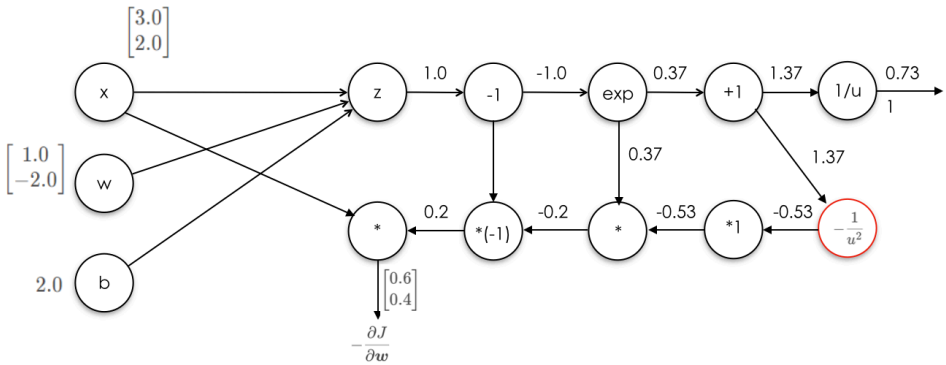


接下来处理 * 节点，其是将上一步的微分值乘以 $\boldsymbol{x}$，如下所示：

$$(0.20) \cdot \boldsymbol{x} = (0.20) \cdot \begin{bmatrix} 3.0 \\ 2.0 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix} \tag{147}$$

结果如下图所示：

Figure 59: 自动微分

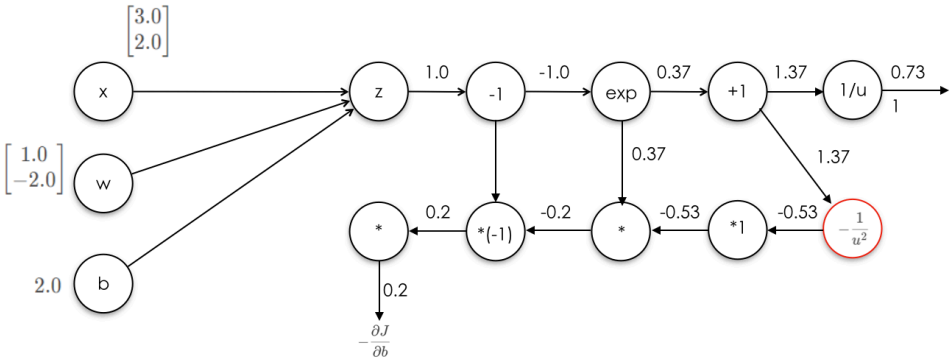


接着我们来看对偏置值 b 的导数，计算公式如下所示：

$$\frac{d(\boldsymbol{w}^T \cdot \boldsymbol{x} + b)}{db} = 1 \tag{148}$$

所以最左边的节点就变成 *1 了，我们求出 $\boldsymbol{z}$ 对 b 的微分，结果如下所示：

Figure 60: 自动微分



5.1.4 softmax 和 Cross Entropy

我们在深度学习中遇到的最常见的问题就多分类问题，这时我们神经网络的输出层的激活函数 softmax 函数，代价函数取的是 Cross Entropy 函数。比较遗憾的是，关于这部数学原理的详细描述非常少，大部分都直接给出 Cross Entropy 用于 softmax 情形下的公式，直接拿来用就可以了。所以在这一节中，我们将详细介绍 softmax 函数和 Cross Entropy 函数应用于多分类问题的物理意义和数学推导过程。

多分类问题的表示 我们在这里仍然以 MNIST 手写数字识别任务为例，我们要识别的类别为 10 类，分别为数字 0∼9，我们通常用 one-hot 向量形式来表示：

$$\boldsymbol{y} = \begin{bmatrix} y_1 & y_2 & y_3 & \dots & y_k & \dots & y_{10} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \dots & 1 & \dots & 0 \end{bmatrix} \tag{149}$$

其中等于 1 的维所对应的数字就是我们希望的识别结果。
通常我们神经网络的输出层为 softmax 函数，代表 0∼9 这 10 个数字出现的概率，并且这些概率之和为 1。

$$\hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \hat{y}_3 & \dots & \hat{y}_k & \dots & \hat{y}_{10} \end{bmatrix} \tag{150}$$

在数学上，我们可以把这个问题看成是有 10 个随机变量，分别对应 0∼9 的出现事件，而对于某个数字对应的事件，其值只可能是出现和不出现两种，其符合我们数学上的 Bernoulli 分布，整个问题也就由这 10 个 Bernoulli 分布组成。我们首先来研究某个数字的 Bernoulli 分布。
我们需要明确一点，我们现在研究的是识别任务的目标，也就是对应的 $\boldsymbol{y}$,Bernoulli 分布的定义如下所示：

$$\begin{aligned} P(y=1) &= \phi \\ P(y=0) &= 1-\phi \\ P(Y=y) &= \phi^y \cdot (1-\phi)^{1-y} \end{aligned} \tag{151}$$

式151中，ϕ 代表该事件出现的概率。

信息论简介 信息论本来是研究在不可靠信道中，例如移动通信，怎样以最小的码长来可靠的传递信息的学科，理论体系非常庞大和复杂，对深度学习来讲，我们只需要知道在信息论中，如果知道最可能发生的事情发生了，那么我们得到的信息量非常小，而如果我们知道不可能发生的事件发生了，则我们得到的信息量将非常大。例如，如果我们知道今天太阳从东方升起，那么我们从中得不到任何有价值的信息，而如果我们知道今天早晨下了雪，路上湿滑，那么我们就知道应该注意出行安全，得到较多的有用信息。
对于一个随机变量 x，取每个可能的值（在这里我们只计论离散值情况）就是一个事件 $x = x$，我们可以定义 self information：

$$I(x) = -\log\big(P(x)\big) \tag{152}$$

在这里我们取的是以 e 为底的对数，单位是 nats。
如果我们把该随机变量所有出现的事件都集中起来，就得到信息论中的香农熵：

$$H(x) = E_{x\sim P}(I(x)) = -E_{x\sim P}\big(\log P(x)\big) \tag{153}$$

从式153可以看出，香农熵可以定义为当 x 符合 P 分布时，x 的 self information 的希望值。
我们可以将在 MNIST 数据集上手写数字识别任务中，正确答案 $\boldsymbol{y}$ 对某个数字的识别视为一个 Bernoulli 分布，将我们神经网络输出层 softmax 函数值输出中某个数字的识别，视为另一个 Bernoulli 分布，我们的目标就是让这两个分布尽可能接近。在信息论，对这个问题可以用 KL 散度来表示，我们假设正确答案所对应的分布为 P，我们神经网络输出的分布为 Q，则 KL 散度定义为：

$$D_{KL}(P\|Q) = E_{x\sim P}\left[\log\frac{P(x)}{Q(x)}\right] = E_{x\sim P}\Big[\log P(x) - \log Q(x)\Big] \tag{154}$$

KL 散度的值永远为正，而且当 P 和 Q 越接近其值越小，当 P 和 Q 是同一分布时，其值为零。但是 KL 散度有一个问题，就是 $D_{KL}(P\|Q)$ 与 $D_{KL}(Q\|P)$ 不相等，不完全等同于 P 和 Q 的距离。这就引出了我们的交叉熵 Cross Entropy 的定义：

$$H(P,Q) = H(P) + D_{KL}(P\|Q) = -E_{x\sim P}\Big[\log Q(x)\Big] \tag{155}$$

多分类问题的交叉熵表示 回到我们 MNIST 手写数字识别任务，我们假设正确答案对某个数字识别的分布为 P，神经网络输出对某个数字的概率的分布为 Q，我们假设要研究的是第 k 维，即 $y_k = 1$，则有：

$$CrossEntropy = H(P) + D_{KL}(P\|Q) = -E_{x\sim P}\Big[\log Q(x)\Big] = -\frac{1}{N_k}\sum_{i=1}^{N_k}\log\hat{y}_k^{(i)} \tag{156}$$

其中 N_k 表示训练样本集中第 k 维等于 1 的样本数，我们对这些样本求出均值就是交叉熵函数了。为了便于处理，我们根据正确答案的特点，只有第 k 维是 1，其余全为零，我们可以得到如下的计算公式：

$$\begin{aligned} CrossEntropy = H(P) + D_{KL}(P\|Q) = -E_{x\sim P}\Big[\log Q(x)\Big] &= -\frac{1}{N_k}\sum_{i=1}^{N_k}\log\hat{y}_k^{(i)} \\ &= -\frac{1}{N_k}\sum_{i=1}^{N_k}\sum_{k=1}^{K=10} y_k\cdot\log\big(\hat{y}_k^{(i)}\big) \end{aligned} \tag{157}$$

这个公式就是其他文档或教程里给出的 Cross Entropy 的公式，希望大家可以通过我们上面的推导过程，对交叉熵和 softmax 函数有更加深入的理解。
以上我们讨论的是针对识别某个特定数字情况，而实际中有 10 个数字，处理方式完全相同，在训练样本集中包括所有这 10 个数字的识别样本，我们只需要将这些情况加在一起就可以了。

softmax 求导 softmax 函数求导我们将以一种微稍不同的方式来讲解，大家在网络上或者教科书，对于 softmax 函数求导，都在在 BP 框架下进行的。但是当前流行的深度学习框架，如 TensorFlow、Pytorch 等，都是基于计算图模式，二者虽然原理相同，但是在具体处理方式上面，却有比较大的差异，在这一节中，我们将以 PyTorch 中采用的计算图的方式来进行讲解。

对我们这个神经网络而言，我们采用 Cross Entropy 作为代价函数：

$$L = -\sum_{i=1}^{K=10} y_i \cdot \log\left(\hat{y}_i\right) = -\log\left(\hat{y}_k\right) \tag{158}$$

我们在这里假设第 k 维为 1，其余维均等于 0。

接下来我们定义代价函数对神经网络输出 $\hat{\boldsymbol{y}}$ 求微分，这是一个标量对一个向量求微分，我们定义其结果为一个 1 行 N 列的矩阵，其中 N 为向量的维数，此时的代价函数只有第 k 维有值，其余维的值为 0，所以可以表示为：

$$\begin{aligned}\frac{\partial L}{\partial \hat{\boldsymbol{y}}} &= \begin{bmatrix} 0 & 0 & \ldots & \left(-\log \hat{y}_k\right)' & \ldots & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & \ldots & -\frac{1}{\hat{y}_k} & \ldots & 0 \end{bmatrix}\end{aligned} \tag{159}$$

在这里我们用到了高等数学中的求导公式 $(\log x)' = \frac{1}{x}$。求出代价函数对输出层输出的微分之后，我们需要求输出层输出 $\hat{\boldsymbol{y}}$ 对输出层输入 $\boldsymbol{a}^2$ 的导数，这是一个向量对向量的导数，其中 $\hat{\boldsymbol{y}} \in R^{N_2}$ 且 $\boldsymbol{a}^2 \in R^{N_2}$，其中 $N_2 = 10$。向量对向量的导数就是 Jacobian 矩阵，其定义为：

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{a}^2} = \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial a_1^2} & \frac{\partial \hat{y}_1}{\partial a_2^2} & \cdots & \frac{\partial \hat{y}_1}{\partial a_{N_2}^2} \\ \frac{\partial \hat{y}_2}{\partial a_1^2} & \frac{\partial \hat{y}_2}{\partial a_2^2} & \cdots & \frac{\partial \hat{y}_2}{\partial a_{N_2}^2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial \hat{y}_{N_2}}{\partial a_1^2} & \frac{\partial \hat{y}_{N_2}}{\partial a_2^2} & \cdots & \frac{\partial \hat{y}_{N_2}}{\partial a_{N_2}^2} \end{bmatrix} \tag{160}$$

式160中 $N_2 = 10$，所以式中的 Jacobian 矩阵是一个 $R^{10\times 10}$ 的方阵。

下面我们来具体求解 softmax 函数的微分，由 sfotmax 函数定义可得：

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{161}$$

对于 $\frac{\partial \hat{y}_i}{\partial z_j}$，我们分两种情况来讨论。

当 $i = j$ 时：

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial z_i} &= \frac{\left(e^{z_i}\right)' \cdot \sum_{j=1}^{K} e^{z_j} - e^{z_i} \cdot \left(\sum_{j=1}^{K} e^{z_j}\right)'}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} \\ &= \frac{e^{z_i} \cdot \sum_{j=1}^{K} e^{z_j} - e^{z_i} \cdot e^{z_i}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} \\ &= \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \cdot \frac{\sum_{j=1}^{K} e^{z_j} - e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \\ &= \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \cdot \left(1 - \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right) \\ &= \hat{y}_i \cdot \left(1 - \hat{y}_i\right)\end{aligned} \tag{162}$$

当 $i \neq j$ 时：

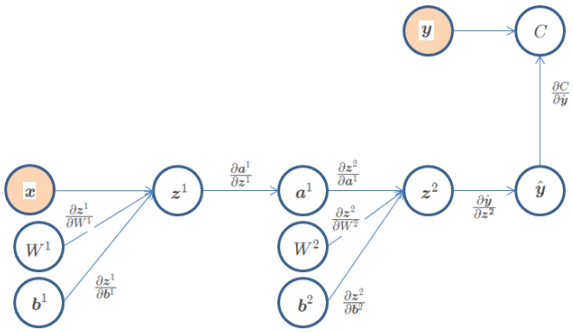
$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial z_j} &= e^{z_i} \cdot \left(-\frac{(\sum_{j=1}^{K} e^{z_j})'}{(\sum_{j=1}^{K} e^{z_j})^2} \right) \\ &= -\frac{e^{z_i} \cdot e^{z_j}}{(\sum_{j=1}^{K} e^{z_j})^2} \\ &= -\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \cdot \frac{e^{z_j}}{\sum_{j=1}^{K} e^{z_j}} \\ &= -\hat{y}_i \cdot \hat{y}_j\end{aligned} \tag{163}$$

综上所述，对 Jacobian 矩阵 $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^2}$，对角线上元素可以用式162来表示，而非对角线上元素可以用式163来表示。

5.1.5 计算图简介

我们所研究网络的计算图如下所示：

Figure 61: 神经网络计算图表示



计算图分为两个阶段：信号前向传播和误差反向传播。

正向传播 我们先来看信号前向传播过程，输入信号为原始图像 $\boldsymbol{x}$，可以得到第 1 层的输入：

$$\boldsymbol{z}^1 = W^1 \cdot \boldsymbol{x} + \boldsymbol{b}^1 \tag{164}$$

求出第 1 层神经元输入后，应用神经元激活函数，得到第 1 层的输出：

$$\boldsymbol{a}^1 = f(\boldsymbol{z}^1) \tag{165}$$

接着我们求出第 2 层的输入信号：

$$\boldsymbol{z}^2 = W^2 \cdot \boldsymbol{a}^1 + \boldsymbol{b}^2 \tag{166}$$

接着求出第 2 层的输出，由于第 2 层即为输出层，所以也就是求出 $\hat{\boldsymbol{y}}$：

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{167}$$

我们取出正确结果 $\boldsymbol{y}$，采用 Cross Entropy 作为代价函数，这里我们假设第 k 维等于 1，其他维均为零：

$$C = -\sum_{i=1}^{K} y_i \cdot \log(\hat{y}_i) = -\log(\hat{y}_k) \tag{168}$$

反向传播 反向传播阶段就是求出图26中在连接上的微分值。对我们这个神经网络而言，我们采用 Cross Entropy 作为代价函数：

$$L = -\sum_{i=1}^{K=10} y_i \cdot \log\left(\hat{y}_i\right) = -\log\left(\hat{y}_k\right) \tag{169}$$

我们在这里假设第 k 维为 1，其余维均等于 0。
接下来我们定义代价函数对神经网络输出 $\hat{\boldsymbol{y}}$ 求微分，这是一个标量对一个向量求微分，我们定义其结果为一个 1 行 N 列的矩阵，其中 N 为向量的维数，此时的代价函数只有第 k 维有值，其余维的值为 0，所以可以表示为：

$$\begin{aligned}\frac{\partial L}{\partial \hat{\boldsymbol{y}}} &= \begin{bmatrix} 0 & 0 & \dots & \left(-\log \hat{y}_k\right)' & \dots & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & \dots & -\frac{1}{\hat{y}_k} & \dots & 0 \end{bmatrix}\end{aligned} \tag{170}$$

在这里我们用到了高等数学中的求导公式 $(\log x)' = \frac{1}{x}$。求出代价函数对输出层输出的微分之后，我们需要求输出层输出 $\hat{\boldsymbol{y}}$ 对输出层输入 $\boldsymbol{z}^2$ 的导数，这是一个向量对向量的导数，其中 $\hat{\boldsymbol{y}} \in R^{N_2}$ 且 $\boldsymbol{z}^2 \in R^{N_2}$，其中 $N_2 = 10$。向量对向量的导数就是 Jacobian 矩阵，其定义为：

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^2} = \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial z_1^2} & \frac{\partial \hat{y}_1}{\partial z_2^2} & \dots & \frac{\partial \hat{y}_1}{\partial z_{N_2}^2} \\ \frac{\partial \hat{y}_2}{\partial z_1^2} & \frac{\partial \hat{y}_2}{\partial z_2^2} & \dots & \frac{\partial \hat{y}_2}{\partial z_{N_2}^2} \\ \dots & \dots & \dots & \dots \\ \frac{\partial \hat{y}_{N_2}}{\partial z_1^2} & \frac{\partial \hat{y}_{N_2}}{\partial z_2^2} & \dots & \frac{\partial \hat{y}_{N_2}}{\partial z_{N_2}^2} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \cdot (1 - \hat{y}_1) & -\hat{y}_1 \cdot \hat{y}_2 & \dots & -\hat{y}_1 \cdot \hat{y}_{N_2} \\ -\hat{y}_2 \cdot \hat{y}_1 & \hat{y}_2 \cdot (1 - \hat{y}_2) & \dots & -\hat{y}_1 \cdot \hat{y}_{N_2} \\ -\hat{y}_{N_2} \cdot \hat{y}_1 & -\hat{y}_{N_2} \cdot \hat{y}_2 & \dots & \hat{y}_{N_2} \cdot (1 - \hat{y}_{N_2}) \end{bmatrix} \tag{171}$$

式171中 $N_2 = 10$，所以式中的 Jacobzian 矩阵是一个 $R^{10\times 10}$ 的方阵。
下面我们来具体求解 softmax 函数的微分，由 sfotmax 函数定义可得：

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{172}$$

对于 $\frac{\partial \hat{y}_i}{\partial z_j}$，我们分两种情况来讨论。
当 $i = j$ 时：

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial z_i} &= \frac{\left(e^{z_i}\right)' \cdot \sum_{j=1}^{K} e^{z_j} - e^{z_i} \cdot \left(\sum_{j=1}^{K} e^{z_j}\right)'}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} \\ &= \frac{e^{z_i} \cdot \sum_{j=1}^{K} e^{z_j} - e^{z_i} \cdot e^{z_i}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} \\ &= \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \cdot \frac{\sum_{j=1}^{K} e^{z_j} - e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \\ &= \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \cdot \left(1 - \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right) \\ &= \hat{y}_i \cdot \left(1 - \hat{y}_i\right)\end{aligned} \tag{173}$$

当 $i \neq j$ 时：

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial z_j} &= e^{z_i} \cdot \left(-\frac{(\sum_{j=1}^{K} e^{z_j})'}{(\sum_{j=1}^{K} e^{z_j})^2} \right) \\ &= -\frac{e^{z_i} \cdot e^{z_j}}{(\sum_{j=1}^{K} e^{z_j})^2} \\ &= -\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \cdot \frac{e^{z_j}}{\sum_{j=1}^{K} e^{z_j}} \\ &= -\hat{y}_i \cdot \hat{y}_j\end{aligned} \tag{174}$$

综上所述，对 Jacobian 矩阵 $\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{a}^2}$，对角线上元素可以用式173来表示，而非对角线上元素可以用式174来表示。
接下来我们求 $\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1}$ 的值，我们首先计算单个值的微分：

$$\frac{\partial z_i^2}{\partial a_j^1} = \frac{\partial (W_{i,1}^2 \cdot a_1^1 + W_{i,2}^2 \cdot a_2^1 + ... + W_{i,j}^2 \cdot a_j^1 + ... + W_{i,N_1}^2 \cdot a_{N_1}^1 + b_1^2)}{\partial a_j^1} = W_{i,j}^2 \tag{175}$$

由式175可得：

$$\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1} = \begin{bmatrix} W_{1,1}^2 & W_{1,2}^2 & ... & W_{1,N_1}^2 \\ W_{2,1}^2 & W_{2,2}^2 & ... & W_{2,N_1}^2 \\ ... & ... & ... & ... \\ W_{N_2,1}^2 & W_{N_2,2}^2 & ... & W_{N_2,N_1}^2 \end{bmatrix} = W^2 \tag{176}$$

在接下来我们求 $\frac{\partial \boldsymbol{z}^2}{\partial W^2}$ 的值，这时我们求的是向量对矩阵的微分，我们可以将 W^2 视为一个行向量组成的向量，如下所示：

$$W^2 = \begin{bmatrix} W_{1,1}^2 & W_{1,2}^2 & ... & W_{1,N_1}^2 \\ W_{2,1}^2 & W_{2,2}^2 & ... & W_{2,N_1}^2 \\ ... & ... & ... & ... \\ W_{N_2,1}^2 & W_{N_2,2}^2 & ... & W_{N_2,N_1}^2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ ... \\ \boldsymbol{w}_{N_2} \end{bmatrix} \tag{177}$$

这样我们就把问题变为向量对向量求微分形式了，根据 Jacobian 矩阵定义可得：

$$\frac{\partial \boldsymbol{z}^2}{\partial W^2} = \begin{bmatrix} \frac{\partial z_1^2}{\partial \boldsymbol{w}_1} & \frac{\partial z_1^2}{\partial \boldsymbol{w}_2} & ... & \frac{\partial z_1^2}{\partial \boldsymbol{w}_{N_2}} \\ \frac{\partial z_2^2}{\partial \boldsymbol{w}_1} & \frac{\partial z_2^2}{\partial \boldsymbol{w}_2} & ... & \frac{\partial z_2^2}{\partial \boldsymbol{w}_{N_2}} \\ ... & ... & ... & ... \\ \frac{\partial z_{N_2}^2}{\partial \boldsymbol{w}_1} & \frac{\partial z^2}{\partial \boldsymbol{w}_2} & ... & \frac{\partial z_{N_2}^2}{\partial \boldsymbol{w}_{N_2}} \end{bmatrix} \tag{178}$$

在式178的矩阵中的每一项均是一个标量对向量的求导，结果为一个只有一行的矩阵，维度为向量的维度，所以其维度为 N_1。我们来看任意一项的微分 $\frac{\partial z_i^2}{\partial W_{j,k}^2}$。当 $i \neq j$ 时：此时连接权值 $W_{j,k}^2$ 不指向第 2 层第 i 个节点，此时求微分值为零，由此可知，式**??**中，非对角线上的元素，其值为元素值全为 0 的向量，维度为 N_1。当 $i = j$ 时：由第 2 层神经元输入信号计算公式：

$$z_i^2 = W_{i,1}^2 \cdot a_1^1 + W_{i,2}^2 \cdot a_2^1 + ... + W_{i,N_1}^2 \cdot a_{N_1}^1 + b_i^2 \tag{179}$$

可得：

$$\frac{\partial z_i^2}{\partial W_{i,k}^2} = \frac{\partial (W_{i,1}^2 \cdot a_1^1 + W_{i,2}^2 \cdot a_2^1 + ... + W_{i,N_1}^2 \cdot a_{N_1}^1 + b_i^2)}{\partial W_{i,k}^2} = a_k^1 \tag{180}$$

由式180可知，式178中不为零的对角线上元素为 $R^{1\times N_1}$ 只有一行的矩阵，其值为第 1 层神经元的输出。如下所示：

$$\frac{\partial \boldsymbol{z}^2}{\partial W^2}=\begin{bmatrix}\frac{\partial z_1^2}{\partial \boldsymbol{w}_1} & \frac{\partial z_1^2}{\partial \boldsymbol{w}_2} & \cdots & \frac{\partial z_1^2}{\partial \boldsymbol{w}_{N_2}}\\ \frac{\partial z_2^2}{\partial \boldsymbol{w}_1} & \frac{\partial z_2^2}{\partial \boldsymbol{w}_2} & \cdots & \frac{\partial z_2^2}{\partial \boldsymbol{w}_{N_2}}\\ \cdots & \cdots & \cdots & \cdots\\ \frac{\partial z_{N_2}^2}{\partial \boldsymbol{w}_1} & \frac{\partial z_{N_2}^2}{\partial \boldsymbol{w}_2} & \cdots & \frac{\partial z_{N_2}^2}{\partial \boldsymbol{w}_{N_2}}\end{bmatrix}=\begin{bmatrix}\boldsymbol{a}^1 & \boldsymbol{0} & \ldots & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{a}^1 & \ldots & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0} & \ldots & \boldsymbol{a}^1\end{bmatrix} \tag{181}$$

接下来我们求 $\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{b}^2}$，由 Jacobian 矩阵定义可得：

$$\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{b}^2}=\begin{bmatrix}\frac{\partial z_1^2}{\partial b_1^2} & \frac{\partial z_1^2}{\partial b_1^2} & \cdots & \frac{\partial z_1^2}{\partial b_{N_2}^2}\\ \frac{\partial z_1^2}{\partial b_1^2} & \frac{\partial z_1^2}{\partial b_1^2} & \cdots & \frac{\partial z_1^2}{\partial b_{N_2}^2}\\ \cdots & \cdots & \cdots & \cdots\\ \frac{\partial z_{N_2}^2}{\partial b_1^2} & \frac{\partial z_{N_2}^2}{\partial b_1^2} & \cdots & \frac{\partial z_{N_2}^2}{\partial b_{N_2}^2}\end{bmatrix}=\begin{bmatrix}1 & 0 & \ldots & 0\\ 0 & 1 & \ldots & 0\\ 0 & 0 & \ldots & 1\end{bmatrix} \tag{182}$$

对于矩阵中任意元素 $\frac{\partial z_i^2}{\partial b_j^2}$，其公式为：

$$\frac{\partial z_i^2}{\partial b_j^2}=\frac{\partial(W_{i,1}^2\cdot a_1^1+W_{i,2}^2\cdot a_2^1+\ldots+W_{i,N_1}^2\cdot a_{N_1}^1+b_i^2)}{\partial b_j^2} \tag{183}$$

当 $i\neq j$ 时，式183的值为 0，当 $i=j$ 时，其值为 1，所以最终结果为 $R^{N_2\times N_2}$ 的单位阵。采用同样方式，我们可以求出 $\frac{\partial \boldsymbol{z}^1}{\partial W^1}$ 和 $\frac{\partial \boldsymbol{z}^1}{\partial \boldsymbol{b}^1}$ 的微分，从而完成反向传播过程。注意：在反向传播过程中，我们只求与网络参数有关的微分，我们一般不求对 $\boldsymbol{x}$ 的微分。

网络参数求微分　计算完所有微分值后，我们就可求网络参数对于代价函数的偏微分了。

$$\frac{\partial C}{\partial W^2}=\frac{\partial C}{\partial \hat{\boldsymbol{y}}}\cdot\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^2}\cdot\frac{\partial \boldsymbol{z}^2}{\partial W^2} \tag{184}$$

$$\frac{\partial C}{\partial \boldsymbol{b}^2}=\frac{\partial C}{\partial \hat{\boldsymbol{y}}}\cdot\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^2}\cdot\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{b}^2} \tag{185}$$

$$\frac{\partial C}{\partial \boldsymbol{b}^1}=\frac{\partial C}{\partial \hat{\boldsymbol{y}}}\cdot\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^2}\cdot\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1}\cdot\frac{\partial \boldsymbol{a}^1}{\partial \boldsymbol{z}^1}\cdot\frac{\partial \boldsymbol{z}^1}{\partial \boldsymbol{b}^1} \tag{186}$$

$$\frac{\partial C}{\partial W^1}=\frac{\partial C}{\partial \hat{\boldsymbol{y}}}\cdot\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}^2}\cdot\frac{\partial \boldsymbol{z}^2}{\partial \boldsymbol{a}^1}\cdot\frac{\partial \boldsymbol{a}^1}{\partial \boldsymbol{z}^1}\cdot\frac{\partial \boldsymbol{z}^1}{\partial W^1} \tag{187}$$

梯度下降算法调参数　当求出网络参数的微分之后，就可以使用各种优化算法来调整网络的参数了，例如采用梯度下降算法，我们假设学习率为 $\alpha>0$：

$$W^2=W^2-\alpha\cdot\frac{\partial C}{\partial W^2} \tag{188}$$

$$\boldsymbol{b}^2=\boldsymbol{b}^2-\alpha\cdot\frac{\partial C}{\partial \boldsymbol{b}^2} \tag{189}$$

$$W^1=W^1-\alpha\cdot\frac{\partial C}{\partial W^1} \tag{190}$$

$$\boldsymbol{b}^1=\boldsymbol{b}^1-\alpha\cdot\frac{\partial C}{\partial \boldsymbol{b}^1} \tag{191}$$

在这里我们只讨论了一个简化再简化的版本，忽略很多细节，例如我们并没有考虑迷你批次问题，而使用的是在线学习的算法，即每次只学习一个样本，而实际工程中，我们会拿例如 128 个样本一次性进行学习，还有就为了增加网络的泛化能力，我们一般会增加 L_2 调整项，就是代价函数在 Cross Entropy 函数基础上，再加上一个调整项：

$$L = C + \omega = C + \lambda \cdot \sum_{l=1}^{L} \|W^l\|_2 \tag{192}$$

5.1.6 Softmax 层实现技术

在前两节中，我们详细讲解反向传播的链式求导和计算图法，这也是 TensorFlow、Pytorch 的底层实现技术。但是如果我们利用 Numpy 手工实现的话，由于我们的网络结构完全确定，完全复现这些计算过程，显然会非常浪费计算资源，在实际中我们会只利用最后的计算结果，而忽略其中的计算过程。

在线学习 我们在实际应用中，网络的输出层通常采用 Softmax 激活函数，而代价函数通常采用 CrossEntropy，因此通常的处理方法为将输出层和代价函数层合而为一，一次性计算出网络输出 $\hat{\boldsymbol{y}}^{(i)}$、代价函数值 $\mathcal{J}$、代价函数对连接权值的微分 $\frac{\partial \mathcal{J}}{\partial W}$、代价函数对本层偏置值的微分 $\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}}$、代价函数对上一层输出的微分 $\frac{\partial \mathcal{J}}{\partial \boldsymbol{x}}$。下面我们分别来看这些值的求法。
在线学习就是一次学习一个样本，在下面的公式推导中，如本层的输入线性和 $\boldsymbol{z}^{l,(i)}$ 代表第 l 层第 i 个样本的线性输入信号，在下面为了公式推层的简洁性，我们将省略上标。同时我们将上一层的输出定义为 $\boldsymbol{x} \in R^n$。我们假设共有 m 个样本，样本特征数为 n，共有 K 个类别。
我们的代价函数采用交叉熵（CrossEntropy）形式，可以表求为如下形式：

$$\mathcal{J} = -\boldsymbol{y} \log \hat{\boldsymbol{y}} \rightarrow np.sum(\begin{bmatrix} 0 \\ 0 \\ \cdots \\ -\log \hat{y}_r \\ \cdots \\ 0 \end{bmatrix}) = -\log \hat{y}_r \tag{193}$$

式中的相乘是 elementwise 相乘，最后通过 numpy.sum 进行求和，得到一个标量值。其中 r 为是第 r 类。
接下来我们将代价函数对 $\hat{\boldsymbol{y}}$ 求微分，根据向量微分章节中标量对向量微分可得：

$$\frac{\partial \mathcal{J}}{\partial \hat{\boldsymbol{y}}} = \begin{bmatrix} 0 & \cdots & -\frac{1}{\hat{y}_r} & \cdots & 0 \end{bmatrix} \in R^{1 \times K} \tag{194}$$

接下来我们求本层输出对本层线性输入的微分，根据我们在前面推导的 softmax 函数微分公式：

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}} = \begin{bmatrix} \hat{y}_1(1-\hat{y_1}) & -\hat{y_1}\hat{y_2} & \cdots & -\hat{y_1}\hat{y_K} \\ -\hat{y_2}\hat{y_1} & \hat{y}_2(1-\hat{y_2}) & \cdots & -\hat{y_2}\hat{y_K} \\ \cdots & \cdots & \cdots & \cdots \\ \hat{y}_K(1-\hat{y_2}) & \hat{y}_K(1-\hat{y_2}) & \cdots & \hat{y}_K(1-\hat{y_K}) \end{bmatrix} \in R^{K \times K} \tag{195}$$

接下来我们求 $\boldsymbol{z}$ 对 W 的偏微分：

$$\frac{\partial \boldsymbol{z}}{\partial W}=\begin{bmatrix}\frac{\partial z_1}{\partial W}\\ \frac{\partial z_2}{\partial W}\\ \dots\\ \frac{\partial z_K}{\partial W}\end{bmatrix}=\begin{bmatrix}\begin{bmatrix}x_1 & x_2 & \dots & x_n\\ 0 & 0 & \dots & 0\\ \dots & \dots & \dots & \dots\\ 0 & 0 & \dots & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0 & \dots & 0\\ x_1 & x_2 & \dots & x_n\\ \dots & \dots & \dots & \dots\\ 0 & 0 & \dots & 0\end{bmatrix}\\ \dots\\ \begin{bmatrix}0 & 0 & \dots & 0\\ 0 & 0 & \dots & 0\\ \dots & \dots & \dots & \dots\\ x_1 & x_2 & \dots & x_n\end{bmatrix}\end{bmatrix}\in R^{K\times(K\times n)} \tag{196}$$

接下来我们求 $\boldsymbol{z}$ 对 $\boldsymbol{b}$ 的偏微分：

$$\frac{\partial z}{\partial \boldsymbol{b}}=\begin{bmatrix}\frac{\partial z_1}{\partial b_1} & \frac{\partial z_1}{\partial b_2} & \dots & \frac{\partial z_1}{\partial b_K}\\ \frac{\partial z_2}{\partial b_1} & \frac{\partial z_2}{\partial b_2} & \dots & \frac{\partial z_2}{\partial b_K}\\ \dots & \dots & \dots & \dots\\ \frac{\partial z_K}{\partial b_1} & \frac{\partial z_K}{\partial b_2} & \dots & \frac{\partial z_K}{\partial b_K}\end{bmatrix}=\begin{bmatrix}1 & 0 & \dots & 0\\ 0 & 1 & \dots & 0\\ \dots & \dots & \dots & \dots\\ 0 & 0 & \dots & 1\end{bmatrix}\in R^{K\times K} \tag{197}$$

接下来我们求 $\boldsymbol{z}$ 对上一层输出 $\boldsymbol{x}$ 的微分：

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}=\begin{bmatrix}\frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} & \dots & \frac{\partial z_1}{\partial x_n}\\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} & \dots & \frac{\partial z_2}{\partial x_n}\\ \dots & \dots & \dots & \dots\\ \frac{\partial z_K}{\partial x_1} & \frac{\partial z_K}{\partial x_2} & \dots & \frac{\partial z_K}{\partial x_n}\end{bmatrix}=\begin{bmatrix}W_{1,1} & W_{1,2} & \dots & W_{1,n}\\ W_{2,1} & W_{2,2} & \dots & W_{2,n}\\ \dots & \dots & \dots & \dots\\ W_{K,1} & W_{K,2} & \dots & W_{K,n}\end{bmatrix}=W\in R^{K\times n} \tag{198}$$

根据链式求导法则，我们可以求出代价函数对本层线性输入的微分：

$$\begin{aligned}&\frac{\partial J}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}}\\ &=\begin{bmatrix}0 & \dots & -\frac{1}{\hat{y}_r} & \dots & 0\end{bmatrix}\begin{bmatrix}\hat{y}_1(1-\hat{y}_1) & -\hat{y}_1\hat{y}_2 & \dots & -\hat{y}_1\hat{y}_r & \dots & -\hat{y}_1\hat{y}_K\\ -\hat{y}_2\hat{y}_1 & \hat{y}_2(1-\hat{y}_2) & \dots & -\hat{y}_1\hat{y}_r & \dots & -\hat{y}_2\hat{y}_K\\ \dots & \dots & \dots & \dots & \dots & \dots\\ -\hat{y}_r\hat{y}_1 & -\hat{y}_r\hat{y}_2 & \dots & \hat{y}_r(1-\hat{y}_r) & \dots & -\hat{y}_r\hat{y}_K\\ \dots & \dots & \dots & \dots & & \\ \hat{y}_K(1-\hat{y}_2) & \hat{y}_K(1-\hat{y}_2) & \dots & -\hat{y}_K\hat{y}_r & \dots & \hat{y}_K(1-\hat{y}_K)\end{bmatrix}\\ &=\begin{bmatrix}\hat{y}_1 & \hat{y}_2 & \dots & \hat{y}_r-1 & \dots & \hat{y}_K\end{bmatrix}\\ &=\hat{\boldsymbol{y}}-\boldsymbol{y}\in R^K\end{aligned} \tag{199}$$

通过链式求导法则求代价函数对连接权值的微分：

$$
\begin{aligned}
\frac{\partial J}{\partial W} &= \frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial W} \\
&= \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \dots & -(1-\hat{y}_r) & \dots & \hat{y}_K \end{bmatrix}
\begin{bmatrix}
\begin{bmatrix} x_1 & x_2 & \dots & x_n \\ 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix} \\
\begin{bmatrix} 0 & 0 & \dots & 0 \\ x_1 & x_2 & \dots & x_n \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix} \\
\dots \\
\begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ x_1 & x_2 & \dots & x_n \end{bmatrix}
\end{bmatrix} \\
&= \begin{bmatrix}
\hat{y}_1 x_1 & \hat{y}_1 x_2 & \dots & \hat{y}_1 x_r & \dots & \hat{y}_1 x_n \\
\hat{y}_2 x_1 & \hat{y}_2 x_2 & \dots & \hat{y}_2 x_r & \dots & \hat{y}_2 x_n \\
\dots & \dots & \dots & \dots & \dots & \dots \\
(\hat{y}_r - 1) x_1 & (\hat{y}_r - 1) x_2 & \dots & (\hat{y}_r - 1) x_r & \dots & (\hat{y}_r - 1) x_r \\
\dots & \dots & \dots & \dots & \dots & \dots \\
\hat{y}_K x_1 & \hat{y}_K x_2 & \dots & \hat{y}_K x_r & \dots & \hat{y}_K x_n
\end{bmatrix} \\
&= \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \dots \\ \hat{y}_r - 1 \\ \dots \\ \hat{y}_K \end{bmatrix}
\begin{bmatrix} x_1 & x_2 & \dots & x_r & \dots & x_n \end{bmatrix} \\
&= (\hat{\boldsymbol{y}} - \boldsymbol{y}) \cdot \boldsymbol{x}^T
\end{aligned}
\tag{200}
$$

接下来我们根据链式求导法则求代价函数对本层偏置值的微分：

$$
\begin{aligned}
\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}} &= \frac{\partial \mathcal{J}}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} \\
&= \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \dots & -(1-\hat{y}_r) & \dots & \hat{y}_K \end{bmatrix}
\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix} \\
&= \hat{\boldsymbol{y}} - \boldsymbol{y}
\end{aligned}
\tag{201}
$$

最后我们根据链式求导法则求代价函数对上一层输出的微分：

$$\begin{aligned}\frac{\partial \mathcal{J}}{\partial \boldsymbol{x}} &= \frac{\partial \mathcal{J}}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \\ &= \begin{bmatrix}\hat{y}_1 & \hat{y}_2 & ... & -(1-\hat{y}_r) & ... & \hat{y}_K\end{bmatrix}\begin{bmatrix}W_{1,1} & W_{1,2} & ... & W1,n \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n}\end{bmatrix} \\ &= \begin{bmatrix}\hat{y}_1 & \hat{y}_2 & ... & -(1-\hat{y}_r) & ... & \hat{y}_K\end{bmatrix}^T \cdot W \in R^n\end{aligned} \tag{202}$$

批量学习 由于在线学习中，样本具有随机性，会产生很多无效的参数调整，因此在实际应用中，通常将数据集划分为较小的迷你批次，这样既可以利用样本的随机性，从而避免落入局部极小值点，又可以减少因为样本随机性所造成的无效参数调整。在本节中，我们将讨论在迷你批次下，Softmax 层学习方法。

我们假设迷你批次大小为 M，每个样本仍然为 $\boldsymbol{x}^{(i)} \in R^N$，即有 N 个特征，在批量学习中，我们一次性输入一个迷你批次：

$$X = \begin{bmatrix}(\boldsymbol{x}^{(1)})^T \\ (\boldsymbol{x}^{(2)})^T \\ ... \\ (\boldsymbol{x}^{(i)})^T \\ ... \\ (\boldsymbol{x}^{(M)})^T\end{bmatrix} \in R^{M\times N} \tag{203}$$

其产生线性输入为：

$$Z = X \cdot W^T + \boldsymbol{b}, \quad Z \in R^{M\times K}, X \in R^{M\times N}, W \in R^{K\times N}, \boldsymbol{b} \in R^K \tag{204}$$

本层的输出为：

$$\hat{Y} = softmax(Z) = \begin{bmatrix}\begin{bmatrix}\frac{z_1^{(1)}}{\sum_{k=1}^K z_k^{(1)}} & \frac{z_2^{(1)}}{\sum_{k=1}^K z_k^{(1)}} & ... & \frac{z_K^{(1)}}{\sum_{k=1}^K z_k^{(1)}}\end{bmatrix} \\ \begin{bmatrix}\frac{z_1^{(2)}}{\sum_{k=1}^K z_k^{(2)}} & \frac{z_2^{(2)}}{\sum_{k=1}^K z_k^{(2)}} & ... & \frac{z_K^{(2)}}{\sum_{k=1}^K z_k^{(2)}}\end{bmatrix} \\ ... \\ \begin{bmatrix}\frac{z_1^{(M)}}{\sum_{k=1}^K z_k^{(M)}} & \frac{z_2^{(M)}}{\sum_{k=1}^K z_k^{(M)}} & ... & \frac{z_K^{(M)}}{\sum_{k=1}^K z_k^{(M)}}\end{bmatrix}\end{bmatrix} \in R^{M\times K} \tag{205}$$

我们采用交叉熵（CrossEntropy）作为代价函数：

$$\mathcal{J} = np.sum(-Y\log\hat{Y}) = np.sum(\begin{bmatrix}\begin{bmatrix}0 & 0 & ... & -\log\hat{Y}_{r_1}^{(1)} & ... & 0\end{bmatrix} \\ \begin{bmatrix}0 & 0 & ... & -\log\hat{Y}_{r_2}^{(2)} & ... & 0\end{bmatrix} \\ ... \\ \begin{bmatrix}0 & 0 & ... & -\log\hat{Y}_{r_M}^{(M)} & ... & 0\end{bmatrix}\end{bmatrix}) \in R \tag{206}$$

接下来我们求代价函数对本层输出的微分：

$$\frac{\partial \mathcal{J}}{\partial \hat{Y}} = \begin{bmatrix} \begin{bmatrix} 0 & 0 & ... & -\frac{1}{\hat{Y}_{1,r_1}} & ... & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & ... & -\frac{1}{\hat{Y}_{2,r_2}} & ... & 0 \end{bmatrix} \\ ... \\ \begin{bmatrix} 0 & 0 & ... & -\frac{1}{\hat{Y}_{M,r_M}} & ... & 0 \end{bmatrix} \end{bmatrix} \in R^{M \times K} \tag{207}$$

接下来我们求本层输出对本层线性输入的微分：

$$\frac{\partial \hat{Y}}{\partial Z} = \begin{bmatrix} \begin{bmatrix} \hat{y}_1^{(1)}(1-\hat{y}_1^{(1)}) & -\hat{y}_1^{(1)}\hat{y}_2^{(1)} & ... & -\hat{y}_1^{(1)}\hat{y}_K^{(1)} \\ -\hat{y}_2^{(1)}\hat{y}_1^{(1)} & \hat{y}_2^{(1)}(1-\hat{y}_2^{(1)}) & ... & -\hat{y}_2^{(1)}\hat{y}_K^{(1)} \\ ... & ... & ... & ... \\ \hat{y}_K^{(1)}(1-\hat{y}_2^{(1)}) & \hat{y}_K^{(1)}(1-\hat{y}_2^{(1)}) & ... & \hat{y}_K^{(1)}(1-\hat{y}_K^{(1)}) \end{bmatrix} \\ \begin{bmatrix} \hat{y}_1^{(2)}(1-\hat{y}_1^{(2)}) & -\hat{y}_1^{(2)}\hat{y}_2^{(2)} & ... & -\hat{y}_1^{(2)}\hat{y}_K^{(2)} \\ -\hat{y}_2^{(2)}\hat{y}_1^{(2)} & \hat{y}_2^{(2)}(1-\hat{y}_2^{(2)}) & ... & -\hat{y}_2^{(2)}\hat{y}_K^{(2)} \\ ... & ... & ... & ... \\ \hat{y}_K^{(2)}(1-\hat{y}_2^{(2)}) & \hat{y}_K^{(2)}(1-\hat{y}_2^{(2)}) & ... & \hat{y}_K^{(2)}(1-\hat{y}_K^{(2)}) \end{bmatrix} \\ ... \\ \begin{bmatrix} \hat{y}_1^{(M)}(1-\hat{y}_1^{(M)}) & -\hat{y}_1^{(M)}\hat{y}_2^{(M)} & ... & -\hat{y}_1^{(M)}\hat{y}_K^{(M)} \\ -\hat{y}_2^{(M)}\hat{y}_1^{(M)} & \hat{y}_2^{(M)}(1-\hat{y}_2^{(M)}) & ... & -\hat{y}_2^{(M)}\hat{y}_K^{(M)} \\ ... & ... & ... & ... \\ \hat{y}_K^{(M)}(1-\hat{y}_2^{(M)}) & \hat{y}_K^{(M)}(1-\hat{y}_2^{(M)}) & ... & \hat{y}_K^{(M)}(1-\hat{y}_K^{(M)}) \end{bmatrix} \end{bmatrix} \in R^{M \times (K \times K)} \tag{208}$$

接下来求本层线性输入对连接权值的微分：

$$\frac{\partial Z}{\partial W}=\begin{bmatrix}\begin{bmatrix}\frac{\partial z_1^{(1)}}{\partial W}\\ \frac{\partial z_2^{(1)}}{\partial W}\\ \cdots\\ \frac{\partial z_K^{(1)}}{\partial W}\end{bmatrix}\\ \begin{bmatrix}\frac{\partial z_1^{(2)}}{\partial W}\\ \frac{\partial z_2^{(2)}}{\partial W}\\ \cdots\\ \frac{\partial z_K^{(2)}}{\partial W}\end{bmatrix}\\ \cdots\\ \begin{bmatrix}\frac{\partial z_1^{(M)}}{\partial W}\\ \frac{\partial z_2^{(M)}}{\partial W}\\ \cdots\\ \frac{\partial z_K^{(M)}}{\partial W}\end{bmatrix}\end{bmatrix}=\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)}\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)}\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \cdots\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)}\end{bmatrix}\end{bmatrix}\\ \begin{bmatrix}\begin{bmatrix}x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)}\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)}\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \cdots\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)}\end{bmatrix}\end{bmatrix}\\ \cdots\\ \begin{bmatrix}\begin{bmatrix}x_1^{(M)} & x_2^{(M)} & \cdots & x_n^{(M)}\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ x_1^{(M)} & x_2^{(M)} & \cdots & x_n^{(M)}\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \cdots\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ x_1^{(M)} & x_2^{(M)} & \cdots & x_n^{(M)}\end{bmatrix}\end{bmatrix}\end{bmatrix}\in R^{M\times K\times(K\times N)} \tag{209}$$

接下来我们求本层输入对偏置值的微分：

$$\frac{\partial Z}{\partial \boldsymbol{b}} = \begin{bmatrix} \begin{bmatrix} \frac{\partial z_1^{(1)}}{\partial b_1} & \frac{\partial z_1^{(1)}}{\partial b_2} & \cdots & \frac{\partial z_1^{(1)}}{\partial b_K} \\ \frac{\partial z_2^{(1)}}{\partial b_1} & \frac{\partial z_2^{(1)}}{\partial b_2} & \cdots & \frac{\partial z_2^{(1)}}{\partial b_K} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(1)}}{\partial b_1} & \frac{\partial z_K^{(1)}}{\partial b_2} & \cdots & \frac{\partial z_K^{(1)}}{\partial b_K} \end{bmatrix} \\ \begin{bmatrix} \frac{\partial z_1^{(2)}}{\partial b_1} & \frac{\partial z_1^{(2)}}{\partial b_2} & \cdots & \frac{\partial z_1^{(2)}}{\partial b_K} \\ \frac{\partial z_2^{(2)}}{\partial b_1} & \frac{\partial z_2^{(2)}}{\partial b_2} & \cdots & \frac{\partial z_2^{(2)}}{\partial b_K} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(2)}}{\partial b_1} & \frac{\partial z_K^{(2)}}{\partial b_2} & \cdots & \frac{\partial z_K^{(2)}}{\partial b_K} \end{bmatrix} \\ \cdots \\ \begin{bmatrix} \frac{\partial z_1^{(M)}}{\partial b_1} & \frac{\partial z_1^{(M)}}{\partial b_2} & \cdots & \frac{\partial z_1^{(M)}}{\partial b_K} \\ \frac{\partial z_2^{(M)}}{\partial b_1} & \frac{\partial z_2^{(M)}}{\partial b_2} & \cdots & \frac{\partial z_2^{(M)}}{\partial b_K} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(M)}}{\partial b_1} & \frac{\partial z_K^{(M)}}{\partial b_2} & \cdots & \frac{\partial z_K^{(M)}}{\partial b_K} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix} \\ ... \\ \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix} \end{bmatrix} \in R^{M \times (K \times K)} \tag{210}$$

接下来我们求本层输入对上一层输出的微分：

$$\frac{\partial Z}{\partial X} = \begin{bmatrix} \begin{bmatrix} \frac{\partial z_1^{(1)}}{\partial x_1} & \frac{\partial z_1^{(1)}}{\partial x_2} & \cdots & \frac{\partial z_1^{(1)}}{\partial x_n} \\ \frac{\partial z_2^{(1)}}{\partial x_1} & \frac{\partial z_2^{(1)}}{\partial x_2} & \cdots & \frac{\partial z_2^{(1)}}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(1)}}{\partial x_1} & \frac{\partial z_K^{(1)}}{\partial x_2} & \cdots & \frac{\partial z_K^{(1)}}{\partial x_n} \end{bmatrix} \\ \begin{bmatrix} \frac{\partial z_1^{(2)}}{\partial x_1} & \frac{\partial z_1^{(2)}}{\partial x_2} & \cdots & \frac{\partial z_1^{(2)}}{\partial x_n} \\ \frac{\partial z_2^{(2)}}{\partial x_1} & \frac{\partial z_2^{(2)}}{\partial x_2} & \cdots & \frac{\partial z_2^{(2)}}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(2)}}{\partial x_1} & \frac{\partial z_K^{(2)}}{\partial x_2} & \cdots & \frac{\partial z_K^{(2)}}{\partial x_n} \end{bmatrix} \\ \cdots \\ \begin{bmatrix} \frac{\partial z_1^{(M)}}{\partial x_1} & \frac{\partial z_1^{(M)}}{\partial x_2} & \cdots & \frac{\partial z_1^{(M)}}{\partial x_n} \\ \frac{\partial z_2^{(M)}}{\partial x_1} & \frac{\partial z_2^{(M)}}{\partial x_2} & \cdots & \frac{\partial z_2^{(M)}}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(M)}}{\partial x_1} & \frac{\partial z_K^{(M)}}{\partial x_2} & \cdots & \frac{\partial z_K^{(M)}}{\partial x_n} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W_{1,n} \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n} \end{bmatrix} \\ \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W_{1,n} \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n} \end{bmatrix} \\ ... \\ \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W_{1,n} \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} W \\ W \\ ... \\ W \end{bmatrix} \in R^{M \times (K \times N)} \tag{211}$$

接下来我们求代价函数对本层线性输入的微分：

$$\frac{\partial \mathcal{J}}{\partial Z} = \frac{\partial \mathcal{J}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial Z}$$

$$= \begin{bmatrix} \begin{bmatrix} \hat{y}_1^{(1)} & \hat{y}_2^{(1)} & ... & \hat{y}_r^{(1)} - 1 & ... & \hat{y}_K^{(1)} \end{bmatrix} \\ \begin{bmatrix} \hat{y}_1^{(2)} & \hat{y}_2^{(2)} & ... & \hat{y}_r^{(2)} - 1 & ... & \hat{y}_K^{(2)} \end{bmatrix} \\ ... \\ \begin{bmatrix} \hat{y}_1^{(M)} & \hat{y}_2^{(M)} & ... & \hat{y}_r^{(M)} - 1 & ... & \hat{y}_K^{(M)} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{y}}^{(1)} - \boldsymbol{y}^{(1)} \\ \hat{\boldsymbol{y}}^{(2)} - \boldsymbol{y}^{(2)} \\ ... \\ \hat{\boldsymbol{y}}^{(M)} - \boldsymbol{y}^{(M)} \end{bmatrix} \in R^{M \times K} \tag{212}$$

接下来求代价函数对连接权值的微分：

$$\frac{\partial \mathcal{J}}{\partial W} = \frac{\partial \mathcal{J}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial Z} \frac{\partial Z}{\partial W}$$

$$= np.sum(\begin{bmatrix} (\hat{\boldsymbol{y}}^{(1)} - \boldsymbol{y}^{(1)}) \cdot (\boldsymbol{x}^{(1)})^T \\ (\hat{\boldsymbol{y}}^{(2)} - \boldsymbol{y}^{(2)}) \cdot (\boldsymbol{x}^{(2)})^T \\ ... \\ (\hat{\boldsymbol{y}}^{(M)} - \boldsymbol{y}^{(M)}) \cdot (\boldsymbol{x}^{(M)})^T \end{bmatrix}, axis = 0) \in R^{K \times N} \tag{213}$$

接下来我们求代价函数对偏置值的微分：

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}} = \frac{\partial \mathcal{J}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial Z} \frac{\partial Z}{\partial \boldsymbol{b}}$$

$$= np.sum(\begin{bmatrix} \hat{\boldsymbol{y}}^{(1)} - \boldsymbol{y}^{(1)} \\ \hat{\boldsymbol{y}}^{(2)} - \boldsymbol{y}^{(2)} \\ ... \\ \hat{\boldsymbol{y}}^{(M)} - \boldsymbol{y}^{(M)} \end{bmatrix}, axis = 0) \in R^{K} \tag{214}$$

最后我们求代价函数对上一层输出的微分：

$$\frac{\partial \mathcal{J}}{\partial X} = \frac{\partial \mathcal{J}}{\partial \hat{Y}} \frac{\partial \hat{Y}}{\partial Z} \frac{\partial Z}{\partial X}$$

$$= \begin{bmatrix} \begin{bmatrix} \hat{y}_1^{(1)} & \hat{y}_2^{(1)} & ... & -(1 - \hat{y}_r^{(1)}) & ... & \hat{y}_K^{(1)} \end{bmatrix} \cdot W \\ \begin{bmatrix} \hat{y}_1^{(2)} & \hat{y}_2^{(2)} & ... & -(1 - \hat{y}_r^{(2)}) & ... & \hat{y}_K^{(2)} \end{bmatrix} \cdot W \\ ... \\ \begin{bmatrix} \hat{y}_1^{(M)} & \hat{y}_2^{(M)} & ... & -(1 - \hat{y}_r^{(M)}) & ... & \hat{y}_K^{(M)} \end{bmatrix} \cdot W \end{bmatrix} = \begin{bmatrix} (\hat{\boldsymbol{y}}^{(1)} - \boldsymbol{y}^{(1)})^T \cdot W \\ (\hat{\boldsymbol{y}}^{(2)} - \boldsymbol{y}^{(2)})^T \cdot W \\ ... \\ (\hat{\boldsymbol{y}}^{(M)} - \boldsymbol{y}^{(M)})^T \cdot W \end{bmatrix} \in R^{M \times N} \tag{215}$$

使用本节公式有一个重要前提条件，就是 Softmax 层后面必须接交叉熵代价函数，如果不是交叉熵代价函数，上述公式就不成立，就必须按照常规方法经过张量运算求出。

5.1.7 ReLU 层实现技术

ReLU 层及其变种 SeLU, ELU、LeakyReLU，是当前神经网络隐藏层神经元激活函数的首选。在本部分，我们将讨论 LeakyReLU 层前向传播和反向传播的问题。

在线学习 我们假设输入信号为 $\boldsymbol{x} \in R^N$，共有 m 个样本。本层有 K 个神经元节点，连接权值为 $W \in R^{K \times N}$，偏置值为 $\boldsymbol{b} \in R^K$，我们先来看线性输入：

$$\boldsymbol{z} = W \cdot \boldsymbol{x}^T + \boldsymbol{b}, \quad \boldsymbol{z} \in R^K \tag{216}$$

经过 LeakyReLU 激活函数产生输出：

$$\begin{aligned} \boldsymbol{a} = f(\boldsymbol{z}) &= \begin{bmatrix} f(z_1) \\ f(z_2) \\ ... \\ f(z_K) \end{bmatrix} \\ f(z_i) &= \begin{cases} z_i, & z_i \geq 0 \\ \epsilon z_i, & z_i < 0 \end{cases} \end{aligned} \tag{217}$$

以上即为正向传播过程，下面我们来看反向传播过程。根据上一节的讨论，上一层传导过来的相对于代价函数的微分为：

$$\frac{\partial j}{\partial \hat{\boldsymbol{y}}} = \begin{bmatrix} g_1 & g_2 & ... & g_N \end{bmatrix} \in R^N \tag{218}$$

接下来我们求本层输出对本层线性输入的微分：

$$\begin{aligned} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} &= \begin{bmatrix} \frac{\partial a_1}{\partial z_1} & \frac{\partial a_1}{\partial z_2} & ... & \frac{\partial a_1}{\partial z_K} \\ \frac{\partial a_2}{\partial z_1} & \frac{\partial a_2}{\partial z_2} & ... & \frac{\partial a_2}{\partial z_K} \\ ... & & & \\ \frac{\partial a_K}{\partial z_1} & \frac{\partial a_K}{\partial z_2} & ... & \frac{\partial a_K}{\partial z_K} \end{bmatrix} \\ &= \begin{bmatrix} v_1 & 0 & ... & 0 \\ 0 & v_2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & v_K \end{bmatrix} \in R^{K \times K} \\ v_i &= \begin{cases} 1, & z_i \geq 0 \\ \epsilon, & z_i < 0 \end{cases} \end{aligned} \tag{219}$$

接下来我们求本层线性输入对上一层输出的微分：

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} & ... & \frac{\partial z_1}{\partial x_n} \\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} & ... & \frac{\partial z_2}{\partial x_n} \\ ... & ... & ... & ... \\ \frac{\partial z_K}{\partial x_1} & \frac{\partial z_K}{\partial x_2} & ... & \frac{\partial z_K}{\partial x_n} \end{bmatrix} = \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W_{1,n} \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n} \end{bmatrix} = W \in R^{K \times n} \tag{220}$$

根据链式求导法则，我们可以求出代价函数对本层线性输入的微分：

$$\begin{aligned} \frac{\partial j}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} &= \begin{bmatrix} g_1 & g_2 & ... & g_K \end{bmatrix} \begin{bmatrix} v_1 & 0 & ... & 0 \\ 0 & v_2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & v_K \end{bmatrix} \\ &= \begin{bmatrix} g_1 * v_1 & g_2 * v_2 & ... & g_K * v_K \end{bmatrix} \in R^{1 \times K} \end{aligned} \tag{221}$$

通过链式求导法则求代价函数对连接权值的微分：

$$
\begin{aligned}
\frac{\partial j}{\partial W} &= \frac{\partial j}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial W} \\
&= \begin{bmatrix} g_1 * v_1 & g_2 * v_2 & \dots & g_K * v_K \end{bmatrix}
\begin{bmatrix}
\begin{bmatrix} x_1 & x_2 & \dots & x_n \\ 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix} \\
\begin{bmatrix} 0 & 0 & \dots & 0 \\ x_1 & x_2 & \dots & x_n \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix} \\
\dots \\
\begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ x_1 & x_2 & \dots & x_n \end{bmatrix}
\end{bmatrix} \\
&= \begin{bmatrix} g_1 * v_1 * x_1 & g_1 * v_1 * x_2 & \dots & g_1 * v_1 * x_n \\ g_2 * v_2 * x_1 & g_2 * v_2 * x_2 & \dots & g_2 * v_2 * x_n \\ \dots & \dots & \dots & \dots \\ g_K * v_K * x_1 & g_K * v_K * x_2 & \dots & g_K * v_K * x_n \end{bmatrix} \\
&= \begin{bmatrix} g_1 * v_1 \\ g_2 * v_2 \\ \dots \\ g_K * v_K \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_r & \dots & x_n \end{bmatrix} \\
&= \begin{bmatrix} g_1 * v_1 \\ g_2 * v_2 \\ \dots \\ g_K * v_K \end{bmatrix} \cdot \boldsymbol{x}^T \in R^{K \times N}
\end{aligned}
\tag{222}
$$

接下来我们根据链式求导法则求代价函数对本层偏置值的微分：

$$
\begin{aligned}
\frac{\partial j}{\partial \boldsymbol{b}} &= \frac{\partial j}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}} \\
&= \begin{bmatrix} g_1 * v_1 & g_2 * v_2 & \dots & g_K * v_K \end{bmatrix}
\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix} \\
&= \begin{bmatrix} g_1 * v_1 & g_2 * v_2 & \dots & g_K * v_K \end{bmatrix} \in R^{1 \times K}
\end{aligned}
\tag{223}
$$

最后我们根据链式求导法则求代价函数对上一层输出的微分：

$$\begin{aligned}\frac{\partial j}{\partial \boldsymbol{x}} &= \frac{\partial j}{\partial \hat{\boldsymbol{y}}}\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \\ &= \begin{bmatrix} g_1 * v_1 & g_2 * v_2 & ... & g_K * v_K \end{bmatrix} \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W1,N \\ W_{2,1} & W_{2,2} & ... & W_{2,N} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,N} \end{bmatrix} \\ &= \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & ... & -(1-\hat{y}_r) & ... & \hat{y}_K \end{bmatrix} \cdot W \in R^{1\times N}\end{aligned} \tag{224}$$

这个值就可以传给下一层继续进行反向传播了。

批量学习 我们来看在批量学习方式下，怎样来处理 LeakyReLU 层。我们假设一次性输入一个迷你批次样本集：

$$X = \begin{bmatrix} \boldsymbol{x}^{(1)} \\ \boldsymbol{x}^{(2)} \\ ... \\ \boldsymbol{x}^{(M)} \end{bmatrix} \in R^{M\times N}, \boldsymbol{x}^{(i)} \in R^N \tag{225}$$

经过前向传播，求出本层的线性输入：

$$Z = X \cdot W^T + \boldsymbol{b} \in R^{M\times K} \tag{226}$$

接下来求出本层输出：

$$A = f(Z) = \begin{bmatrix} f(z_{1,1}) & f(z_{1,2}) & ... & f(z_{1,K}) \\ f(z_{2,1}) & f(z_{2,2}) & ... & f(z_{2,K}) \\ ... & ... & ... & ... \\ f(z_{M,1}) & f(z_{M,2}) & ... & f(z_{M,K}) \end{bmatrix} \in R^{M\times K} Z_{i,j} = \begin{cases} Z_{i,j}, & Z_{i,j} \geq 0 \\ \epsilon \cdot Z_{i,j}, & Z < 0 \end{cases} \tag{227}$$

以上就是前向传播过程。接下来我们看反向传播。上一层传导过来的微分值为：

$$\frac{\partial \mathcal{J}}{\partial A} = \begin{bmatrix} \begin{bmatrix} g_1^{(1)} & g_2^{(1)} & ... & g_K^{(1)} \\ g_1^{(2)} & g_2^{(2)} & ... & g_K^{(2)} \\ ... & ... & ... & ... \\ g_1^{(M)} & g_2^{(M)} & ... & g_K^{(M)} \end{bmatrix} \end{bmatrix} \tag{228}$$

接下来计算本层输出对本层线性输入的微分：

$$\frac{\partial A}{\partial Z} = \begin{bmatrix} \begin{bmatrix} v_1^{(1)} & 0 & ... & 0 \\ 0 & v_2^{(1)} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & v_K^{(1)} \end{bmatrix} \\ \begin{bmatrix} v_1^{(2)} & 0 & ... & 0 \\ 0 & v_2^{(2)} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & v_K^{(2)} \end{bmatrix} \\ ... \\ \begin{bmatrix} v_1^{(M)} & 0 & ... & 0 \\ 0 & v_2^{(M)} & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & v_K^{(M)} \end{bmatrix} \end{bmatrix} \in R^{M \times (K \times K)} \tag{229}$$

$$v_j^{(i)} = \begin{cases} 1, & z_j^{(i)} \geq 0 \\ \epsilon, & z_j^{(i)} < 0 \end{cases}$$

接下来求本层线性输入对连接权值的微分：

$$\frac{\partial Z}{\partial W}=\begin{bmatrix}\begin{bmatrix}\frac{\partial z_1^{(1)}}{\partial W}\\ \frac{\partial z_2^{(1)}}{\partial W}\\ \cdots\\ \frac{\partial z_K^{(1)}}{\partial W}\end{bmatrix}\\ \begin{bmatrix}\frac{\partial z_1^{(2)}}{\partial W}\\ \frac{\partial z_2^{(2)}}{\partial W}\\ \cdots\\ \frac{\partial z_K^{(2)}}{\partial W}\end{bmatrix}\\ \cdots\\ \begin{bmatrix}\frac{\partial z_1^{(M)}}{\partial W}\\ \frac{\partial z_2^{(M)}}{\partial W}\\ \cdots\\ \frac{\partial z_K^{(M)}}{\partial W}\end{bmatrix}\end{bmatrix}=\begin{bmatrix}\begin{bmatrix}\begin{bmatrix}x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)}\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)}\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \cdots\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)}\end{bmatrix}\end{bmatrix}\\ \begin{bmatrix}\begin{bmatrix}x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)}\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)}\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \cdots\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)}\end{bmatrix}\end{bmatrix}\\ \cdots\\ \begin{bmatrix}\begin{bmatrix}x_1^{(M)} & x_2^{(M)} & \cdots & x_n^{(M)}\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ x_1^{(M)} & x_2^{(M)} & \cdots & x_n^{(M)}\\ \cdots & \cdots & \cdots & \cdots\\ 0 & 0 & \cdots & 0\end{bmatrix}\\ \cdots\\ \begin{bmatrix}0 & 0 & \cdots & 0\\ 0 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots\\ x_1^{(M)} & x_2^{(M)} & \cdots & x_n^{(M)}\end{bmatrix}\end{bmatrix}\end{bmatrix}\in R^{M\times K\times(K\times N)} \tag{230}$$

接下来我们求本层输入对偏置值的微分：

$$\frac{\partial Z}{\partial \boldsymbol{b}} = \begin{bmatrix} \begin{bmatrix} \frac{\partial z_1^{(1)}}{\partial b_1} & \frac{\partial z_1^{(1)}}{\partial b_2} & \cdots & \frac{\partial z_1^{(1)}}{\partial b_K} \\ \frac{\partial z_2^{(1)}}{\partial b_1} & \frac{\partial z_2^{(1)}}{\partial b_2} & \cdots & \frac{\partial z_2^{(1)}}{\partial b_K} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(1)}}{\partial b_1} & \frac{\partial z_K^{(1)}}{\partial b_2} & \cdots & \frac{\partial z_K^{(1)}}{\partial b_K} \end{bmatrix} \\ \begin{bmatrix} \frac{\partial z_1^{(2)}}{\partial b_1} & \frac{\partial z_1^{(2)}}{\partial b_2} & \cdots & \frac{\partial z_1^{(2)}}{\partial b_K} \\ \frac{\partial z_2^{(2)}}{\partial b_1} & \frac{\partial z_2^{(2)}}{\partial b_2} & \cdots & \frac{\partial z_2^{(2)}}{\partial b_K} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(2)}}{\partial b_1} & \frac{\partial z_K^{(2)}}{\partial b_2} & \cdots & \frac{\partial z_K^{(2)}}{\partial b_K} \end{bmatrix} \\ \cdots \\ \begin{bmatrix} \frac{\partial z_1^{(M)}}{\partial b_1} & \frac{\partial z_1^{(M)}}{\partial b_2} & \cdots & \frac{\partial z_1^{(M)}}{\partial b_K} \\ \frac{\partial z_2^{(M)}}{\partial b_1} & \frac{\partial z_2^{(M)}}{\partial b_2} & \cdots & \frac{\partial z_2^{(M)}}{\partial b_K} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(M)}}{\partial b_1} & \frac{\partial z_K^{(M)}}{\partial b_2} & \cdots & \frac{\partial z_K^{(M)}}{\partial b_K} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix} \\ ... \\ \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix} \end{bmatrix} \in R^{M \times (K \times K)} \tag{231}$$

接下来我们求本层输入对上一层输出的微分：

$$\frac{\partial Z}{\partial X} = \begin{bmatrix} \begin{bmatrix} \frac{\partial z_1^{(1)}}{\partial x_1} & \frac{\partial z_1^{(1)}}{\partial x_2} & \cdots & \frac{\partial z_1^{(1)}}{\partial x_n} \\ \frac{\partial z_2^{(1)}}{\partial x_1} & \frac{\partial z_2^{(1)}}{\partial x_2} & \cdots & \frac{\partial z_2^{(1)}}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(1)}}{\partial x_1} & \frac{\partial z_K^{(1)}}{\partial x_2} & \cdots & \frac{\partial z_K^{(1)}}{\partial x_n} \end{bmatrix} \\ \begin{bmatrix} \frac{\partial z_1^{(2)}}{\partial x_1} & \frac{\partial z_1^{(2)}}{\partial x_2} & \cdots & \frac{\partial z_1^{(2)}}{\partial x_n} \\ \frac{\partial z_2^{(2)}}{\partial x_1} & \frac{\partial z_2^{(2)}}{\partial x_2} & \cdots & \frac{\partial z_2^{(2)}}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(2)}}{\partial x_1} & \frac{\partial z_K^{(2)}}{\partial x_2} & \cdots & \frac{\partial z_K^{(2)}}{\partial x_n} \end{bmatrix} \\ \cdots \\ \begin{bmatrix} \frac{\partial z_1^{(M)}}{\partial x_1} & \frac{\partial z_1^{(M)}}{\partial x_2} & \cdots & \frac{\partial z_1^{(M)}}{\partial x_n} \\ \frac{\partial z_2^{(M)}}{\partial x_1} & \frac{\partial z_2^{(M)}}{\partial x_2} & \cdots & \frac{\partial z_2^{(M)}}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial z_K^{(M)}}{\partial x_1} & \frac{\partial z_K^{(M)}}{\partial x_2} & \cdots & \frac{\partial z_K^{(M)}}{\partial x_n} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W_{1,n} \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n} \end{bmatrix} \\ \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W_{1,n} \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n} \end{bmatrix} \\ ... \\ \begin{bmatrix} W_{1,1} & W_{1,2} & ... & W_{1,n} \\ W_{2,1} & W_{2,2} & ... & W_{2,n} \\ ... & ... & ... & ... \\ W_{K,1} & W_{K,2} & ... & W_{K,n} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} W \\ W \\ ... \\ W \end{bmatrix} \in R^{M \times (K \times N)} \tag{232}$$

接下来我们求代价函数对本层线性输入的微分：

$$\frac{\partial \mathcal{J}}{\partial Z} = \frac{\partial \mathcal{J}}{\partial A}\frac{\partial A}{\partial Z}$$

$$= \begin{bmatrix} \begin{bmatrix} g_1^{(1)} * v_1^{(1)} & g_2^{(1)} * v_2^{(1)} & ... & g_K^{(1)} * v_K^{(1)} \end{bmatrix} \\ \begin{bmatrix} g_1^{(2)} * v_1^{(2)} & g_2^{(2)} * v_2^{(2)} & ... & g_K^{(2)} * v_K^{(2)} \end{bmatrix} \\ ... \\ \begin{bmatrix} g_1^{(M)} * v_1^{(M)} & g_2^{(M)} * v_2^{(M)} & ... & g_K^{(M)} * v_K^{(M)} \end{bmatrix} \end{bmatrix} \in R^{M \times K} \tag{233}$$

接下来求代价函数对连接权值的微分：

$$\frac{\partial \mathcal{J}}{\partial W} = \frac{\partial \mathcal{J}}{\partial A}\frac{\partial A}{\partial Z}\frac{\partial Z}{\partial W}$$

$$= np.sum(\begin{bmatrix} \begin{bmatrix} g_1^{(1)} * v_1^{(1)} \\ g_2^{(1)} * v_2^{(1)} \\ ... \\ g_K^{(1)} * v_K^{(1)} \end{bmatrix} \cdot (\boldsymbol{x}^{(1)})^T \\ \begin{bmatrix} g_1^{(2)} * v_1^{(2)} \\ g_2^{(2)} * v_2^{(2)} \\ ... \\ g_K^{(2)} * v_K^{(2)} \end{bmatrix} \cdot (\boldsymbol{x}^{(2)})^T \\ ... \\ \begin{bmatrix} g_1^{(M)} * v_1^{(M)} \\ g_2^{(M)} * v_2^{(M)} \\ ... \\ g_K^{(M)} * v_K^{(M)} \end{bmatrix} \cdot (\boldsymbol{x}^{(M)})^T \end{bmatrix}, axis = 0) \in R^{K \times N} \tag{234}$$

接下来我们求代价函数对偏置值的微分：

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}} = \frac{\partial \mathcal{J}}{\partial A}\frac{\partial A}{\partial Z}\frac{\partial Z}{\partial \boldsymbol{b}}$$

$$= np.sum(\begin{bmatrix} \begin{bmatrix} g_1^{(1)} * v_1^{(1)} & g_2^{(1)} * v_2^{(1)} & ... & g_K^{(1)} * v_K^{(1)} \end{bmatrix} \\ \begin{bmatrix} g_1^{(2)} * v_1^{(2)} & g_2^{(2)} * v_2^{(2)} & ... & g_K^{(2)} * v_K^{(2)} \end{bmatrix} \\ ... \\ \begin{bmatrix} g_1^{(M)} * v_1^{(M)} & g_2^{(M)} * v_2^{(M)} & ... & g_K^{(M)} * v_K^{(M)} \end{bmatrix} \end{bmatrix}, axis = 0) \in R^{K} \tag{235}$$

最后我们求代价函数对上一层输出的微分：

$$\frac{\partial \mathcal{J}}{\partial X} = \frac{\partial \mathcal{J}}{\partial A}\frac{\partial A}{\partial Z}\frac{\partial Z}{\partial X}$$

$$= \begin{bmatrix} \left[g_1^{(1)} * v_1^{(1)} \quad g_2^{(1)} * v_2^{(1)} \quad ... \quad g_K^{(1)} * v_K^{(1)}\right] \cdot W \\ \left[g_1^{(2)} * v_1^{(2)} \quad g_2^{(2)} * v_2^{(2)} \quad ... \quad g_K^{(2)} * v_K^{(2)}\right] \cdot W \\ ... \\ \left[g_1^{(M)} * v_1^{(M)} \quad g_2^{(M)} * v_2^{(M)} \quad ... \quad g_K^{(M)} * v_K^{(M)}\right] \cdot W \end{bmatrix} \in R^{M \times N} \tag{236}$$

5.2 Numpy 实现技术

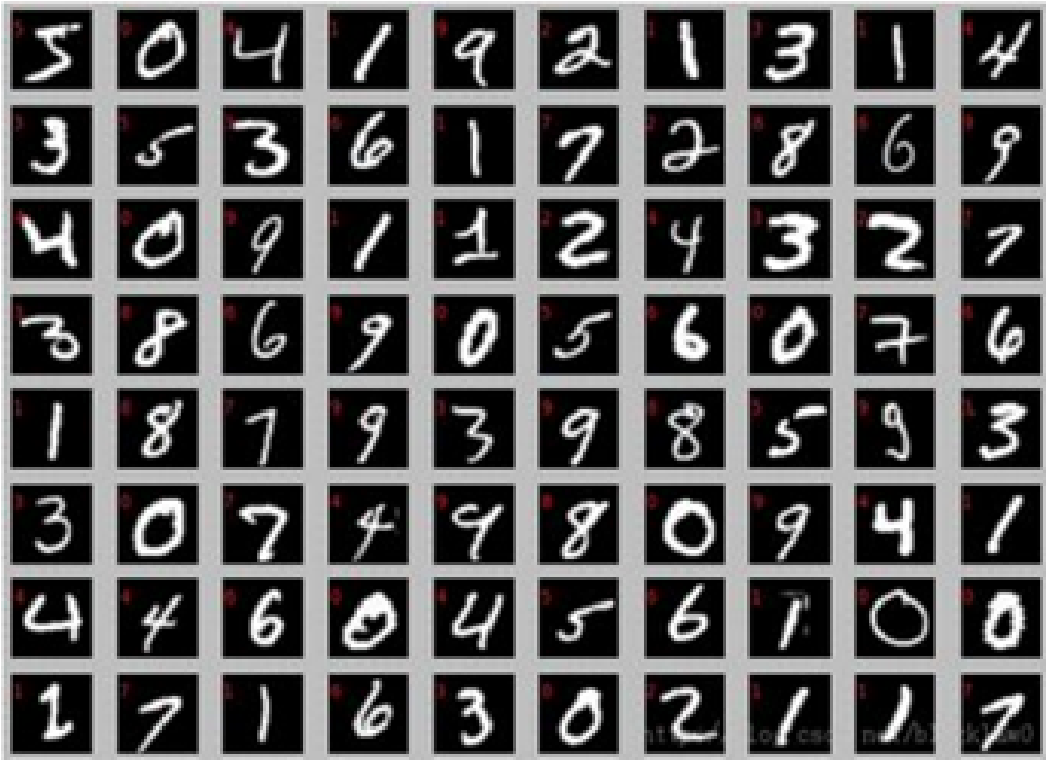
5.2.1 MNIST 数据集简介

载入 MNIST 数据集 MNIST 是 Mixed National Institute of Standards and Technology 的简称，是国际公认的大型手写数字识别数据集，是机器学习算法验证的标准数据集，其地位有点像生物学研究中的大肠杆菌，可以方便地用于各种机器学习算法间性能的比较。实际上，MNIST 每年都会组织算法竞赛，以检验各种算法的有效性。

MNIST 数据集由纽约大学 Yann LeCun 教授整理推出，每个手写数字图片的大小均为 28~28，黑底白字，并且位于图片中央，共有 60000 个训练样本集，10000 个测试样本集，其中测试样本集是不公开的。

MNIST 手写数字图片如下图所示：

Figure 62: MNIST 手写数字识别图片示例



因为 MNIST 数据集在国外网站上，通过 sklearn.datasets.ftech_openml 下载数据集会有很慢，有时会下载不下来。因此我们采取直接从网站下载别人整理好的 CSV 格式的文件，然后载入该 CSV 文件，程序如下所示：

```
from __future__ import print_function
import csv
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import sklearn.datasets as skds
#
from util.npai_ds import NpaiDs
from util.npai_stats import NpaiStats
from ann.ml.logistic_regression import LogisticRegression
from util.npai_plot import NpaiPlot

classMlpApp(object):
    def __init__(self):
        self.name = 'app.ml.Mlp'

    def test_load_mnist_ds(self):
        X, y = self.load_mnist_ds()
        idx = 101
        plt.title('{0}th sample: {1}'.format(idx, np.argmax(y[idx])))
        plt.imshow(X[idx].reshape(28, 28), cmap='gray')
        plt.show()


    def load_mnist_ds(self):
        # 文件下载链接：CSVhttps://www.openml.org/d/554
        # 从网络上获取数据集：
    X, y = skds.fetch_openml('mnist_784', version=1, return_X_y=True)
        with open('E:/alearn/dl/npai/data/mnist_784.csv', newline='',
    encoding='UTF-8') as fd:
            rows = csv.reader(fd, delimiter=',', quotechar='|')
            X0 = None
            y0 = None
            next(rows)
            cnt = 0
            rst = 0
            amount = 1000 # 每条记录保存一次1000
            X = None
            y = None
            for row in rows:
                x = np.array(row[:784], dtype=np.float)
                x /= 255.0
                y_ = np.array(row[784:])
                if None is X:
                    X = np.array([x])
                    y = np.zeros((1, 10))
```

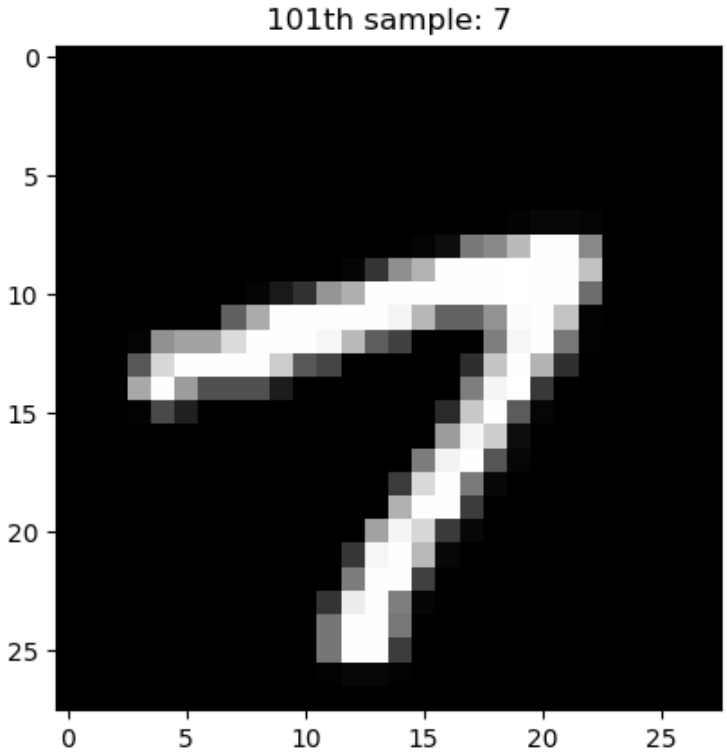
```
45                     y[cnt, int(y_[0])] = 1
46                 else:
47                     X = np.append(X, x.reshape(1, 784), axis=0)
48                     yi = np.zeros((1, 10))
49                     yi[0, int(y_[0])] = 1
50                     y = np.append(y, yi.reshape(1, 10), axis=0)
51                 if cnt % amount == 0 and cnt > 0:
52                     if None is X0:
53                         X0 = X
54                         y0 = y
55                     else:
56                         X0 = np.append(X0, X, axis=0)
57                         y0 = np.append(y0, y, axis=0)
58                     X = None
59                     y = None
60                     cnt = 0
61                     rst += amount
62                     print('处理完记录{0}'.format(rst))
63                 else:
64                     cnt += 1
65         return X0, y0
```

Listing 31: 通过预先下载的 CSV 文件载入 MNIST 数据集

代码解读如下所示：

- 第 17 行：载入手写数字识别 MNIST 数据集并可视化样本程序；
- 第 18 行：调用 load_mnist_ds 方法（在后面讲解）载入样本集和标答集；
- 第 19 行：指定要显示样本的索引号；
- 第 20 行：设定图片标题，包括图片的正确分信息，其中 $y[idx] \in R^{10}$ 为 one-hot 向量形式，为 1 的维代表正确数字，np.argmax 是求出最大元素的下标值；
- 第 21 行：将样本由 R^{784} 转为 $R^{28\times28}$，然后以黑白图像方式显示并利用 matplotlib 进行绘制，如下所示：

Figure 63: MNIST 手写数字识别单位样本示例



- 第 28 行：打开数据集文件；
- 第 29 行：创建 CSV 读取对象；
- 第 30、31 行：定义最终需要返回的样本集和标签集，初始值为空；
- 第 32 行：忽略 CSV 文件的第一行表头信息；
- 第 33 行：由于我们为了提高 CSV 文件读取和转换为数据集的效率，我们以 amount 条记录为单位，先形成这 1000 条记录的样本集 X 和标签集 y，然后将其添加到最终样本集 X0 和最终标签集 y0 的后面，cnt 代表这 1000 条记录的索引号；
- 第 34 行：共处理了多少条数据；
- 第 35 行：第 1000 条数据形成样本集 X 和标签集 y，然后添加到最终样本集 X0 和最终标签集 y0 的后面；
- 第 36、37 行：amount 条数据形成的样本集 X 和标签集 y；
- 第 38∼64 行：循环处理 CSV 文件的每一行；
- 第 39 行：由该行前 784 个元素形成样本，即一个 28×28 的黑白图片；
- 第 40 行：由于 CSV 文件中像素值的取值范围是 0∼255，除以 255 可以变为 0∼1 区间，方便将来利用 matplotlib 绘制图形；
- 第 41 行：由该行第 785 个元素形成标签，其值为 0∼9 的数字，代表该行样本对应的数字，我们在后面将会将该值变为 10 维的 one-hot 向量形式来表式；
- 第 42∼45 行：如果 amount 条记录形成的样本集 X 和标签集 y 为空，则生成只含有当前样本的样本集 X 和只含有当前标签的标签集 y，对于标签处理，我们先成一个全为零的 10 维向量，然后根据标签数值，将 one-hot 向量对应位置的值变为 1；

- 第 46~50 行：如果 amount 条记录形成的样本集 X 和标签集 y 不为空，则将当前样本添加到样本集 X 后面，先成一个全为零的 10 维向量，然后根据标签数值，将 one-hot 向量对应位置的值变为 1，然后将此标签添加到标签集 y 的后面；
- 第 51~62 行：如果正好处理了 amount 条 CSV 文件中的数据，进行如下处理：
  - 第 52~54 行：如果最终样本集为空，则最终样本集等于当前样本集，最终标签集等于当前标签集；
  - 第 55~57 行：如果最终样本集不为空，则将当前样本集添加到最终样本集后面，当前标签集添加到最终标签集后面；
- 第 58、59 行：将 amount 条数据对应的样本集 X 和标签集 y 置为空；
- 第 60 行：amount 对应的样本集索引号置为 0；
- 第 61 行：更新处理完成的总记录数；
- 第 63、64 行：如果没处理完 amount 条 CSV 文件中数据，增加 amount 样本集的索引号；
- 第 65 行：返回最终样本集 X0 和最终标签集 y0；

MNIST 数据集可视化 由于手写数字识别数据集 MNIST 为 28×28 的黑白图像，所以每个样本是 784 维，而我们只能直观的看到二维样本的情形。为了能够直观的显示 MNIST 数据集，我们使用 t-SNE 算法，将其变为二维形式进行展示。代码如下所示（app/ml/mlp_app.py）：

```
1  class MlpApp(object):
2      def __init__(self):
3          self.name = 'ann.ml.MlpApp'
4
5      def show_mnist_in_tsne(self):
6          X, y_ = self.load_mnist_ds()
7          y = np.argmax(y_, axis=1)
8          row_embedded = skmd.TSNE(n_components=2).fit_transform(X)
9          pos = pd.DataFrame(row_embedded, columns=['X', 'Y'])
10         pos['species'] = y
11         ax = pos[pos['species']==0].plot(kind='scatter', x='X', y='Y',
       color='blue', label='0')
12         pos[pos['species']==1].plot(kind='scatter', x='X', y='Y', color='
       red', label='1', ax=ax)
13         pos[pos['species']==2].plot(kind='scatter', x='X', y='Y', color='
       green', label='2', ax=ax)
14         pos[pos['species']==3].plot(kind='scatter', x='X', y='Y', color='
       yellow', label='3', ax=ax)
15         pos[pos['species']==4].plot(kind='scatter', x='X', y='Y', color='
       brown', label='4', ax=ax)
16         pos[pos['species']==5].plot(kind='scatter', x='X', y='Y', color='
       orange', label='5', ax=ax)
17         pos[pos['species']==6].plot(kind='scatter', x='X', y='Y', color='
       black', label='6', ax=ax)
18         pos[pos['species']==7].plot(kind='scatter', x='X', y='Y', color='
       pink', label='7', ax=ax)
```

```
19          pos[pos['species']==8].plot(kind='scatter', x='X', y='Y', color='
    purple', label='8', ax=ax)
20          pos[pos['species']==9].plot(kind='scatter', x='X', y='Y', color='
    cyan', label='9', ax=ax)
21          plt.show()
```

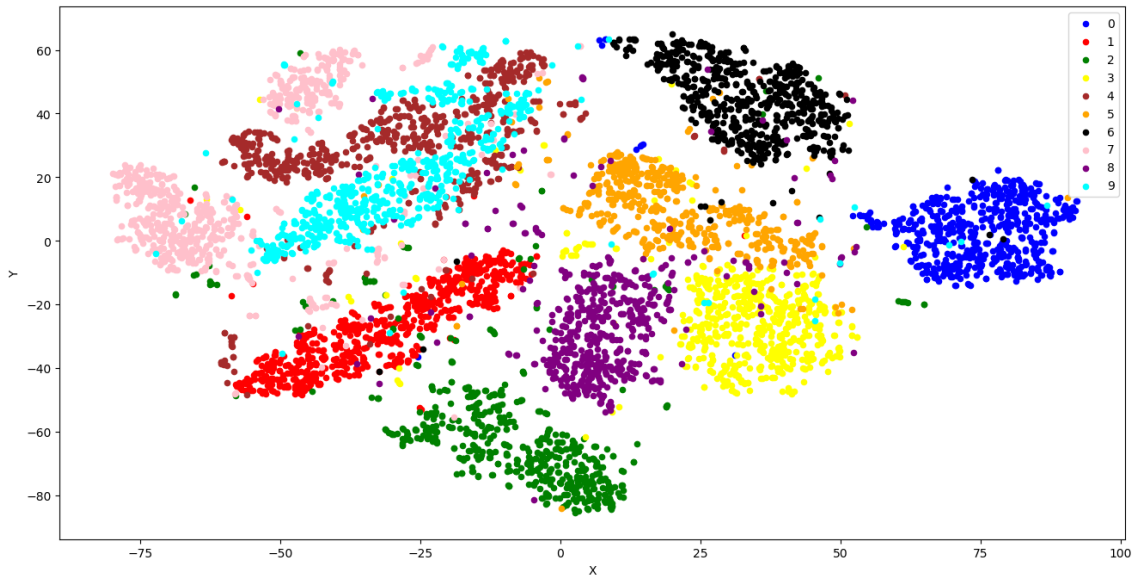
Listing 32: t-sne 可视化 MNIST 数据集

代码解读如下所示：

- 第 5 行：定义 t-sne 算法显示 MNIST 数据集方法；
- 第 6 行：读入 MNIST 数据集，$X \in R^{m \times 784}$ 为样本集，每一行为一个图片，$\boldsymbol{y} \in R^{10}$ 为 one-hot 向量，不为 0 的索引值代表对应的数据；
- 第 7 行：将原来的以 one-hot 向量形式的标签集，变为索引号形式；
- 第 8 行：调用 sklearn.manifold.TSNE 方法进行降维处理；
- 第 9 行：将其变为两列的 pandas.DataFrame；
- 第 10 行：将标签作为 DataFrame 的第三列；
- 第 11 行：将标签为 0 的记录，以蓝色散点图形式绘制到图片中；
- 第 12~19 行：将类别为 1 到 9 的样本，以不同颜色散点图形大绘制到图片中；
- 第 20 行：显示图片；

t-SNE 图片如下所示：

Figure 64: MNIST 数据集 t-SNE 图



由上图可以看出，通过选择合理的特征，我们还是可以将手写数字区分开的。

5.2.2 应用入口

使用多层感知器（MLP）模型的应用入口如下所示（app/pytorch/book/chp004/e1/mlp_mnist_app.py）：

```
#
from __future__ import print_function
import csv
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sklearn.manifold as skmd
from mlp_mnist_model import MlpMnistModel
from npai_ds import NpaiDs
from adam import Adam
from cross_entropy import CrossEntropy
#from ann.loss.negative_log_likelyhood import NegativeLogLikelyhood
from layer import Layer
import layer_common as allc
from dense import Dense
from dropout import Dropout
from activation import Activation
from fclr_layer import FclrLayer
from ces_layer import CesLayer

class MlpMnistApp(object):
    def __init__(self):
        self.name = 'ann.ml.MlpApp'

    def run(self):
        print('用于手写数字识别MLP')
        # self.show_mnist_in_tsne()
        # 载入数据集MNIST
        X, y = self.load_mnist_ds()
        n_samples, n_features = X.shape
        n_hidden = 512
        X_raw, X_test, y_raw, y_test = NpaiDs.train_test_split(X, y,
    test_size=0.2, seed=1)
        X_train, X_validate, y_train, y_validate = NpaiDs.
    train_test_split(X_raw, y_raw, test_size=0.15, seed=1)
        model = MlpMnistModel(optimizer=Adam(),
                              loss=CrossEntropy,
                              validation_data=(X_validate, y_validate))
        # 添加连接层
        model.add(FclrLayer(optimizer=Adam, K=512, N=784, epsilon=0.1))
        model.add(CesLayer(optimizer=Adam(), K=10, N=512))
        model.summary()
        #
        train_err, val_err = model.fit(X_train, y_train, n_epochs=5,
    batch_size=256)
        # Training and validation error plot
```

```
        n = len(train_err)
        training, = plt.plot(range(n), train_err, label="Training Error")
        validation, = plt.plot(range(n), val_err, label="Validation Error
    ")
        plt.legend(handles=[training, validation])
        plt.title("Error Plot")
        plt.ylabel('Error')
        plt.xlabel('Iterations')
        plt.show()

        _, accuracy = model.test_on_batch(X_test, y_test)
        print ("Accuracy:", accuracy)


    def load_mnist_ds(self):
        # 文件下载链接：CSVhttps://www.openml.org/d/554
        # 从网络上获取数据集：X, y = skds.fetch_openml('mnist_784', \
        # version=1, return_X_y=True)
        with open('data/mnist_784.csv', newline='', encoding='UTF-8') as
    fd:
            rows = csv.reader(fd, delimiter=',', quotechar='|')
            X0 = None
            y0 = None
            next(rows)
            cnt = 0
            rst = 0
            amount = 1000 # 每条记录保存一次1000
            X = None
            y = None
            for row in rows:
                x = np.array(row[:784], dtype=np.float)
                x /= 255.0
                y_ = np.array(row[784:])
                if None is X:
                    X = np.array([x])
                    y = np.zeros((1, 10))
                    y[cnt, int(y_[0])] = 1
                else:
                    X = np.append(X, x.reshape(1, 784), axis=0)
                    yi = np.zeros((1, 10))
                    yi[0, int(y_[0])] = 1
                    y = np.append(y, yi.reshape(1, 10), axis=0)
                if cnt % amount == 0 and cnt > 0:
                    if None is X0:
                        X0 = X
                        y0 = y
                    else:
                        X0 = np.append(X0, X, axis=0)
                        y0 = np.append(y0, y, axis=0)
                    X = None
```

```
92                      y = None
93                      cnt = 0
94                      rst += amount
95                      print('处理完记录{0}'.format(rst))
96                  else:
97                      cnt += 1
98          return X0, y0
99
100     def show_mnist_in_tsne(self):
101         X, y_ = self.load_mnist_ds()
102         y = np.argmax(y_, axis=1)
103         row_embedded = skmd.TSNE(n_components=2).fit_transform(X)
104         pos = pd.DataFrame(row_embedded, columns=['X', 'Y'])
105         pos['species'] = y
106         ax = pos[pos['species']==0].plot(kind='scatter', x='X', y='Y',
    color='blue', label='0')
107         pos[pos['species']==1].plot(kind='scatter', x='X', y='Y', color='
    red', label='1', ax=ax)
108         pos[pos['species']==2].plot(kind='scatter', x='X', y='Y', color='
    green', label='2', ax=ax)
109         pos[pos['species']==3].plot(kind='scatter', x='X', y='Y', color='
    yellow', label='3', ax=ax)
110         pos[pos['species']==4].plot(kind='scatter', x='X', y='Y', color='
    brown', label='4', ax=ax)
111         pos[pos['species']==5].plot(kind='scatter', x='X', y='Y', color='
    orange', label='5', ax=ax)
112         pos[pos['species']==6].plot(kind='scatter', x='X', y='Y', color='
    black', label='6', ax=ax)
113         pos[pos['species']==7].plot(kind='scatter', x='X', y='Y', color='
    pink', label='7', ax=ax)
114         pos[pos['species']==8].plot(kind='scatter', x='X', y='Y', color='
    purple', label='8', ax=ax)
115         pos[pos['species']==9].plot(kind='scatter', x='X', y='Y', color='
    cyan', label='9', ax=ax)
116         plt.show()
```

Listing 33: 多层感知器（MLP）模型应用入口

代码解析如下所示：

- 第 29 行：读入 MNIST 数据集，其中 $X \in R^{70000\times 784}$，每一行代表一个 28×28 的黑白图片，$\boldsymbol{y} \in R^{10}$ 为 one-hot 向量形式，其不为零元素的下标即为对应样本所代表的数字；
- 第 30 行：n_samples 为 70000，n_features 为 784；
- 第 31 行：定义隐藏层神经元个数为 512；
- 第 32 行：将原来的数据集进行重新排序，然后将 80% 的数据集作为训练数据集，将 20% 数据集作为测试数据集；
- 第 33 行：将 raw 数据集进一步细分为 15% 的验证数据集和 85% 的训练数据集，其中验证数据集仅用于模型参数选择和 early stopping，不参与实际训练过程；

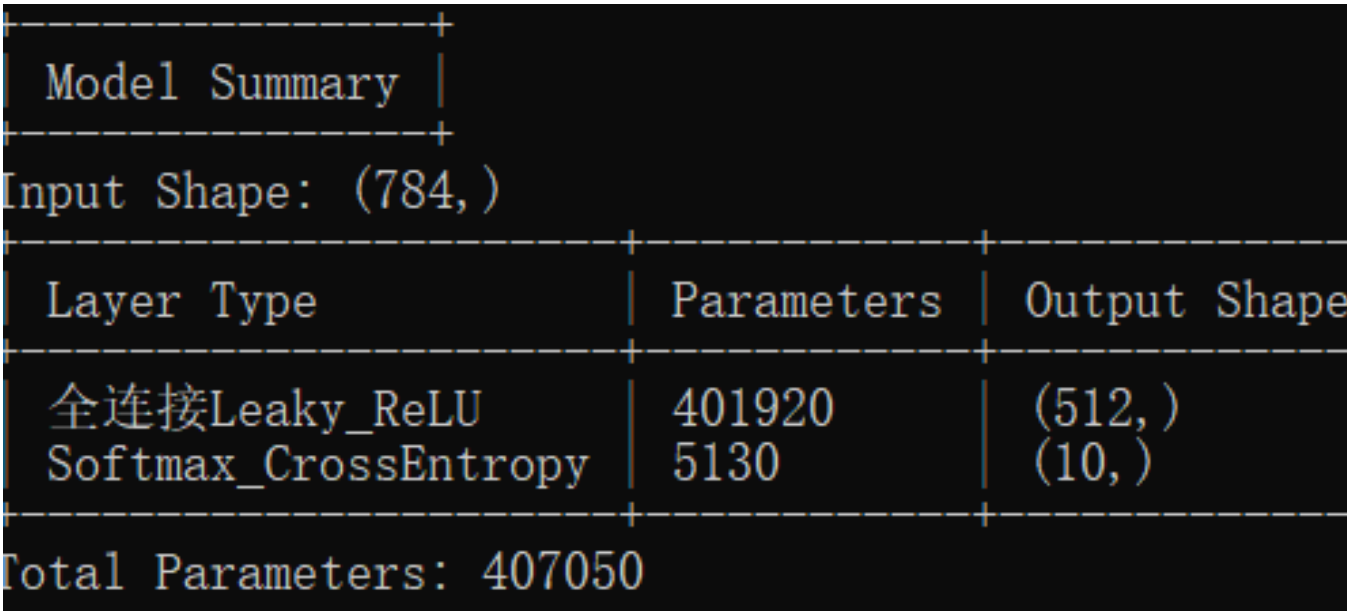
- 第 34~36 行：创建多层感知器模型类，其中优化器采用 Adam，代价函数采用交叉熵 CrossEntropy，验证样本集采用，多层感器模型类定义见多层感知器类章节；

- 第 38 行：添加第 1 个全连接层并且激活函数为 leaky_relu，神经元数量为 n_hidden=512，由于其是第一层，所以需要指定输入信号维度，其输入信号维度为 n_features=784，全连接层定义见神经网络层定义章节；

- 第 39 行：添加具有 softmax 激活函数的全连接层并且将交叉熵作为代价函数，具体介绍见激活函数章节；

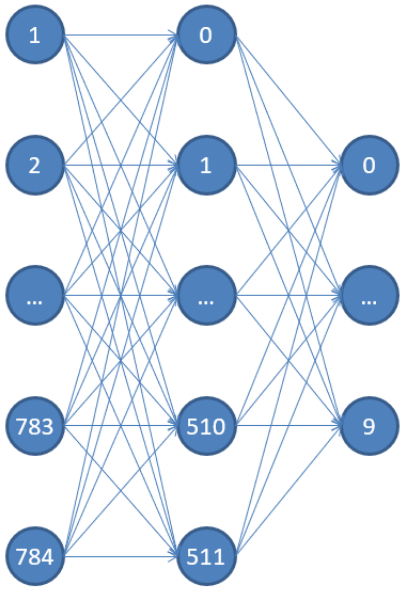
- 第 40 行：打印多层感知器模型的基本信息：

Figure 65: 多层感知器模型汇总信息



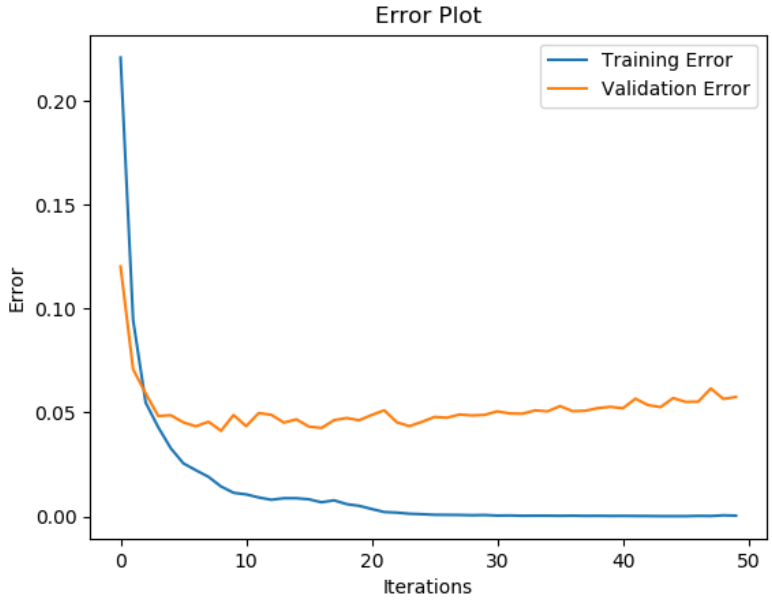
网络结构图如下所示：

Figure 66: 多层感知器模型网络结构图



- 第 42 行：训练多层感知器模型；
- 第 44~51 行：绘制在训练数据集和验证数据集上的误差曲线：

Figure 67: 在训练数据集和验证数据集上的误差曲线变化图



由图中可以看出，在开始时二者同时下降，但是训练一定次数之后，在验证数据集上的误差就会高于训练数据集上的误差，这就说明模型出现了过拟合（Overfitting），这是由于模型能力超出解题需要的能力（参数大小），这时就需要引入调整项，如 L2 或 early stopping；

- 第 53、54 行：统计并打印在测试数据集上的精度，如下所示：

Figure 68: 在测试数据集上的精度

```
Training: 100% [------------------------------------------------] Time: 0:03:18
Accuracy: 0.9358867610324729
```

5.2.3 多层感知器类

接下来我们来看多层感知器模型类的定义，如下所示（app/pytorch/book/chp004/e1/mlp_mnist_model.py）：

```
from __future__ import print_function, division
from terminaltables import AsciiTable
import numpy as np
import progressbar
#from mlfromscratch.utils import batch_iterator
#from mlfromscratch.utils.misc import bar_widgets
from npai_ds import NpaiDs
from npai_plot import bar_widgets


class MlpMnistModel(object):
    """Neural Network. Deep Learning base model.

    Parameters:
    -----------
    optimizer: class
        The weight optimizer that will be used to tune the weights in
    order of minimizing
        the loss.
    loss: class
        Loss function used to measure the model's performance. SquareLoss
     or CrossEntropy.
    validation: tuple
        A tuple containing validation data and labels (X, y)
    """
    def __init__(self, optimizer, loss, validation_data=None):
        self.optimizer = optimizer
        self.layers = []
        self.errors = {"training": [], "validation": []}
        self.loss_function = loss()
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)

        self.val_set = None
        if validation_data:
            X, y = validation_data
            self.val_set = {"X": X, "y": y}

    def set_trainable(self, trainable):
        """ Method which enables freezing of the weights of the network's
     layers. """
```

```
        for layer in self.layers:
            layer.trainable = trainable

    def add(self, layer):
        """ Method which adds a layer to the neural network """
        # If this is not the first layer added then set the input shape
        # to the output shape of the last added layer
        # If the layer has weights that needs to be initialized
        if hasattr(layer, 'initialize'):
            layer.initialize(optimizer=self.optimizer)
        # Add layer to the network
        self.layers.append(layer)

    def test_on_batch(self, X, y):
        """ Evaluates the model over a single batch of samples """
        _, y_pred = self._forward_pass(X, y, training=False)
        loss = np.mean(self.loss_function.loss(y, y_pred))
        acc = self.loss_function.acc(y, y_pred)

        return loss, acc

    def train_on_batch(self, X, y):
        """ Single gradient update over one batch of samples """
        _, y_pred = self._forward_pass(X, y)
        loss = np.mean(self.loss_function.loss(y, y_pred))
        acc = self.loss_function.acc(y, y_pred)
        # Backpropagate. Update weights
        self._backward_pass(loss_grad=self.loss_function.gradient(y,
y_pred))

        return loss, acc

    def fit(self, X, y, n_epochs, batch_size):
        """ Trains the model for a fixed number of epochs """
        best_val_acc = -1.0 # 记录在验证数据集上最佳的正确率
        delta_threshold = 0.01 # 当变化幅度超过时算显著改善或恶化1%
        run_epochs_max = 3 # 连续多少个没有显著改进的停止条件epoch
        run_epochs = 0 # 已经运行了多少个，当有显著改进时清零epoch
        for _ in self.progressbar(range(n_epochs)):
            batch_error = []
            for X_batch, y_batch in NpaiDs.batch_iterator(X, y,
batch_size=batch_size):
                loss, _ = self.train_on_batch(X_batch, y_batch)
                batch_error.append(loss)

            self.errors["training"].append(np.mean(batch_error))

            if self.val_set is not None:
                val_loss, _ = self.test_on_batch(self.val_set["X"], self.
val_set["y"])
```

```
85                 self.errors["validation"].append(val_loss)
86
87         return self.errors["training"], self.errors["validation"]
88
89     def _forward_pass(self, X, y, training=True):
90         """ Calculate the output of the NN """
91         layer_output = X
92         for layer in self.layers:
93             Z, layer_output = layer.forward_pass(layer_output, Y=y,
    training=training)
94
95         return Z, layer_output
96
97     def _backward_pass(self, loss_grad):
98         """ Propagate the gradient 'backwards' and update the weights in
    each layer """
99         for layer in reversed(self.layers):
100             loss_grad = layer.backward_pass(loss_grad)
101
102     def summary(self, name="Model Summary"):
103         # Print model name
104         print (AsciiTable([[name]]).table)
105         # Network input shape (first layer's input shape)
106         print ("Input Shape: %s" % str(self.layers[0].input_shape))
107         # Iterate through network and get each layer's configuration
108         table_data = [["Layer Type", "Parameters", "Output Shape"]]
109         tot_params = 0
110         for layer in self.layers:
111             layer_name = layer.layer_name()
112             params = layer.parameters()
113             out_shape = layer.output_shape()
114             table_data.append([layer_name, str(params), str(out_shape)])
115             tot_params += params
116         # Print network configuration table
117         print (AsciiTable(table_data).table)
118         print ("Total Parameters: %d\n" % tot_params)
119
120     def predict(self, X):
121         """ Use the trained model to predict labels of X """
122         return self._forward_pass(X, training=False)
```

Listing 34: 多层感知器（MLP）模型类定义

初始化 我们首先来看初始化过程：

- 第 24 行：定义构造函数，参数：
  - optimizer：优化器，即调整模型参数使代价函数达到最小值的算法，这里我们采用 Adam 算法；
  - loss：代价函数，由模型参数决定，用于衡量模型好坏，机器学习和深度学习算法的目标就是使代价函数值最小；

  - validation_data：验证数据集，在训练网络时，从原始训练数据集中拿出一部分（如 10∼20%）作为验证数据集，其余才作为真正的训练数据集。在训练数据集学习一段时间后，会拿验证样本集来计算精度，并与之前计算的精度进行比较，当有显著改进时才继续学习，当很长时间没有改进甚至出现恶化时，表明此时已经出现过拟合（Over-fitting），此时就应该停止学习（early stopping），以在验证数据集上取得最高精度时的参数值作为最终的模型参数；

  ；

- 第 25 行：指定优化器算法，这里我们使用 Adam 算法；
- 第 26 行：以列表形式存放组成多层感知器（MLP）模型的层；
- 第 27 行：以列表形式记录在训练数据集和验证数据集上误差的历史数据，我们将根据这些数据绘制误差变化曲线；
- 第 28 行：指定代价函数，这里我们采用交叉熵 CrossEntropy 函数；
- 第 29 行：用于显示训练进度；
- 第 31∼34 行：为验证数据集赋值；
- 第 36 行：定义用于设置模型中哪些层需要参与训练；
- 第 38、39 行：逐层设置每层是否参与训练，这里设置每层均参与训练。如果在图像识别等应用中，我们使用 ResNet 或 Inception 等预训练模型时，我们可以设置底部的层不参与训练，这样可以固定这些层的参数，加快训练过程；

添加层 我们通过向模型中添加层来构造完整的模型，添加层方法如下所示：

- 第 41 行：定义添加层的方法，layer 为要添加的层，可以添加的层定义在 ann.layer 目录下；
- 第 46、47 行：如果要加入的层需要初始化，则初始化该层；
- 第 49 行：将该层添加为层列表属性的最后一层；

前向及反向传播 接下来我们来看模型前向传播产生网络输出，以及反向传输根据代价函数更新模型参数的过程。
我们首先来看前向传播过程实现：

- 第 89 行：定义模型前向传播方法；
- 第 91 行：将网络输出作为初始的上层输出；
- 第 92、93 行：遍历层列表，依次调用每一行的前向传播方法，将前一层的输出作为后一层的输入，以此类推，最终得到整个模型的输出；
- 第 95 行：将层列表中最后一层的输出作为模型的输出返回；

接下来我们来看反向传播过程：

- 第 97 行：定义模型反向传播方法，参数 loss_grad 就模代价函数对模型输出的微分；
- 第 99、100 行：遍历层列表中所有层，调用每一层的反向传播方法，得到该层对模型参数的微分，并将前一层对模型参数的微分作为参数，继续调用下一层的反向传播方法，直到所有层循环结束；

5.2.4 神经网络层定义

我们定义的模型是由层来组成的，在这一部分，我们将详细讲解本例中所用到的层。

Softmax 层 在通常情况下，Softmax 通常做为多分类问题的输出层，其会与交叉熵（CrossEntropy）代价函数相配合，我们在“Softmax 层实现技术”一节中，已经对这个应用场景进行了详细的描述。在本节中，我们将向大家介绍，利用 Numpy 来实现 Softmax 激活函数加交叉熵（）代价函数的情况。
我们首先定义 Softmax 层类（app/layer/ces_layer.py）：

```
# 输出层加交叉熵（）代价函数用于多分类问题Softmax
import math
import copy
import numpy as np

class CesLayer(object):
    def __init__(self, optimizer, K, N):
        '''
        参数:
          : 本层神经元个数K
          N: 特征维度（下一层神经元数）
        '''
        self.name = 'ann.layer.CesLayer'
        self.X = None
        self.Y = None
        self.Y_ = None
        self.W = None # 连接权值
        self.b = None # 偏置值
        self.K = K
        self.N = N
        self.trainable = True # 是否参加训练过程
        self.activation = self.softmax
        #self.initialize(optimizer)

    def layer_name(self):
        return 'Softmax_CrossEntropy'

    def parameters(self):
        return np.prod(self.W.shape) + np.prod(self.b.shape)

    def output_shape(self):
        return (self.K, )

    def initialize(self, optimizer):
        '''
        初始化网络参数
        '''
        # Initialize the weights
        limit = 1 / math.sqrt(self.N)
        self.W  = np.random.uniform(-limit, limit, (self.K, self.N))
```

```
        self.b = np.zeros((self.K, 1))
        # Weight optimizers
        self.W_opt  = copy.copy(optimizer)
        self.b_opt = copy.copy(optimizer)

    def forward_pass(self, X, Y, training=True):
        '''
        前向传播过程
        参数:
          : 输入信号，XM*，其中为迷你批次大小NM
          : 正确值，Yone-向量形式，hotM*K
        '''
        Z = X.dot(np.transpose(self.W)) + np.transpose(self.b)
        Y_ = self.softmax(Z)
        self.X = X
        self.Y = Y
        self.Y_ = Y_
        return Z, Y_

    def backward_pass(self, accum_grad):
        org_W = self.W
        M, _ = self.X.shape
        delta = self.Y_ - self.Y
        pJ_pW_raw = None
        pJ_pb_raw = None
        pJ_pX = None

        for m in range(M):
            y = delta[m, :].reshape((self.K, 1))
            vx = np.transpose(y).dot(org_W)
            if self.trainable:
                xt = self.X[m, :].reshape((1, self.N))
                v = y.dot(xt)
                if pJ_pW_raw is None:
                    pJ_pW_raw = np.array([v])
                else:
                    pJ_pW_raw = np.append(pJ_pW_raw, [v], axis=0)
                if pJ_pb_raw is None:
                    pJ_pb_raw = np.array([y])
                else:
                    pJ_pb_raw = np.append(pJ_pb_raw, [y], axis=0)
            if pJ_pX is None:
                pJ_pX = np.array(vx)
            else:
                pJ_pX = np.append(pJ_pX, vx, axis=0)
        # 更新模型参数
        if self.trainable:
            pJ_pW = np.sum(pJ_pW_raw, axis=0)
            pJ_pb = np.sum(pJ_pb_raw, axis=0)
            self.W = self.W_opt.update(self.W, pJ_pW)
```

```
            self.b = self.b_opt.update(self.b, pJ_pb)
        return pJ_pX

    def softmax(self, X):
        e_x = np.exp(X - np.max(X, axis=-1, keepdims=True))
        return e_x / np.sum(e_x, axis=-1, keepdims=True)
```

Listing 35: Softmax 层类定义

前向传播和反向传播单元测试用例如下所示（tdd.ann.layer.t_ces_layer.py）：

```
# CrossEntropy and 层测试类Softmax
import numpy as np
import unittest
from ann.layer.ces_layer import CesLayer

class TCesLayer(unittest.TestCase):
    def test_forward_pass(self):
        np.random.seed(100)
        # X: M*N=4*3
        X = np.array([
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9],
            [10, 11, 12]
        ])
        M, N = X.shape
        # : YM*K=4*5
        Y = np.array([
            [1, 0, 0, 0, 0],
            [0, 1, 0, 0, 0],
            [0, 0, 1, 0, 0],
            [0, 0, 0, 1, 0]
        ])
        _, K = Y.shape
        ces = CesLayer(None, K, N)
        print('W:{0}'.format(ces.W))
        print('b:{0}'.format(ces.b))
        Z, Y_, J = ces.forward_pass(X, Y)
        print('Z:{0}'.format(Z))
        print('Y_:{0}'.format(Y_))
        print('J:{0}'.format(J))

    def test_backward_pass(self):
        np.random.seed(100)
        # X: M*N=4*3
        X = np.array([
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9],
            [10, 11, 12]
```

```
41          ])
42          M, N = X.shape
43          Y = np.array([
44              [1, 0, 0, 0, 0],
45              [0, 1, 0, 0, 0],
46              [0, 0, 1, 0, 0],
47              [0, 0, 0, 1, 0]
48          ])
49          _, K = Y.shape
50          Y_ = np.array([
51              [1.99079000e-01, 5.24721261e-02, 3.01297519e-01, 4.03417091e
    -01, 4.37342639e-02],
52              [6.30149785e-02, 6.40152167e-03, 3.64466069e-01, 5.65024351e
    -01, 1.09308014e-03],
53              [1.59187808e-02, 6.23283236e-04, 3.51857098e-01, 6.31579034e
    -01, 2.18036827e-05],
54              [3.83084009e-03, 5.78103880e-05, 3.23588995e-01, 6.72521940e
    -01, 4.14310477e-07]
55          ])
56          ces = CesLayer(None, K, N)
57          ces.backward_pass(X, Y, Y_)
```

Listing 36: Softmax 层类单元测试用例

执行如下命令运行前向传播过程：

```
1 python -m unittest tdd.ann.layer.t_ces_layer.TCesLayer.test_forward_pass
```

Listing 37: Softmax 层类单元测试用例运行命令

代码35解读如下所示：

- 第 7 行：定义 CesLayer 类的构造函数；
- 第 17 行：连接权值矩阵 $W \in R^{K \times N}$，其中 K 为本层神经元数，N 底层神经元数；
- 第 18 行：偏置值向量 $\boldsymbol{b} \in R^{K}$；
- 第 19 行：K 为本层的神经元数；
- 第 20 行：N 为底层的神经元数；
- 第 18 行：是否参加训练，即调整参数值。为 False 时，则只传递微分而不调整参数，这在使用预训练模型，需要固定底层网络层连接权值时使用；
- 第 21 行：激活函数选择 Softmax 函数；
- 第 22 行：调用初始化方法；
- 第 39 行：根据特征数求出 $limit = \frac{1}{\sqrt{N}}$；
- 第 40 行：利用 $[-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}]$ 之间均匀分布的随机数来初始化连接权值矩阵 $W \in R^{K \times N}$；
- 第 41 行：用 0 来初始化偏置值 $\boldsymbol{b}$；
- 第 43 行：指定调速连接权值 W 的优化器；
- 第 44 行：指定调整偏置值 $\boldsymbol{b}$ 的优化器；

- 第 53 行：求本层线性和：

$$Z = X \cdot W^T + \boldsymbol{b}^T, \quad Z \in R^{M \times K}, \quad X \in R^{M \times N}, \quad W \in R^{K \times N}, \quad \boldsymbol{b} \in R^K \tag{237}$$

- 第 54 行：利用 Softmax 函数求出本行输出 $\hat{Y}$；
- 第表3462 行：根据交叉熵公式：

$$\mathcal{J} = numpy.sum(-Y \odot \log \hat{Y}) \tag{238}$$

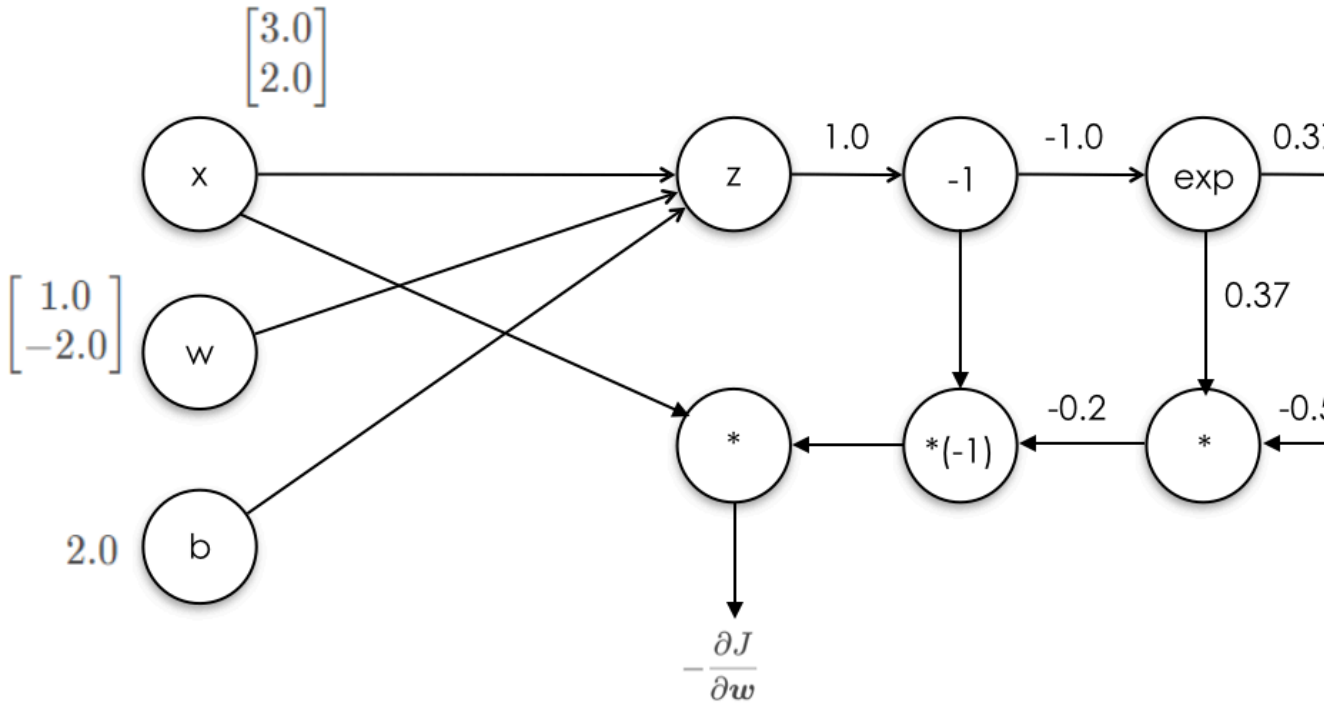
上式是进行 element-wise 相乘，然后利用 numpy.sum 对所有元素进行求和，最后得到为标量的代价函数。

接下来我们来看代码36中运行前向传播方法的测试用例：

- 第 7 行：定义测试前向传播的方法；
- 第 10~15 行：生成输入信号设计矩阵 $X \in R^{M \times N}$；
- 第 16 行：由 X 求出批次大小 $M = 4$ 且特征数 $N = 3$；
- 第 18~23 行：生成 one-hot 向量组成的正确标签 $Y \in R^{M \times K}$；
- 第 24 行：由 Y 求出类别数 $K = 5$；
- 第 25 行：生成 Softmax 层对象，并完成连接权值和偏置值初始化；
- 第 26、27 行：打印连接权值和偏置值内容；
- 第 28 行：调用前向传播方法，求出本层线性输入 Z、本层输出 $\hat{y}$、代价函数值 $\mathcal{J}$；

运行结果如下所示：

Figure 69: Softmax 层前向传播测试用例运行结果



接下来我们来看反向传播过程。在代码35中反向传播代码解读如下所示：

- 第 60 行：定义反向传播方法，参数：$X \in R^{M\times N}$ 为输入信号，$Y \in R^{M\times K}$ 为 one-hot 向量形式正确标签，$\hat{Y} \in R^{M\times K}$ 为 Softmax 函数的输出值，代表各类别出现的概率；
- 第 61 行：记录当前的连接权值矩阵，当向底层传播时，需要使用未经调整过的连接权值进行计算；
- 第 62 行：通过输入信号 $X \in R^{M\times N}$ 求出批次大小 M；
- 第 63 行：求出正确结果与计算结果之间的差异 $delta \in R^{M\times K}$；
- 第 64 行：记录求 $\frac{\partial \mathcal{J}}{\partial W}$ 时的中间结果，我们是先求出对每个样本的结果，再求出批次内样本的和，$pJ_pW_raw \in R^{M\times(K\times N)}$；
- 第 65 行：记录求 $\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}}$ 的中间结果，$pJ_pb_raw \in R^{M\times K}$；
- 第 66 行：保存 $\frac{\partial \mathcal{J}}{\partial X}$ 的值，其将向底层网络继续传播；
- 第 68~85 行：对于批次中每个样本循环处理；
- 第 69 行：取出第 m 个样本，求出正确结果与计算结果的差异 $y \in R^{1\times K}$；
- 第 70 行：计算 $vx = (\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}) \cdot W, \quad vx \in R^{1\times N}$；
- 第 71~81 行：如果本行参与训练过程则执行下面代码：
- 第 72 行：将 $\boldsymbol{x}^{(i)} \in R^{N\times 1}$ 变为 $xt \in R^{1\times N}$；
- 第 73 行：计算 $v = (\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}) \cdot \boldsymbol{x}^T$；
- 第 74、75 行：将上面计算的 v 添加到 pJ_pW_raw 的后面；
- 第 76、77 行：将 $\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}$ 添加到 pJ_pb_raw 的后面；
- 第 78、79 行：将 $vx = (\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}) \cdot W, \quad vx \in R^{1\times N}$ 添加到 pJ_pX 后面，以便于向底层传播；
- 第 80、81 行：求出利用 np.sum(*, axis=0) 对批次中每个样本的参数微分求和，然后调用优化器调整参数值；
- 第 82~85 行：求该批次的 $\frac{\partial \mathcal{J}}{\partial X}$；
- 第 87 行：若本层参与训练执行 88~91 行；
- 第 88 行：对批次所有样本的 $\frac{\partial \mathcal{J}}{\partial W}$ 求和；
- 第 89 行：对批次所有样本的 $\frac{\partial \mathcal{J}}{\partial b}$ 求和；
- 第 90 行：调用权值优化器根据权值的微分更新权值；
- 第 91 行：调用偏置值优化器根据偏置值微分更新偏置值；
- 第 92 行：返回 $\frac{\partial \mathcal{J}}{\partial X}$，以便于向底层传播；

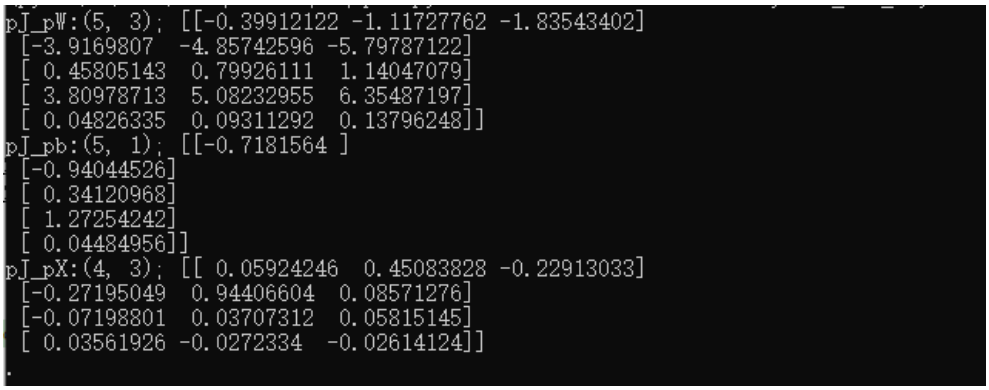
接下来我们看代码36中运行反向传播测试用例的代码：

- 第 33 行：定义反向传播单元测试方法；
- 第 34 行：设置随机数种子，使每次运行时的结果相同；
- 第 36~41 行：定义输入信号 $X \in R^{M\times N}$；
- 第 42 行：从 X 中求出批次大小 M 和特征数（底层神经网络层神经元数）N；
- 第 43~48 行：定义正确结果 $Y \in R^{M\times K}$；
- 第 44 行：从 Y 中求出类别数 K，同时也是本层神经元数；

- 第 50~55 行：定义本层的输出层 $\hat{Y} \in R^{M \times K}$；
- 第 56 行：生成并初始化 Softmax 层；
- 第 57 行：运行反向传播算法；

运行结果如下所示：

Figure 70: Softmax 层反向传播运行结果



在上面的代码中，参数的调整是通过优化器来进行的，我们这里使用的是 Adam 优化算法，具体细节见优化器章节。

LeakyReLU 层 我们在神经网络的隐藏层中，通常使用 ReLU 及其变种来做为激活函数，在这里我们使用 LeakyReLU 作为激活函数。
我们先来看 LeakyReLU 类定义（ann/pytorch/book/chp004/e1/leaky_relu.py）：

```
1  # 全连接层并用采用作为激活函数LeakyReLU
2  import math
3  import copy
4  import numpy as np
5
6  class FclrLayer(object):
7      def __init__(self, optimizer, K, N, epsilon=0.2):
8          '''
9          参数:
10           : 本层神经元个数K
11           N: 特征维度（下一层神经元数）
12         '''
13         self.epsilon = epsilon
14         self.name = 'ann.layer.FclrLayer'
15         self.W = None # 连接权值
16         self.b = None # 偏置值
17         self.W_opt  = None
18         self.b_opt = None
19         self.K = K
20         self.N = N
21         self.trainable = True # 是否参加训练过程
22         self.activation = self.leaky_relu
23         self.initialize(optimizer)
```

```
24
25      def initialize(self, optimizer):
26          '''
27          初始化网络参数
28          '''
29          # Initialize the weights
30          limit = 1 / math.sqrt(self.N)
31          self.W  = np.random.uniform(-limit, limit, (self.K, self.N))
32          self.b = np.zeros((self.K, 1))
33          # Weight optimizers
34          self.W_opt  = copy.copy(optimizer)
35          self.b_opt = copy.copy(optimizer)
36
37      def leaky_relu(self, X):
38          return np.where(X >= 0, X, self.epsilon * X)
39
40      def forward_pass(self, X, Y, training=True):
41          '''
42          前向传播过程
43          参数:
44            : 输入信号，XM*，其中为迷你批次大小NM
45            : 正确值，Yone-向量形式，hotM*K
46          '''
47          Z = X.dot(np.transpose(self.W)) + np.transpose(self.b)
48          Y_ = self.activation(Z)
49          return Z, Y_
50
51      def backward_pass(self, accum_grad, X, Y, A):
52          org_W = self.W
53          M, N = X.shape
54          _, K = accum_grad.shape
55          # 求出的微分leaky_relu
56          A[A>0] = 1
57          A[A<=0] = self.epsilon
58          pJ_pW_raw = None
59          pJ_pb_raw = None
60          pJ_pX = None
61          for i in range(M):
62              gi = accum_grad[i, :]
63              ai = A[i, :]
64              gai = gi * ai
65              gvw = gai.dot(org_W)
66              if self.trainable:
67                  gai = gai.reshape((K, 1))
68                  xi = X[i, :].reshape((1, N))
69                  gai_xi = gai.dot(xi)
70                  if pJ_pW_raw is None:
71                      pJ_pW_raw = np.array([gai_xi])
72                  else:
73                      pJ_pW_raw = np.append(pJ_pW_raw, [gai_xi], axis=0)
```

```
                if pJ_pb_raw is None:
                    pJ_pb_raw = np.array([gai])
                else:
                    pJ_pb_raw = np.append(pJ_pb_raw, [gai], axis=0)
            if pJ_pX is None:
                pJ_pX = np.array([gvw])
            else:
                pJ_pX = np.append(pJ_pX, [gvw], axis=0)
        if self.trainable:
            pJ_pW = np.sum(pJ_pW_raw, axis=0)
            pJ_pb = np.sum(pJ_pb_raw, axis=0)
            self.W = self.W_opt.update(self.W, pJ_pW)
            self.b = self.b_opt.update(self.b, pJ_pb)
        print('pJ_pW:{0}'.format(pJ_pW.shape))
        print('pJ_pb:{0}'.format(pJ_pb.shape))
        return pJ_pX
```

Listing 38: 全连接加 LeakyReLU 层

单元测试用例代码如下所示（tdd/app/layer/t_fclr_layer.py）：

```
import unittest
import numpy as np
from ann.layer.fclr_layer import FclrLayer
from ann.optimizer.adam import Adam

class TFclrLayer(unittest.TestCase):
    def test_leaky_relu(self):
        K = 5
        N = 3
        Z = np.array([
            [0.1, 0.2, 0.3, -1.0, 2.0],
            [0.5, -0.2, 0.8, 0.05, -1.5]
        ])
        fc = FclrLayer(Adam(), K, N)
        Y_ = fc.leaky_relu(Z)
        print(Y_)

    def test_forward_pass(self):
        print('开发辅助测试程序')
        X = np.array([
            [101, 102, 103],
            [201, 202, 203]
        ])
        # M*K
        Y = np.array([
            [1, 0, 0, 0, 0],
            [0, 1, 0, 0, 0]
        ])
        _, N = X.shape
        K = 5
```

```
31          fc = FclrLayer(Adam(), K, N)
32          Z, Y_ = fc.forward_pass(X, Y)
33          print('Y_:{0}; \r\n{1}'.format(Y_.shape, Y_))
34
35
36
37      def test_backward_pass(self):
38          print('开发辅助测试程序')
39          X = np.array([
40              [101, 102, 103],
41              [201, 202, 203]
42          ])
43          M, N = X.shape
44          # M*K
45          accum_grad = np.array([
46              [11, 12, 13, 14, 15],
47              [21, 22, 23, 24, 25]
48          ])
49          _, K = accum_grad.shape
50          # M*K
51          Y = np.array([
52              [1, 0, 0, 0, 0],
53              [0, 1, 0, 0, 0]
54          ])
55          Y_ = np.array([
56              [0.1, 0.2, -13, 1.4, -1.5],
57              [0.2, -0.22, 0.23, -0.1, -0.2]
58          ])
59          fc = FclrLayer(Adam(), K, N)
60          pJ_pX = fc.backward_pass(accum_grad, X, Y, Y_)
61          print('pJ_pX:{0}'.format(pJ_pX))
```

Listing 39: 全连接加 LeakyReLU 层测试用例

我们先来看类的定义和初始化，见代码38：

- 第 7 行：定义构造函数，参数：optimizer 为优化器我们统一采用 Adam 算法，K 为本层神经元数量，N 为底层神经元数量，epsilon 为当 leaky_relu 函数为负数时的系数；
- 第 15 行：连接权值矩阵 $W \in R^{K \times N}$，其中 K 为本层神经元数，N 底层神经元数；
- 第 16 行：偏置值向量 $\boldsymbol{b} \in R^K$；
- 第 17 行：定义连接权值参数调整优化器；
- 第 18 行：定义偏置值参数调整优化器；
- 第 19 行：K 为本层的神经元数；
- 第 20 行：N 为底层的神经元数；
- 第 21 行：是否参加训练，即调整参数值。为 False 时，则只传递微分而不调整参数，这在使用预训练模型，需要固定底层网络层连接权值时使用；

- 第 22 行：激活函数选择 leaky_relu 函数；
- 第 25 行：调用初始化方法；
- 第 30 行：根据特征数求出 $limit = \frac{1}{\sqrt{N}}$；
- 第 31 行：利用 $[-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}]$ 之间均匀分布的随机数来初始化连接权值矩阵 $W \in R^{K \times N}$；
- 第 32 行：用 0 来初始化偏置值 $\boldsymbol{b}$；
- 第 34 行：指定调速连接权值 W 的优化器；
- 第 35 行：指定调整偏置值 $\boldsymbol{b}$ 的优化器；

接下来我们来看 LeakyReLU 激活函数实现和单元测试代码。我们首先来看 LeakyReLU 激活函数实现（见在代码38）：

- 第 37 行：定义 leaky_relu 函数，参数 X 为输入信号，即底层的输出值；
- 第 38 行：当输入 X 的元素大于零时，输出值为该元素值；如果小于等于零时，则其值为 $\epsilon \cdot X_{i,j}$，公式如下所示：

$$Z_{i,j} = \begin{cases} X_{i,j}, & X_{i,j} > 0 \\ -\epsilon \cdot X_{i,j}, & X_{i,j} \leq 0 \end{cases} \tag{239}$$

下面来看单元测试代码（见代码39）：

- 第 7 行：定义测试 leaky_relu 激活函数的单元测试方法；
- 第 8 行：设置底层神经元数量 K；
- 第 9 行：设置本层神经元数量 N；
- 第 10~13 行：设置本层的线性输入；
- 第 14 行：初始化全连接 LeakyReLU 层类实例；
- 第 15 行：调用 leaky_relu 方法；

单元测试运行命令：

```
python -m unittest tdd.ann.layer.t_fclr_layer.TFclrLayer.test_leaky_relu
```

Listing 40: 运行 LeakyReLU 激活函数测试用例

运行结果如下所示：

Figure 71: 运行结果

```
[[ 0.1   0.2   0.3  -0.2   2.  ]
 [ 0.5  -0.04  0.8   0.05 -0.3 ]]
.
```

下面我们来看前向传播过程（见代码38）：

- 第 40 行：定义前向传播方法，参数：X 为输入信号，即底层的输出信号；Y 为正确结果标签；
- 第 47 行：根据公式 $Z = X \cdot W^T + \boldsymbol{b}^T$ 计算本层线性输入；

- 第 48 行：调用 leaky_relu 激活函数，产生本层输出；
- 第 49 行：返回本层线性输入和输出；

接下来我们来看前向传播单元测试用例（见代码39）：

- 第 18 行：定义前向传播单元测试用例；
- 第 20~23 行：定义输入信号；
- 第 25~28 行：定义正确结果；
- 第 29 行：求出底层神经元数量 N；
- 第 30 行：设置本层神经元数量；
- 第 31 行：生成全连接加 LeakyReLU 层类实例；
- 第 32 行：调用前向传播方法；

运行命令为：

```
python -m unittest tdd.ann.layer.t_fclr_layer.TFclrLayer.
    test_forward_pass
```

Listing 41: 运行前向传播测试用例

运行结果如下所示：

Figure 72: 前向传播测试用例运行结果

```
Y_:(2, 5);
[[  1.5820947    5.94112364  65.28152905  32.56603328  65.69328811]
 [  2.63331517  11.72812904 129.39725356  64.78314921 129.93832336]]
.
```

接下来我们来看反向传播实现方法（见代码38）：

- 第 51 行：定义反向传播方法，参数：
  - X：输入信号；
  - Y：正确结果，只有在本层为输出层时才会用；
  - Y_：本层的输出值；
- 第 52 行：备份原始的连接权值；
- 第 53 行：通过输入信号 X 求出迷你批次大小 M 和底层神经元数量 N；
- 第 54 行：根据上层传过来的微分，求出本层神经元数量 K；
- 第 56、57 行：求出 leaky_relu 激活函数的微分：

$$Y_{i,j} = \begin{cases} Z_{i,j}, & Z_{i,j} \geq 0 \\ \epsilon \cdot Z_{i,j}, & Z_{i,j} < 0 \end{cases} \quad \rightarrow \quad \frac{\partial Y_{i,j}}{\partial Z_{i,j}} = \begin{cases} 1, & Z_{i,j} \geq 0 \\ \epsilon, & Z_{i,j} < 0 \end{cases} \tag{240}$$

- 第 58 行：定义代价函数对每个样本求出的连接权值微分列表变量 pJ_pW_raw，其形状为 $R^{M \times (K \times (K \times N))}$；
- 第 59 行：定义代价函数对每个样本求出的偏置值微分列表变量 pJ_pb_raw；
- 第 60 行：定义代价函数对本层输入值的微分，用于继续反向传播；

- 第 61~81 行：对迷你批次中的每个样本循环执行：
  - 第 62 行：根据 LeakyReLU 层实现技术》批量学习中，上层传过来的微分为 g，从中取出当前样本的微分 gi：

$$\begin{bmatrix} g_1 & g_2 & ... & g_K \end{bmatrix} \in R^{1\times K} \tag{241}$$

  - 第 63 行：取出当前样本本层输出 ai：

$$\begin{bmatrix} a_1 & a_2 & ... & a_K \end{bmatrix} \in R^{1\times K} \tag{242}$$

  - 第 64 行：对 vi 和 gi 做 element-wise 相乘，得到 gai：

$$\begin{bmatrix} g_1 & g_2 & ... & g_K \end{bmatrix} \otimes \begin{bmatrix} a_1 & a_2 & ... & a_K \end{bmatrix} = \begin{bmatrix} g_1 * a_1 & g_2 * a_2 & ... & g_K * a_K \end{bmatrix} \in R^{1\times K} \tag{243}$$

  - 第 65 行：对出对于当前样本的代价函数对连接权值的微分：

$$\begin{bmatrix} g_1 * a_1 & g_2 * a_2 & ... & g_K * a_K \end{bmatrix} \cdot \begin{bmatrix} W1,1 & W1,2 & ... & W1,N \\ W2,1 & W2,2 & ... & W2,N \\ ... & ... & ... & ... \\ WK,1 & WK,2 & ... & WK,N \end{bmatrix} \in R^{1\times N} \leftarrow R^{1\times K} \cdot R^{K\times N} \tag{244}$$

  - 第 66~77 行：如果本层参加训练过程，则执行此段代码；
  - 第 67 行：将 gai 由 $R^{1\times K}$ 变为 $R^{K\times 1}$；
  - 第 68 行：将当前输入信号 $\boldsymbol{x}$ 由 $R^{K\times 1}$ 变为 $R^{1\times K}$；
  - 第 69 行：将二者相乘得到 gai_xi：

$$\begin{bmatrix} g_1 * a_1 \\ g_2 * a_2 \\ ... \\ g_K * a_K \end{bmatrix} \cdot \begin{bmatrix} x_1 & x_2 & ... & x_N \end{bmatrix} \in R^{K\times N} \leftarrow R^{K\times 1} \cdot R^{1\times N} \tag{245}$$

  - 第 70~73 行：将上步求出对当前样本的 gai_xi 竖着叠加起来赋给 pJ_pW_raw，最终形状为 $R^{M\times(K\times N)}$；
  - 第 74~77 行：将上面求出的对当前样本的 gai 竖着叠加起来赋给 pJ_pb_raw，最终形状为 $R^{1\times K}$；
  - 第 78~81 行：将上面求出的对当前样本的代价函数对底层输出的微分 gvw 竖着叠加起来赋给 pJ_pX，最终形成形状为 $R^{M\times N}$；
- 第 82~86 行：如果本层参与训练则执行本段代码；
- 第 83 行：将 M 个样本的代价函数对于连接权值的微分叠加起来形成最终的 pJ_pW，形状为 $R^{K\times N}$；
- 第 84 行：将 M 个样本的代价函数对于偏置值的微分叠加起来形成最终的 pJ_pb，形状为 $R^{K\times 1}$；
- 第 85 行：将代价函数对连接权值的微分作为参数调用优化器调整连接权值；
- 第 86 行：将代价函数对偏置值的微分作为参数调用优化器调整偏置值；

接下来我们来看反向传播单元测试代码（见代码39）：

- 第 37 行：定义反向传播单元测试用例；

- 第 39~42 行：定义本层输入信号 X；
- 第 43 行：由本层输入信号 X 求出迷你批次大小 M 和底层神经元数量 N；
- 第 46~48 行：定义上层传过来的微分；
- 第 47 行：确定本层神经元数；
- 第 51~54 行：定义正确结果；
- 第 55~58 行：定义本层计算出的输出；
- 第 59 行：定义全连接加 LeakyReLU 层类实例；
- 第 60 行：调用反向传播方法并得到代价函数对底层输出的微分；
- 第 61 行：打印代价函数对底层输出的微分；

运行命令为：

```
python -m unittest tdd.ann.layer.t_fclr_layer.TFclrLayer.
    test_backward_pass
```

Listing 42: 运行前向传播测试用例

运行结果如下所示：

Figure 73: 前向传播测试用例运行结果

```
pJ_pW:(5, 3)
pJ_pb:(5, 1)
pJ_pX:[[ 6.55990187  6.97126399 -8.75436584]
 [ 6.10220315  9.61176858 -5.47609237]]
.
```

5.2.5 优化器

在深度学习算法中，超参数学习率 α 的选择非常重大，太小的话学习速度过慢，太大的话又可能出现不收敛、反复震荡的情况。针对这些问题，在实际中有很多优化算法，如：SGD、AdaGrad、RMSProp、ADAM 等。最近两三年，ADAM 算法逐渐占据了主导地位，成为优化算法的首选。ADAM 算法结合了 AdaGrad 和 RMSProp 两个算法的优点，同时加上了一些创新，具有如下特点：

1. 计算高效，方便实现，内存使用也很少；
2. 更新步长和梯度大小无关，只和 α、β_1、β_2 有关系。并且由它们决定步长的理论上限；
3. 对目标函数没有平稳要求，即代价函数可以随着时间变化；
4. 能较好的处理噪音样本，并且天然具有退火效果；
5. 能较好处理稀疏梯度，即梯度在很多 step 处都是 0 的情况；

关于 Adam 算法的详细信息，请参考**?**。Adam 算法如1所示。在我们的算法实现中，我们循环只执行一次，因此就省略了循环。实现代码如下所示（ann/pytorch/book/chp004/e1/adam.py）：

Procedure 1 Adam 算法

Input: α：学习率

Input: $\epsilon \leftarrow 10^{-8}$

Input: $\beta_1, \beta_2 \in [0, 1)$：当前时刻预估的指数衰减率

Input: $f(\boldsymbol{\theta})$：为代价函数，θ 为参数

Input: $\boldsymbol{\theta}_0$：初始状态的参数向量；

$m_0 \leftarrow 0$：初始化第一个动量向量；

$v_0 \leftarrow 0$：初始化第二个动量向量；

$t \leftarrow 0$：初始时刻

procedure UPDATE(W, $grad$)

for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.

while θ_t 未收敛： **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\boldsymbol{\theta}} f_t(\boldsymbol{\theta}_{t-1})$

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$

$\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1}$

$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2}$

$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$

return $\boldsymbol{\theta}_t$

```python
import numpy as np

class Adam():
    def __init__(self, learning_rate=0.001, b1=0.9, b2=0.999):
        self.learning_rate = learning_rate
        self.eps = 1e-8
        self.m = None
        self.v = None
        # Decay rates
        self.b1 = b1
        self.b2 = b2

    def update(self, w, grad_wrt_w):
        # If not initialized
        if self.m is None:
            self.m = np.zeros(np.shape(grad_wrt_w))
            self.v = np.zeros(np.shape(grad_wrt_w))

        self.m = self.b1 * self.m + (1 - self.b1) * grad_wrt_w
        self.v = self.b2 * self.v + (1 - self.b2) * np.power(grad_wrt_w,
    2)

        m_hat = self.m / (1 - self.b1)
        v_hat = self.v / (1 - self.b2)

```

```
25          self.w_updt = self.learning_rate * m_hat / (np.sqrt(v_hat) + self
    .eps)
26
27          return w - self.w_updt
```

Listing 43: Adam 算法实现

代码解读如下所示：

- 第 4 行：定义构造函数，学习率和衰减率 β_1 和 β_2 设置初始值；
- 第 5 行：设置学习率 α 的值；
- 第 7 行：设置公式中 m 的初始值；
- 第 8 行：设置公式中 v 的初始值；
- 第 10 行：设置第一个衰减率 β_1；
- 第 11 行：设置第二个衰减率 β_2；
- 第 13 行：定义参数更新方法，参数：w 为当前参数值，grad_wrt_w 为代价函数对参数的微分；
- 第 15~17 行：对公式中的 m 和 v 进行初始化；
- 第 19 行：更新 $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$，其中 g_t 为函数参数 grad_wrt_w；
- 第 20 行：更新 $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$，其中 g_t 为函数参数 grad_wrt_w；
- 第 22 行：计算 $\hat{m}_t = \frac{m_t}{1-\beta_1}$；
- 第 23 行：计算 $\hat{v}_t = \frac{v_t}{1-\beta_2}$；
- 第 25 行：计算参数的调整量 $\alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$；
- 第 27 行：返回调整后的参数；

5.2.6 训练过程

多层感知器类 目前为止，我们已经讲解了隐藏层的标准形式 ReLU 层，以及多分类中输出层的标准形式 Softmax 加交叉熵（Cross Entropy），在本节中，我们将以这些为基础，搭建一个多层感知器模型。

我们首先定义多层感器类（ann/pytorch/book/chp004/e1/mlp_mnist_model.py）：

```
1  from __future__ import print_function, division
2  from terminaltables import AsciiTable
3  import numpy as np
4  import progressbar
5  #from mlfromscratch.utils import batch_iterator
6  #from mlfromscratch.utils.misc import bar_widgets
7  from npai_ds import NpaiDs
8  from npai_plot import bar_widgets
9
10
11 class MlpMnistModel(object):
12     """Neural Network. Deep Learning base model.
13
14     Parameters:
```

```
    ----------
    optimizer: class
        The weight optimizer that will be used to tune the weights in
    order of minimizing
        the loss.
    loss: class
        Loss function used to measure the model's performance. SquareLoss
     or CrossEntropy.
    validation: tuple
        A tuple containing validation data and labels (X, y)
    """
    def __init__(self, optimizer, loss, validation_data=None):
        self.optimizer = optimizer
        self.layers = []
        self.errors = {"training": [], "validation": []}
        self.loss_function = loss()
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)

        self.val_set = None
        if validation_data:
            X, y = validation_data
            self.val_set = {"X": X, "y": y}

    def set_trainable(self, trainable):
        """ Method which enables freezing of the weights of the network's
     layers. """
        for layer in self.layers:
            layer.trainable = trainable

    def add(self, layer):
        """ Method which adds a layer to the neural network """
        # If this is not the first layer added then set the input shape
        # to the output shape of the last added layer
        # If the layer has weights that needs to be initialized
        if hasattr(layer, 'initialize'):
            layer.initialize(optimizer=self.optimizer)
        # Add layer to the network
        self.layers.append(layer)

    def test_on_batch(self, X, y):
        """ Evaluates the model over a single batch of samples """
        _, y_pred = self._forward_pass(X, y, training=False)
        loss = np.mean(self.loss_function.loss(y, y_pred))
        acc = self.loss_function.acc(y, y_pred)

        return loss, acc

    def train_on_batch(self, X, y):
        """ Single gradient update over one batch of samples """
        _, y_pred = self._forward_pass(X, y)
```

```
62          loss = np.mean(self.loss_function.loss(y, y_pred))
63          acc = self.loss_function.acc(y, y_pred)
64          # Backpropagate. Update weights
65          self._backward_pass(loss_grad=self.loss_function.gradient(y,
    y_pred))
66
67          return loss, acc
68
69      def fit(self, X, y, n_epochs, batch_size):
70          """ Trains the model for a fixed number of epochs """
71          best_val_acc = -1.0 # 记录在验证数据集上最佳的正确率
72          delta_threshold = 0.01 # 当变化幅度超过时算显著改善或恶化1%
73          run_epochs_max = 3 # 连续多少个没有显著改进的停止条件epoch
74          run_epochs = 0 # 已经运行了多少个，当有显著改进时清零epoch
75          for _ in self.progressbar(range(n_epochs)):
76              batch_error = []
77              for X_batch, y_batch in NpaiDs.batch_iterator(X, y,
    batch_size=batch_size):
78                  loss, _ = self.train_on_batch(X_batch, y_batch)
79                  batch_error.append(loss)
80
81              self.errors["training"].append(np.mean(batch_error))
82
83              if self.val_set is not None:
84                  val_loss, _ = self.test_on_batch(self.val_set["X"], self.
    val_set["y"])
85                  self.errors["validation"].append(val_loss)
86
87          return self.errors["training"], self.errors["validation"]
88
89      def _forward_pass(self, X, y, training=True):
90          """ Calculate the output of the NN """
91          layer_output = X
92          for layer in self.layers:
93              Z, layer_output = layer.forward_pass(layer_output, Y=y,
    training=training)
94
95          return Z, layer_output
96
97      def _backward_pass(self, loss_grad):
98          """ Propagate the gradient 'backwards' and update the weights in
    each layer """
99          for layer in reversed(self.layers):
100             loss_grad = layer.backward_pass(loss_grad)
101
102     def summary(self, name="Model Summary"):
103         # Print model name
104         print (AsciiTable([[name]]).table)
105         # Network input shape (first layer's input shape)
106         print ("Input Shape: %s" % str(self.layers[0].input_shape))
```

```
107         # Iterate through network and get each layer's configuration
108         table_data = [["Layer Type", "Parameters", "Output Shape"]]
109         tot_params = 0
110         for layer in self.layers:
111             layer_name = layer.layer_name()
112             params = layer.parameters()
113             out_shape = layer.output_shape()
114             table_data.append([layer_name, str(params), str(out_shape)])
115             tot_params += params
116         # Print network configuration table
117         print (AsciiTable(table_data).table)
118         print ("Total Parameters: %d\n" % tot_params)
119 
120     def predict(self, X):
121         """ Use the trained model to predict labels of X """
122         return self._forward_pass(X, training=False)
```

Listing 44: 多层感知器模型类

代码解读如下所示：

- 第 24 行：定义构造函数，参数：
  - optimizer 优化器类，用于调整模型参数；
  - loss 代价函数类；
  - validation_data 验证数据集，主要用于超参数选择和 Early Stopping；
- 第 25 行：设置优化器属性；
- 第 26 行：用列表保存神经网络的层，这里有采用 ReLU 激活函数的全连接层，采用 Softmax 激活函数的输出层；
- 第 27 行：以训练数据集或测试数据集的在迷你批次中代价函数的平均值作为错误率，分别记录在训练数据集和验证数据集下的错误率历史数据供后面绘制在训练数据集和验证数据集上的错误率随训练次数的变化曲线；
- 第 28 行：指定代价函数；
- 第 29 行：我们通过一个进度条表示训练的进展程度；
- 第 41∼49 行：如果有验证数据集参数，则为验证数据集属性赋值，通常验证数据集用于 Early Stopping 算法，即在每学习完一遍训练数据集之后，在验证样本集上计算准确率，如果准确率有提高，则继续训练，如果连续几次在验证数据集上准确率都没有提高，或者在验证数据集上的准确率开始出现下降，即出现过拟合（Over Fitting）现象，则停止训练过程，以验证数据集上准确率最高的一次时的参数作为模型最终的参数，但是在本例中，我们没有采用 Early Stopping 算法，我们会在相应代码处给出提示，读者可以尝试自行添加；
- 第 31∼39 行：向模型中添加神经网络层，添加到 layers 列表中，添加前，如果该层有初始化函数，会先运行初始化函数；

接下来我们来看训练过程，与 sklearn 和 keras 类似，我们训练过程也是调用 fit 方法，如下所示：

- 第 69 行：定义训练方法 fit，参数为：

  - X：样本集，$X \in R^{M \times N}$，其中 M 为迷你批次大小，N 为特征数；
  - y：标签集，即正确结果，与叫 Ground Truth，$y \in R^{M \times K}$，其中 K 为类别数；
  - n_epochs：训练数据集学习的遍数；
  - batch_size：迷你批次大小；
- 第 71 行：定义变量用于存储在验证数据集上取得的最佳正确率；
- 第 72 行：定义在验证数据集上正确率提高 1% 才算有显著改善；
- 第 73 行：连续多少个 epoch 没有显著改善就停止训练过程；
- 第 74 行：已经运行了多少个 epoch，当有显著改善时该值清零；
- 第 75∼85 行：循环学习整个训练样本集共 n_epochs 次，在每次循环体中执行下列语句：
  - 第 76 行：batch_error 用于记录训练数据集上每个迷你批次以代价函数值表示的错误率，将其进行平均后，得到训练数据集上的错误率；
  - 第 77∼79 行：将训练数据集划分为迷你批次，对每个迷你批次执行；
  - 第 80 行：计算在每个迷你批次上的代价函数值，train_on_batch 方法将；
  - 第 81 行：将代价函数值添加到 batch_error 列表中；
- 第 81 行：将所有迷你批次的错误率求出平均值，将其添加到训练样本集错误率列表 self.errors['training'] 中；
- 第 83 行：如果有验证数据集则执行 70、71 行；
- 第 84 行：将整个验证样本集作为一个批次，运行 test_on_batch 方法，求出代价函数值的平均值，我们将在后面具体讲解 test_on_batch 方法；
- 第 85 行：将求出的代价函数值的平均值作为错误率，保存到 self.errors['validation'] 列表中；
- 第 87 行：将训练样本集、验证样本集上的错误率列表返回给调用者，用于绘制错误率曲线随学习次数的变化趋势；

接下来我们来具体的训练过程 train_on_batch：

- 第 59 行：定义 train_on_batch 方法，参数为一个迷你批次的样本集 X 和标签集 y；
- 第 61 行：调用本类的 _forward_pass 方法，求出模型输出层的输出，这里就是调 self.layers 列表中的每一层的 forward_pass 方法，求出每一层的输出，然后作为上面一层的输入，直到输出层为止；
- 第 62 行：根据代价函数求出迷你批次中每个样本的代价函数值，然求出其平均数作为最终的代价函数值；
- 第 63 行：根据代价函数求出在此迷你批次上的精度；
- 第 65 行：调用本类的后向传播算法，求出对模型参数的微分，并自动更新参数值，我们在神经网络层和优化器中已经给大家做了介绍；
- 第 67 行：返回求出的代价函数值和精度；

接下来我们来看 test_on_batch 方法：

- 第 51 行：定义 test_on_batch 方法，参数为一个迷你批次的样本集 X 和标签集 y；

- 第 53 行：调用本类的 _forward_pass 方法，求出模型输出层的输出，这里就是调 self.layers 列表中的每一层的 forward_pass 方法，求出每一层的输出，然后作为上面一层的输入，直到输出层为止；
- 第 54 行：根据代价函数求出迷你批次中每个样本的代价函数值，然求出其平均数作为最终的代价函数值；
- 第 55 行：根据代价函数求出在此迷你批次上的精度；
- 第 57 行：返回求出的代价函数值和精度；

接下来我们来看前向传播方法：

- 第 89 行：定义前向传播方法，参数：X 为迷你批次样本集，y 为迷你批次标签集，training 为是否是训练过程，当在实际进行预测时 training=False，训练过程中为 True；
- 第 91 行：我们需要循环处理 self.layers 列表中所有层，对每一层来说，其输入就是底下一层的输出，此行为进行初始化；
- 第 92 行：对模型中的层进行循环处理；
- 第 93 行：首先求出该层的线性输入：$Z = X \cdot W^T + \boldsymbol{b}$，然后经过激活函数 $A = f(Z)$ 得到本层的输出；
- 第 95 行：返回输出层的线性输入 Z 和输出；

接下来我们来看反向传播过程：

- 第 97 行：定义反向传播方法，参数为其上面一层代价函数对模型参数的微分；
- 第 99、100 行：循环处理 self.layers 中的每一层，首先求出本层代价函数对模型参数的微分，然后调用其底下一层的反向传播方法；

多层感知器应用 在定义了多层感知器模型之后，我们来看定义多层感知器应用类，使用多层感知器模型来开发实际的应用。
多层感知器模应用类型如下所示（app/pytorch/book/chp004/e1/mlp_mnist_app.py）：

```
1  #
2  from __future__ import print_function
3  import csv
4  import numpy as np
5  import matplotlib.pyplot as plt
6  import pandas as pd
7  import sklearn.manifold as skmd
8  from mlp_mnist_model import MlpMnistModel
9  from npai_ds import NpaiDs
10 from adam import Adam
11 from cross_entropy import CrossEntropy
12 #from ann.loss.negative_log_likelyhood import NegativeLogLikelyhood
13 from layer import Layer
14 import layer_common as allc
15 from dense import Dense
16 from dropout import Dropout
17 from activation import Activation
```

```
18 from fclr_layer import FclrLayer
19 from ces_layer import CesLayer
20
21 class MlpMnistApp(object):
22     def __init__(self):
23         self.name = 'ann.ml.MlpApp'
24
25     def run(self):
26         print('用于手写数字识别MLP')
27         # self.show_mnist_in_tsne()
28         # 载入数据集MNIST
29         X, y = self.load_mnist_ds()
30         n_samples, n_features = X.shape
31         n_hidden = 512
32         X_raw, X_test, y_raw, y_test = NpaiDs.train_test_split(X, y,
    test_size=0.2, seed=1)
33         X_train, X_validate, y_train, y_validate = NpaiDs.
    train_test_split(X_raw, y_raw, test_size=0.15, seed=1)
34         model = MlpMnistModel(optimizer=Adam(),
35                               loss=CrossEntropy,
36                               validation_data=(X_validate, y_validate))
37         # 添加连接层
38         model.add(FclrLayer(optimizer=Adam, K=512, N=784, epsilon=0.1))
39         model.add(CesLayer(optimizer=Adam(), K=10, N=512))
40         model.summary()
41         #
42         train_err, val_err = model.fit(X_train, y_train, n_epochs=5,
    batch_size=256)
43         # Training and validation error plot
44         n = len(train_err)
45         training, = plt.plot(range(n), train_err, label="Training Error")
46         validation, = plt.plot(range(n), val_err, label="Validation Error
    ")
47         plt.legend(handles=[training, validation])
48         plt.title("Error Plot")
49         plt.ylabel('Error')
50         plt.xlabel('Iterations')
51         plt.show()
52
53         _, accuracy = model.test_on_batch(X_test, y_test)
54         print ("Accuracy:", accuracy)
55
56
57     def load_mnist_ds(self):
58         # 文件下载链接：CSVhttps://www.openml.org/d/554
59         # 从网络上获取数据集：X, y = skds.fetch_openml('mnist_784', \
60         # version=1, return_X_y=True)
61         with open('data/mnist_784.csv', newline='', encoding='UTF-8') as
    fd:
62             rows = csv.reader(fd, delimiter=',', quotechar='|')
```

```
            X0 = None
            y0 = None
            next(rows)
            cnt = 0
            rst = 0
            amount = 1000 # 每条记录保存一次1000
            X = None
            y = None
            for row in rows:
                x = np.array(row[:784], dtype=np.float)
                x /= 255.0
                y_ = np.array(row[784:])
                if None is X:
                    X = np.array([x])
                    y = np.zeros((1, 10))
                    y[cnt, int(y_[0])] = 1
                else:
                    X = np.append(X, x.reshape(1, 784), axis=0)
                    yi = np.zeros((1, 10))
                    yi[0, int(y_[0])] = 1
                    y = np.append(y, yi.reshape(1, 10), axis=0)
                if cnt % amount == 0 and cnt > 0:
                    if None is X0:
                        X0 = X
                        y0 = y
                    else:
                        X0 = np.append(X0, X, axis=0)
                        y0 = np.append(y0, y, axis=0)
                    X = None
                    y = None
                    cnt = 0
                    rst += amount
                    print('处理完记录{0}'.format(rst))
                else:
                    cnt += 1
        return X0, y0

    def show_mnist_in_tsne(self):
        X, y_ = self.load_mnist_ds()
        y = np.argmax(y_, axis=1)
        row_embedded = skmd.TSNE(n_components=2).fit_transform(X)
        pos = pd.DataFrame(row_embedded, columns=['X', 'Y'])
        pos['species'] = y
        ax = pos[pos['species']==0].plot(kind='scatter', x='X', y='Y',
    color='blue', label='0')
        pos[pos['species']==1].plot(kind='scatter', x='X', y='Y', color='
    red', label='1', ax=ax)
        pos[pos['species']==2].plot(kind='scatter', x='X', y='Y', color='
    green', label='2', ax=ax)
```

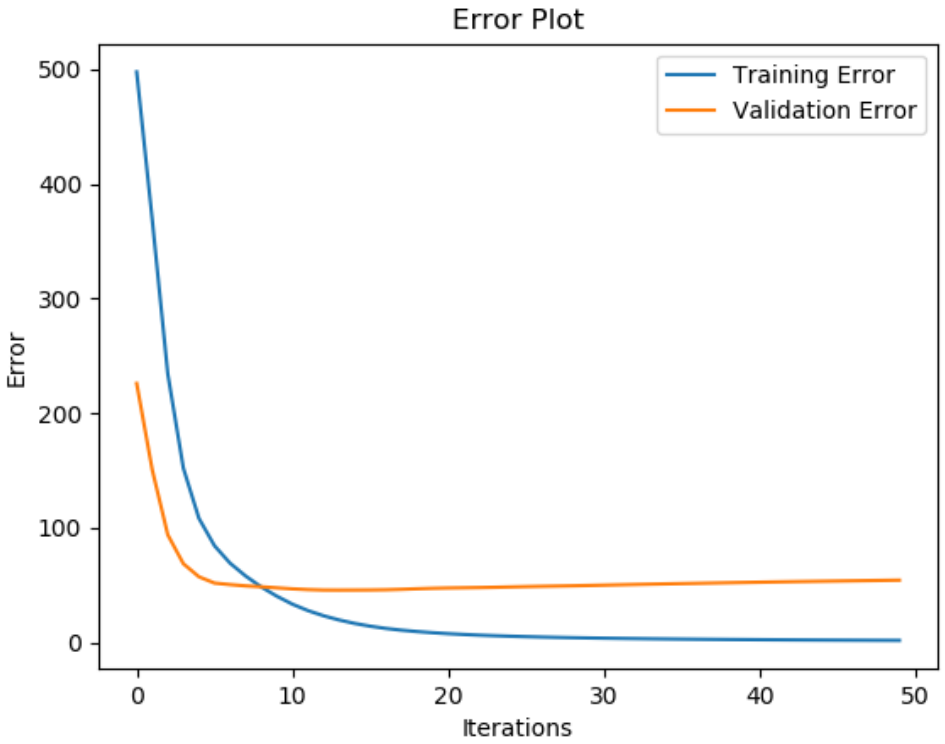
```
109         pos[pos['species']==3].plot(kind='scatter', x='X', y='Y', color='
    yellow', label='3', ax=ax)
110         pos[pos['species']==4].plot(kind='scatter', x='X', y='Y', color='
    brown', label='4', ax=ax)
111         pos[pos['species']==5].plot(kind='scatter', x='X', y='Y', color='
    orange', label='5', ax=ax)
112         pos[pos['species']==6].plot(kind='scatter', x='X', y='Y', color='
    black', label='6', ax=ax)
113         pos[pos['species']==7].plot(kind='scatter', x='X', y='Y', color='
    pink', label='7', ax=ax)
114         pos[pos['species']==8].plot(kind='scatter', x='X', y='Y', color='
    purple', label='8', ax=ax)
115         pos[pos['species']==9].plot(kind='scatter', x='X', y='Y', color='
    cyan', label='9', ax=ax)
116         plt.show()
```

Listing 45: 多层感知器模型应用类

在这里我们仅列出了多层感知器模型应用类训练方法，关于 MNIST 手写数字识别数据集的载入和可视免费下载已经在 MNIST 数据集简介中进行了介绍，大家可以参考该节内容。多层感知器模型应用类代码解读如下所示：

- 第 29 行：载入 MNIST 手写数字识别数据集，X 为样本集，y 为标签集；
- 第 30 行：标本集 $X \in R^{M \times N}$，取出样本数量为 n_samples，特征数量为 n_features；
- 第 31 行：定义隐藏层神经元数为 512；
- 第 32 行：将数据集进行随机排列后，将 20% 的样本作为测试数据集，其余作为原始数据集；
- 第 33 行：将原始数据集进行随机排列后，将 15% 的样本作为验证数据集，其余作为训练数据集；
- 第 34~36 行：定义多层感知器模型类实例，优化器采用 Adam 算法，代价函数采用交叉熵（Cross Entropy），参数中包括验证数据集；
- 第 38 行：添加以 Leaky ReLU 为激活函数的全连接层，参数优化器为 Adam，底下一层神经元数 $N = 784$，本层神经元数为 $K = 512$，当线性输入 $Z < 0$ 时取的值 $\epsilon = 0.1$；
- 第 39 行：添加 softmax 层并将其作为输出层，参数优化器为 Adam，本层神经元数为 $K = 10$，底下一层神经元数为 $N = 512$；
- 第 42 行：调用多层感知器模型 fit 方法进行训练，将训练数据集中的样本集 X 和标签集 y 作为参数，并且规定学习训练数据集 50 遍，迷你批次的大小为 256，返回值为在训练样本集上的错误值（代价函数值）列表和验证数据集上的错误值列表；
- 第 44~51 行：绘制在训练数据集上的错误率（代价函数值）随学习训练数据集遍数的变化曲线，同时绘制在验证数据集上的错误率（代价函数值）随学习训练数据集遍数的变化曲线，图中横坐标为学习训练数据集的遍数，变化曲线如下所示：

Figure 74: 训练精度变化图



大概在训练 10 遍左右，验证样本集上的错误率就稳定地大于在训练样本集上的错误率了，也就说明从该点开始，模型就开始有过拟合（Over Fitting）现象了，我们可以通过 Early Stopping 算法来识别到这一点，从而获得理好的泛化能力。在多层感知器模型类 Mlp 的 fit 方法中，如果验证数据集上的精确度达到比原来的最大值更大的值，则将其记录为新的最大值，并保留当前的模型参数，同时可以通过识别验证样本集上的精确度 acc 是否稳定上升，如果长时间没有显著上升，甚至出现下降，则停止训练过程，将记录下来的精确度值最大时的模型参数作为训练好的模型参数，这就是著名的 Early Stopping 算法。

- 第 53、54 行：计算并打印训练完的模型在测试样本集上的精度，上面的程序可以达到 90.5% 的精确度；

5.2.7 预测过程

训练好的模型，就可以应用到实际生产环境，进行预测了。在具体讲述预测方法之前，我们需要先来讲解一下模型参数的保存和恢复。

序列化和反序列化 在研究保存和恢复模型参数之前，我们先来研究一下 Python 中的序列化和反序列化问题。在 Python 中通过 pickle 模块，实现 Python 对象的序列化和反序列化，代码如下所示（doc/npai/resources/book/c00/c00/c14.py）：

```
1 import pickle
2 import numpy as np
3
4 class Ann(object):
5     def __init__(self):
```

```
6           self.W = np.array([
7               [1.1, 1.2, 1.3, 1.4, 1.5],
8               [2.1, 2.2, 2.3, 2.4, 2.5],
9               [3.1, 3.2, 3.3, 3.4, 3.5]
10          ])
11          self.b = np.array([
12              [100.1, 100.2, 100.3]
13          ])
14          self.model_file = 'ann.pkl'
15
16      def save_model(self):
17          params = [self.W, self.b]
18          with open(self.model_file, 'wb') as fd:
19              pickle.dump(params, fd)
20
21      def restore_model(self, filename):
22          with open(filename, 'rb') as fd:
23              params = pickle.load(fd)
24          self.W = np.array(params[0])
25          self.b = np.array(params[1])
26
27  def main():
28      print('test pickle')
29      ann = Ann()
30      ann.save_model()
31      ann.W[0, 1] = 2019.0
32      print('W:{0}; {1} {2}'.format(type(ann.W), ann.W.dtype, ann.W))
33      ann.restore_model(ann.model_file)
34      print('W:{0}; {1} {2}'.format(type(ann.W), ann.W.dtype, ann.W))
35
36
37  if '__main__' == __name__:
38      main()
```

Listing 46: pickle 模块应用示例

代码解读如下所示：

- 第 4 行：定义神经网络类；
- 第 5 行：定义构造函数；
- 第 6~10 行：定义连接权值参数；
- 第 11~13 行：定义偏置值参数；
- 第 14 行：定义模型参数的保存文件；
- 第 16 行：定义保存模型参数方法，通常在验证数据集上取得最佳精确度时调用本方法保存模型参数；
- 第 17 行：将连接权值和偏置值放入列表 params 中；
- 第 18 行：以二进制写的方式打开模型保存文件；
- 第 19 行：将参数列表序列化到模型文件中；

- 第 21 行：定义恢复模型参数方法，filename 为模型参数保存文件；
- 第 22、23 行：以二进制读形式打开模型保存文件，将文件内容反序列化到 params 变量中；
- 第 24 行：将 params 列表第一个元素变为 np.ndarray 对象实例作为连接权值；
- 第 25 行：将 params 列表第二个元素变为 np.ndarray 对象实例作为偏置值；
- 第 29 行：创建神经网络类对象；
- 第 301 行：调用 save_model 方法将模型参数保存到文件中；
- 第 31、32 行：修改其中一个连接权值参数，并打印最新的连接权值；
- 第 33、34 行：恢复保存的模型参数，并印恢复的模型参数，运行结果如下所示：

Figure 75: 保存和恢复模型参数运行结果

由上图可以看出，我们确实实现了保存和恢复模型参数的功能，当我们在保存模型参数后，对模型参数进行修改，当恢复模型参数时，还是会恢复之前的模型参数。

预测方法实现 在应用类中修改运行代码（app/pytorch/book/chp004/e1/mlp_mnist_app.py）：

```
1  class MlpMnistApp(object):
2  ......
3      def run(self):
4          ......
5          # 添加连接层
6          model.add(FclrLayer(optimizer=Adam, K=512, N=784, epsilon=0.1))
7          model.add(CesLayer(optimizer=Adam(), K=10, N=512))
8          model.summary()
9          run_mode = 0
10         if 1 == run_mode:
11             #
12             train_err, val_err = model.fit(X_train, y_train, n_epochs=1,
       batch_size=256)
13             # Training and validation error plot
14             n = len(train_err)
15             training, = plt.plot(range(n), train_err, label="Training
       Error")
16             validation, = plt.plot(range(n), val_err, label="Validation
       Error")
17             plt.legend(handles=[training, validation])
18             plt.title("Error Plot")
19             plt.ylabel('Error')
20             plt.xlabel('Iterations')
21             plt.show()
22
23             _, accuracy = model.test_on_batch(X_test, y_test)
```

```
24              print ("Accuracy:", accuracy)
25              model.save_model()
26          else:
27              print('进行预测')
28              Xt = X_test[0:1, :]
29              _, yt = model.predict(Xt)
30              yg = y_test[0:1, :]
31              print('y_hat:{0}; y:{1}'.format(yt, yg))
```

Listing 47: 预测方式实现

代码解读如下所示：

- 第 9 行：增加运行模式变量，为 1 时是训练模式，为 0 时为运行预测模式；
- 第 10~24 行：是原来的训练模型程序，讲解见前面章节；
- 第 25 行：当训练结束后保存模型参数；
- 第 26~31 行：当为运行预测模式时执行；
- 第 28 行：从测试样本集中取出一个样本；
- 第 29 行：求出网络的输出值；
- 第 20 行：取出测试样本集第一个样本的正确结；
- 第 21 行：打印预测结果和正确结果；

接着我们来看对模类的修改（app/pytorch/book/chp004/e1/mlp_mnist_model.py）：

```
1  class MlpMnistModel(object):
2  ......
3      def predict(self, X):
4          """ Use the trained model to predict labels of X """
5          return self._forward_pass(X, y=None, training=False)
6
7      def save_model(self):
8          for layer in self.layers:
9              layer.save_layer()
```

Listing 48: 预测、保存模型方法

在上面代码中，预测方法直接调用前向传播方法求出网络输出。在 save_model 方法中，对所有层进行遍历，依次调用相应层的保存层参数的方法。下面我们以全连接层为例，讲解模型保存和恢复的代码：

```
1  class FclrLayer(object):
2  ......
3      def __init__(self, optimizer, K, N, epsilon=0.2, param_file='work/
       fclr.pkl'):
4      ......
5          if os.path.exists(param_file):
6              self.can_restore_layer = True
7          else:
8              self.can_restore_layer = False
9          self.param_file = param_file
```

```
10
11      def save_layer(self):
12          params = [self.W, self.b]
13          with open(self.param_file, 'wb') as fd:
14              pickle.dump(params, fd)
15
16      def restore_layer(self):
17          with open(self.param_file, 'rb') as fd:
18              params = pickle.load(fd)
19          self.W = np.array(params[0])
20          self.b = np.array(params[1])
21
22      def initialize(self, optimizer):
23          '''
24          初始化网络参数
25          '''
26          if self.can_restore_layer:
27              self.restore_layer()
28          else:
29              # Initialize the weights
30              limit = 1 / math.sqrt(self.N)
31              self.W  = np.random.uniform(-limit, limit, (self.K, self.N))
32              self.b = np.zeros((self.K, 1))
33              # Weight optimizers
34              self.W_opt  = copy.copy(optimizer)
35              self.b_opt = copy.copy(optimizer)
```

Listing 49: 全连接层模型保存和恢复

代码解读如下所示：

- 第 3 行：模造函数中增加了模型参数文件路径参数；
- 第 5、6 行：如果存在参数模型文件，则设置可以恢复模型；
- 第 7、8 行：否则设置不可以恢复模型；
- 第 11~14 行：将连接权值和偏置值保存到模型参数文件中；
- 第 16~20 行：从模型文件中恢复连接权值和偏置值参数；
- 第 26、27 行：如果存在模型参数文件，则从中恢复网络参数；
- 第 28~34 行：否则像上文所述对参数进行初始化；

5.3 PyTorch 方法

在前面的章节中，我们详细讲解了多层感知器（MLP）模型的数学原理和 Numpy 实现技术，由上面的讲解可以看到，如果纯手写一个简单的像多层感知器（MLP）模型，也是非常复杂的事情。因此在实际应用中，我们更多地会选择使用 PyTorch 框架，使我们专注于问题本身，而不是实现细节。在本节中，我们将讲解使用 PyTorch 的典型方法，在下一节中，我们将深入讲解 PyTorch 的可定制性，通过这些特性，我们可以实现更加复杂的功能。

5.3.1 载入数据集

在这里我们使用 torchvision 来载入手写数字识别 MNIST 数据集，如下所示（app/pytorch/book/chp004/e2/mlp_mnist_app.py）：

```
    def load_dataset(self):
        batch_size = 32
        raw_train_set = torchvision.datasets.MNIST(
            root='./data/MNIST',
            train=True,
            download=True,
            transform=transforms.Compose([
                transforms.ToTensor(),
                transforms.Normalize((0.1307,),(0.3081,))
            ])
        )
        train_set, validate_set = torch.utils.data.random_split(
    raw_train_set, [50000, 10000])
        train_loader = torch.utils.data.DataLoader(
            train_set, batch_size=batch_size, shuffle=True
        )
        validate_loader = torch.utils.data.DataLoader(
            train_set, batch_size=batch_size, shuffle=False
        )
        test_set = torchvision.datasets.MNIST(
            root='./data/MNIST',
            train=False,
            download=True,
            transform=transforms.Compose([
                transforms.ToTensor(),
                transforms.Normalize((0.1307,),(0.3081,))
            ])
        )
        test_loader = torch.utils.data.DataLoader(
            test_set, batch_size=10, shuffle=False
        )
        return train_loader, validate_loader, test_loader

    def test_load_dataset(self):
        rand_num = 15
        idx = 3
        train_loader, validate_loader, test_loader = self.load_dataset()
        for batch_idx, (data, target) in enumerate(test_loader):
            rand_num +=1
            Xt = data
            yt = target
            if rand_num > 3:
                break
        Xt0 = Xt[idx]
        Xt0 = torch.unsqueeze(Xt0, 0)
        # 绘制该样本
```

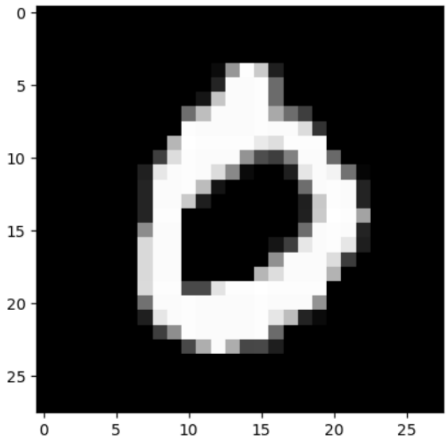
```
46          img = Xt0[0][0].numpy()
47          plt.imshow(img, cmap='gray')
48          plt.show()
```

Listing 50: 利用 torchvision 载入手写数字识别 MNIST 数据集

代码解读如下所示：

- 第 2 行：设置迷你批次大小为 32；
- 第 3∼11 行：调用 torchvision 方法载入原始 MNIST 训练数据集，即 60000 个样本的数据；
- 第 12 行：将原始训练数据集按照 5:1 的比例划分为训练数据集和验证数据集，其中验证数据集仅用于 Early Stopping 和超参数选择，不参与网络参数的训练过程；
- 第 13 行：将训练数据集打乱平排后，按照迷你批次大小进行划分，生成 train_loader；
- 第 14 行：将验证数据集打乱平排后，按照迷你批次大小进行划分，生成 validate_loader；
- 第 19∼27 行：调用 torchvision 方法载入 MNIST 测试数据集；
- 第 28∼30 行：将测试数据集打乱平排后，按照迷你批次大小进行划分，生成 test_loader；
- 第 31 行：返回数据集加载器；
- 第 33 行：定义载入数据集测试程序；
- 第 34 行：随机取的批次编号；
- 第 35 行：选中批次中样本索引号；
- 第 36 行：加载数据集；
- 第 37∼42 行：遍历找到对应的批次；
- 第 43 行：在批次中找到指定索引号的样本；
- 第 44 行：此时我们找到的样本形状为 $[1, 28, 28]$，如果要输入到神经网络中需要为 $[1, 1, 28, 28]$，需要在最前面加一维批次索引号；
- 第 46 行：将其转变化 28×28 的图像格式；
- 第 47、48 行：显示该图像，如下所示：

Figure 76: 显示 MNIST 样本图像



5.3.2 模型定义

下面我们来看模型类的定义（app/pytorch/book/chp004/e2/mlp_mnist_model.py）：

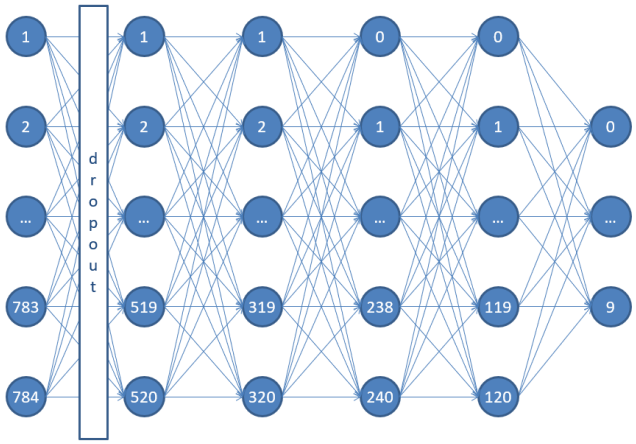
```
#
import torch
from torch.autograd import Variable
import torch.nn.functional as F

class MlpMnistModel(torch.nn.Module):
    def __init__(self):
        super(MlpMnistModel, self).__init__()
        self.l1 = torch.nn.Linear(784, 520)
        self.d1 = torch.nn.Dropout(0.2)
        self.l2 = torch.nn.Linear(520, 320)
        self.l3 = torch.nn.Linear(320, 240)
        self.l4 = torch.nn.Linear(240, 120)
        self.l5 = torch.nn.Linear(120, 10)

    def forward(self, x):
        relu = torch.nn.ReLU()
        # x (batch_index, channel_index, 28, 28) => (batch_index, 784)
        x = x.view(-1, 784)
        z1 = self.l1(x)
        d1 = self.d1(z1)
        a1 = relu(d1)
        #a1 = relu(self.l1(x))
        a2 = relu(self.l2(a1))
        a3 = relu(self.l3(a2))
        a4 = relu(self.l4(a3))
        return self.l5(a4)
```

Listing 51: 多层感知器（MLP）模型类

代码解读如下：

- 第 6 行：定义 MlpMnistModel 类并继承自 torch.nn.Module；
- 第 7、8 行：定义构造函数，并调用基类的构造函数；
- 第 9~14 行：定义网络结构，如下所示：

Figure 77: 网络架构图



从左边起是输入层，紧接着的才是第 1 层。这里需要特别注意的是，我们在第 1 层和第 2 层之间，添加了一个 Dropout 层，该层在训练阶段，会随机屏蔽 20% 的节点，在实际运行时再打开所有的节点，这样相当于同时训练了非常多的神经网络，在运行时，运行结果由这些网络集体投票决定，这项技术可以极大提高网络的精度，是近年来流行的技术之一，我们可以在每层之间均增加 Droput 层，这里为了示例，我们只添加了一层。

- 第 16 行：定义网络前向传播过程；
- 第 19 行：原始输入信号形状为 $[-1, 1, 28, 28]$，我们将其变为 $[-1, 784]$，大家注意到前面第 1 维为 -1，代表是在迷你批次中任意的样本索引号；
- 第 20 行：求第 1 层的线性和；
- 第 21 行：随机屏蔽 20% 神经元，得到最终结果线性输入；
- 第 22 行：求出第 1 层的输出；
- 第 24 行：将第 1 层的输出作为第 2 层的输入，求出线性和然后经过激活函数（ReLU）得到第 2 层的输出；
- 第 25 行：将第 2 层的输出作为第 3 层的输入，求出线性和然后经过激活函数（ReLU）得到第 3 层的输出；
- 第 26 行：将第 3 层的输出作为第 4 层的输入，求出线性和然后经过激活函数（ReLU）得到第 4 层的输出；
- 第 27 行：将第 4 层的输出作为第 5 层的输入，求出线性和作为结果并返回；

在 PyTorch 中，我们只需要定义模型的前向传播过程，反向传播求微分可以使用自动微分技术得到，不用我们编写任何代码，这就是使用 PyTorch 框架方便的地方。

5.3.3 训练和预测过程

下面我们来看模型训练和预测过程（app/pytorch/book/chp004/e2/mlp_mnist_app.py）：

```
class MlpMnistApp(object):
    RUN_MODE_TRAIN = 1
    RUN_MODE_PREDICT = 2

    def __init__(self):
        self.name = ''
        self.model_file = './data/mlp.pt'
        self.optz_file = './data/mlp_opt.pt'

    def run(self, run_mode=RUN_MODE_TRAIN, continue_train=False):
        train_loader, validate_loader, test_loader = self.load_dataset()
        model = MlpMnistModel()
        criterion = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum
    =0.5)
        #optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
        if MlpMnistApp.RUN_MODE_TRAIN == run_mode:
            if continue_train:
                model.load_state_dict(torch.load(self.model_file))
                optimizer.load_state_dict(torch.load(self.optz_file))
            self.train(train_loader, validate_loader, test_loader, model,
     criterion, optimizer)
        else:
            print('载入模型参数并进行预测')
            model.load_state_dict(torch.load(self.model_file))
            model.eval()
            for batch_idx, (data, target) in enumerate(test_loader):
                Xt = data
                yt = target
            idx = 8
            Xt0 = Xt[idx]
            Xt0 = torch.unsqueeze(Xt0, 0)
            yt0 = yt[idx]
            rsts = self.predict(model, Xt0)
            rst = rsts[0]
            print('正确值:{0}; 预测值:{1}'.format(yt0, rst))
            # 绘制该样本
            img = Xt0[0][0].numpy()
            plt.imshow(img, cmap='gray')
            plt.show()

    def predict(self, model, X):
        y_hat = model(X)
        return torch.argmax(y_hat, dim=1)

    def train(self, train_loader, validate_loader, test_loader, model,
    criterion, optimizer):
        best_accuracy = 0.0
        threshold = 0.001 # 精度提高算是有显著提高0.01
        unimproved_epochs = 0
```

```
48          unimproved_patience = 3
49          epochs = 5 #
50          for epoch in range(epochs):
51              self.train_batch(epoch, model, criterion, optimizer,
    train_loader, validate_loader)
52              validate_accuracy = self.evaluate(model, criterion, optimizer
    , validate_loader)
53              print('验证精度：{0}; 最佳精度：{1}; 累积次数：
    {2}'.format(validate_accuracy, best_accuracy, unimproved_epochs))
54              if validate_accuracy > best_accuracy*(1+threshold):
55                  best_accuracy = validate_accuracy
56                  unimproved_epochs = 0
57                  torch.save(model.state_dict(), self.model_file) # 保存模型
58                  torch.save(optimizer.state_dict(), self.optz_file)
59              else:
60                  unimproved_epochs += 1
61              if unimproved_epochs > unimproved_patience:
62                  print('Early 已经运行Stopping')
63                  break
64          test_accuracy = self.evaluate(model, criterion, optimizer,
    test_loader)
65          print('最终精度：{0}'.format(test_accuracy))
66          torch.save(model.state_dict(), self.model_file) # 保存模型
67          torch.save(optimizer.state_dict(), self.optz_file)
68
69      def train_batch(self, epoch, model, criterion, optimizer,
    train_loader, validate_loader):
70          model.train()
71          for batch_idx, (data, target) in enumerate(train_loader):
72              data, target = Variable(data), Variable(target)
73              optimizer.zero_grad()
74              y_hat = model(data)
75              loss = criterion(y_hat, target)
76              loss.backward()
77              optimizer.step()
78              if batch_idx % 10 == 0:
79                  print('train epoch:{0} [{1}/{2}    ({3:.2f})]    loss:{4:.6
    f}'.format(epoch,
80                      batch_idx*len(data), len(train_loader.dataset),
81                      100. * batch_idx /len(train_loader),
82                      loss.data)
83                  )
84
85      def evaluate(self, model, criterion, optimizer, data_loader):
86          model.eval()
87          test_loss = 0
88          correct = 0
89          for data, target in data_loader:
90              data, target = Variable(data), Variable(target)
91              y_hat = model(data)
```

```
92              test_loss += criterion(y_hat, target)
93              pred = torch.max(y_hat.data, 1)[1]
94              correct += pred.eq(target.data.view_as(pred)).cpu().sum()
95          test_loss /= len(data_loader.dataset)
96          print('\测试nTestSet: Average_loss:{0:.4f}, Accuracy: {1}/{2}
    ({3:.4f})\n'.format(test_loss,
97              correct, len(data_loader.dataset),
98              100. * correct / len(data_loader.dataset))
99          )
100
101         return 1.0 * correct / len(data_loader.dataset)
```

Listing 52: 多层感知器（MLP）训练和预测

代码解读如下所示：

- 第 2、3 行：定义运行状态常量，分别为训练和预测状态；
- 第 15 行：定义构造函数；
- 第 17 行：定义模型参数的保存文件；
- 第 18 行：定义优化器的保存文件，如果我们停止训练过程，再继续训练过程时，需要载入优化器的状态；
- 第 10 行：定义运行方法，参数为：run_mode 的取值为训练和预测状态，缺省为训练状态；continue_train 是否为继续训练，如果是继续训练需要载入模型参数和优化器状态，缺省为否；
- 第 11 行：加载 MNIST 数据集，生成训练数据集加载器、验证数据集加载器、测试数据集加载器；
- 第 12 行：创建模型；
- 第 13 行：定义代价函数为交叉熵函数；
- 第 14 行：优化器采用随机梯度下降算法；
- 第 16~20 行：如果运行模式为训练模式，执行 17~20 行代码；
- 第 17~19 行：从上次运行保存的参数文件中，加载模型参数和优化器状态；
- 第 20 行：调用本类的训练方法（在后面详细讲解）；
- 第 21~38 行：当运行模式为预测模式时，执行 22~38；
- 第 22、23 行：载入模型参数；
- 第 25~27 行：取测试数据集中取出第 1 个批次；
- 第 28 行：指定批次中要选取样本的索引号；
- 第 29 行：取出该样本；
- 第 30 行：由一网络需要的样本形状为 $[1,1,28,28]$，而我们取到的样本为 $[1,28,28]$，所以需要在最前面增加一维；
- 第 31 行：取出正确结果；
- 第 32 行：调用本类 predict 方法求出网络的输出值；
  - 第 40 行：定义预测试法，model 为模型实例，X 为样本，形状为 $[1,1,28,28]$，分别为批次内索引号、颜色通道号、宽、高；

  - 第 41 行：调用前向传播过程求出网络输出层输出；
  - 第 42 行：求出最大值所在的索引号并返回；
- 第 33、34 行：求出预测值，并打印正确值和预测值；
- 第 36 行：取出样本的 28×28 的图像数据；
- 第 37、38 行：绘制样本图像

下面我们来看训练方法：

- 第 44 行：定义训练方法，参数：train_loader、validate_loader、test_loader 分别为训练数据集加载器、验证数据集加载器、测试数据集加载器，model 为模型实例，criterion 为指定的代价函数实例，optimizer 为指定的优化器实例；
- 第 45 行：在验证数据集上取得的最佳精度；
- 第 46 行：在验证数据集上取得的精度提高 0.1% 才算是显著提高；
- 第 47 行：在验证数据集上取得的精度没有提高的训练遍数，训练一遍就是学习完整个训练数据集一次；
- 第 48 行：允许验证数据集上的精度没有显著改进，最大允许训练遍数；
- 第 50∼63 行：循环指定训练遍数；
- 第 51 行：调用本类 train_batch 方法（在后面详细讲解），以迷你批次为单位，训练一遍训练数据集；
- 第 52 行：调用本类 evaluate 方法（在后面详细讲解），求出当前在验证数据集上的精度；
- 第 54∼58 行：如果当前精度有显著改善执行 55∼58 行；
- 第 55 行：更新在验证数据集上所取得的最佳精度；
- 第 56 行：将精度没有改善的训练遍数清零；
- 第 57 行：保存模型参数；
- 第 58 行：保存优化器状态；
- 第 59、60 行：若在验证数据集上的精度没有显著改善，则精度没有明显改善训练遍数加 1；
- 第 61∼63 行：如果精度没有显著改善的训练遍数大于最大允许次数，则退出训练过程，这就是著名的 Early Stopping 算法；
- 第 64、65 行：求出并打印在测试数据集上的精度；
- 第 66 行：保存模型参数；
- 第 67 行：保存优化器状态；
- 第 69 行：定义以批次为单位的训练方法，参数：epoch 为当前训练遍数，model 模型实例，criterion 为指定的代价函数实例，optimizer 为指定的优化器实例，train_loader、validate_loader 分别为训练数据集加载器、验证数据集加载器；
- 第 70 行：调用模型基类的训练方法；
- 第 71∼83 行：依次从训练数据集中取出一个迷你批次，执行 72 ∼ 83 行；
- 第 72 行：以 Variable 格式取出样本 data 和标签 target；
- 第 73 行：清空之前的微分缓存（否则会累加）；

- 第 74 行：调用模型的前向传播过程求出网络输出层的输出；
- 第 75 行：计算代价函数的值；
- 第 76 行：调用 PyTorch 自动微分求出模型参数的微分值；
- 第 77 行：调用优化器来更新模型参数；
- 第 78~83 行：当批次内索引号为 10 的倍数时，打印训练进展情况；

我们接下来看 evaluate 方法：

- 第 85 行：定义 evaluate 方法，该方法用于求在指定数据集上的模型精度，参数：model 模型实例，criterion 为指定的代价函数实例，optimizer 为指的优化器实例，data_loader 为指定数据集的加载器；
- 第 86 行：准备模型；
- 第 87 行：代价函数值；
- 第 88 行：正确分类样本个数；
- 第 89~94 行：依次从数据集中取出迷你批次，执行 90~94 行；
- 第 90 行：以 Variable 格式取出样本 data 和标签 target；
- 第 91 行：调用模型的前向传播算法求出网络输出层输出；
- 第 92 行：计算代价函数的值；
- 第 93 行：利用 torch.max 求出网络输出值按行计算每行最大值所在的下标值,torch.max(x,1) 代表为按行，其返回值为 (val,idx)，我们这里取下标值；
- 第 94 行：将第 93 行求出的计算值与正确值进行比较（相同为 1，不同为 0）然后统计出不为零的元素个数，即为在本迷你批次中正确的样本数；
- 第 95 行：求出在指定数据集上的代价函数值；
- 第 96~99 行：打印评估结果；
- 第 101 行：用正确分类的样本数除以总的样本数作为在指定数据集上的精度并返回；

运行模式为训练模式时程序输出如下所示：

Figure 78: 训练模式输出

```
train epoch:4 [49280/50000    (98.53)]    loss:0.061138
train epoch:4 [49600/50000    (99.17)]    loss:0.012534
train epoch:4 [49920/50000    (99.81)]    loss:0.003757

测试TestSet: Average_loss:0.0002, Accuracy: 49904/50000 (99.8080)

验证精度：0.9980800151824951; 最佳精度：0.9976599812507629; 累积次数：0

测试TestSet: Average_loss:0.0070, Accuracy: 9806/10000 (98.0600)

最终精度：0.9805999994277954
```

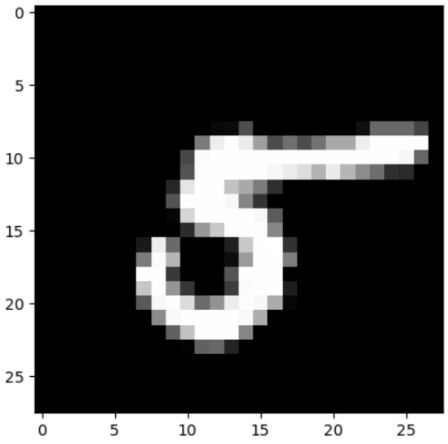
运行模式为预测模式时程序输出如下所示：

Figure 79: 预测模式输出

```
载入模型参数并进行预测
正确值:5; 预测值:5
```

所预测的样本图像为：

Figure 80: 预测样本图像



5.4 PyTorch 核心原理

在上一节中，我们看到使用 PyTorch 只有原来五分之一左右的代码，就实现了多层感知器模型，体验了 PyTorch 的强大功能。但是如果我们拿 PyTorch 来做研究的话，PyTorch 的封装太好了，使我们很难实现各种复杂的网络结构和算法。在本节中，我们将介绍 PyTorch 中各种定制方法。在本质上来说，我们所需要功能在 torch.nn 中都已经具备了，我们只需要调用即可，但是正如本节所示，这里面的模块，都是可以替换的，在实际应用中，我们可以替换其中的一些模块，来满足我们的特殊要求，同时使用其他缺省模块，来保持代码的简洁的高效。如Howard [2019] 所述，我们将使用 PyTorch 的张量开始，一步一步添加各种高级特性，通过这一过程，大家可以了解 torch.nn 各模块的功能和替换方法。

5.4.1 载入数据集

我们通过 pytorch 来载入 MNIST 数据集，并将数据集转化为张量，如下所示（app/pytorch/book/chp004/e3/mlp_mnist_app.py）：

```
class MlpMnistApp(object):
    DATA_PATH = Path("data")
    PATH = DATA_PATH / "mnist"

    def __init__(self):
        self.name = ''

    def run(self):
        X_train, y_train, X_validate, y_validate, X_test, y_test = self.
    load_dataset()

    def load_dataset(self):
        MlpMnistApp.PATH.mkdir(parents=True, exist_ok=True)
        URL = "http://deeplearning.net/data/mnist/"
        FILENAME = "mnist.pkl.gz"
```

```
15
16          if not (MlpMnistApp.PATH / FILENAME).exists():
17                  content = requests.get(URL + FILENAME).content
18                  (MlpMnistApp.PATH / FILENAME).open("wb").write(content)
19          with gzip.open((MlpMnistApp.PATH / FILENAME).as_posix(), "rb") as
     f:
20              ((X_train, y_train), (X_valid, y_valid), (X_test, y_test)) =
    pickle.load(f, encoding="latin-1")
21          plt.imshow(X_train[0].reshape((28, 28)), cmap="gray")
22          plt.show()
23          print('x_train:{0}; y_train:{1}'.format(X_train.shape, y_train.
    shape))
24          return map(
25              torch.tensor, (X_train, y_train, X_valid, y_valid, X_test,
    y_test)
26          )
```

Listing 53: 载入数据集

代码解读如下所示：

- 第 11 行：定义载入 MNIST 数据集方法；
- 第 12 行：如果 data/mnist 目录不存在，则创建该目录；
- 第 13 行：定义下载 URL；
- 第 14 行：定义下载文件；
- 第 16~18 行：如果不存在数据文件，则下载并保存该文件；
- 第 19 行：以二进制只读方式打开该文件；
- 第 20 行：从文件中读出 numpy 数组形式的训练数据集、验证数据集、测试数据集；
- 第 21、22 行：将训练样本集中的第一个样本转化为 28×28 的黑白图像并显示；
- 第 24~26 行：将训练数据集、验证数据集、测试数据集由 Numpy 数组形式转换为 PyTorch 张量形式并返回；

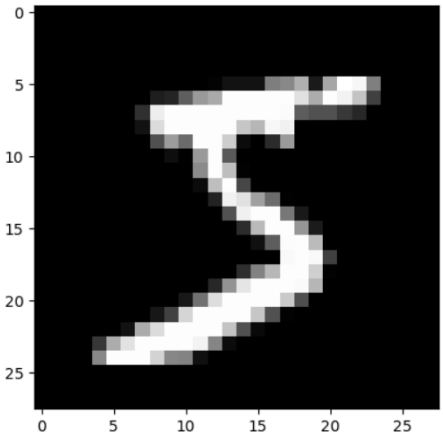
运行结果为：

Figure 81: 载入 MNIST 数据集运行结果

```
training dataset: data:torch.Size([50000, 784]); target:torch.Size([50000])
validate dataset: data:torch.Size([10000, 784]); target:torch.Size([10000])
test dataset: data:torch.Size([10000, 784]); target:torch.Size([10000])
```

样本图像为：

Figure 82: MNIST 数据集样本图像



5.4.2 基本模型类

下面我们来定义模型类，如下所示（app/pytorch/book/chp004/e3/mlp_mnist_model.py）：

```
1  #
2  import math
3  import torch
4
5  class MlpMnistModel(object):
6      def __init__(self):
7          self.name = ''
8          self.weights = torch.randn(784, 10) / math.sqrt(784)
9          self.weights.requires_grad_()
10         self.bias = torch.zeros(10, requires_grad=True)
11
12     def log_softmax(self, x):
13         return x - x.exp().sum(-1).log().unsqueeze(-1)
14
15     def forward(self, xb):
16         return self.log_softmax(xb @ self.weights + self.bias)
```

Listing 54: 定义最简模型类

代码解读如下所示：

- 第 8 行：采用均值为 0 方差为 1 的标准正态分布生成一个形状为 $R^{784\times10}$ 的张量，将该张量除以 $\sqrt{784}$（采用 broadcast 方式进行）；
- 第 9 行：设置该张量需要参与微分计算；
- 第 10 行：定义偏置值为形状为 R^{10} 的元素全为 0 的张量；
- 第 12 行：定义激活函数，这里的激活函数与我们前面讲述的不同，我们重点介绍一下：假设我们输入为：

$$\boldsymbol{x} = \begin{bmatrix} \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \end{bmatrix} \\ \begin{bmatrix} 6.0 & 7.0 & 8.0 & 9.0 & 10.0 \end{bmatrix} \end{bmatrix} \tag{246}$$

先对其求指数运算得到：

$$\boldsymbol{x} = \begin{bmatrix}\begin{bmatrix} 2.7183e+00 & 7.3891e+00 & 2.0086e+01 & 5.4598e+01 & 1.4841e+02 \\ 4.0343e+02 & 1.0966e+03 & 2.9810e+03 & 8.1031e+03 & 2.2026e+04 \end{bmatrix}\end{bmatrix} \tag{247}$$

由 sum(-1) 的求和得到：

$$\boldsymbol{x} = \begin{bmatrix} 233.2042 & 34610.5703 \end{bmatrix} \tag{248}$$

再对其取对数：

$$\boldsymbol{x} = \begin{bmatrix} 5.4519 & 10.4519 \end{bmatrix} \tag{249}$$

调用 unsqueeze 方法添加维数：先对其求指数运算得到：

$$\boldsymbol{x} = \begin{bmatrix}\begin{bmatrix} 5.4519 \end{bmatrix} \\ \begin{bmatrix} 10.4519 \end{bmatrix}\end{bmatrix} \tag{250}$$

最后用 $\boldsymbol{x}$ 减去此值：

$$\boldsymbol{x} = \begin{bmatrix}\begin{bmatrix} -4.4519 & -3.4519 & -2.4519 & -1.4519 & -0.4519 \\ -4.4519 & -3.4519 & -2.4519 & -1.4519 & -0.4519 \end{bmatrix}\end{bmatrix} \tag{251}$$

- 第 13 行：返回所计算的值；
- 第 15 行：定义前向传播方法；
- 第 16 行：输出值 $\hat{y} = \boldsymbol{w}^T \cdot \boldsymbol{x} + b$，式中 @ 为矩阵相乘的意思；

下面我们来看怎样调用这个前向传播过程得到网络的输出：

```
1     def run(self):
2         X_train, y_train, X_validate, y_validate, \
3                     X_test, y_test = self.load_dataset()
4         model = MlpMnistModel()
5         batch_size = 64
6         X0 = X_train[0:batch_size]
7         preds = model.forward(X0)
8         y_hat = torch.argmax(preds, dim=1)
9         print('preds[0]:{0}; shape:{1}'.format(preds[0], preds.shape))
10        print(y_hat)
```

Listing 55: 求网络输出值

代码解读如下所示：

- 第 2、3 行:；
- 第 4 行：生成模型类；
- 第 5 行：定义迷你批次的大小为 64；
- 第 6 行：取出一个迷你批次；
- 第 7 行：调用前向传播过程求出网络输出；
- 第 8 行：求出输出中每行最大元素的下标作为分类结果；

运行结果如下所示：

Figure 83: 求网络输出运行结果

```
preds[0]:tensor([-1.8453, -2.0666, -2.8819, -2.3487, -1.8263, -2.9501, -2.4517, -2.1968,
        -2.2392, -3.0642], grad_fn=<SelectBackward>); shape:torch.Size([64, 10])
tensor([4, 0, 4, 7, 0, 4, 0, 0, 0, 6, 0, 7, 0, 4, 0, 0, 8, 1, 1, 0, 0, 0, 9, 7,
        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 6, 0, 3,
        0, 0, 8, 0, 0, 0, 0, 0, 7, 0, 0, 7, 9, 0, 0, 0])
```

接下来我们来看代价函数定义，我们假设使用负对数似然函数作为代价函数，代码如下所示：

```
1  class MlpMnistModel(object):
2  ......
3      def nll(self, input, target):
4          return -input[range(target.shape[0]), target].mean()
5
6      def accuracy(self, out, yb):
7          preds = torch.argmax(out, dim=1)
8          return (preds == yb).float().mean()
9  # test code in mlp_mnist_app.py
10     def run(self):
11         X_train, y_train, X_validate, y_validate, \
12                     X_test, y_test = self.load_dataset()
13         model = MlpMnistModel()
14         batch_size = 64
15         X0 = X_train[0:batch_size]
16         y0 = y_train[0:batch_size]
17         preds = model.forward(X0)
18         y_hat = torch.argmax(preds, dim=1)
19         loss = model.nll(preds, y0)
20         accuracy = model.accuracy(preds, y0)
21         print('loss={0}; accuracy={1}'.format(loss, accuracy))
```

Listing 56: 负对数似然函数

代码解读如下所示：

- 第 3 行：定义负对数似然函数为代价函数，参数为网络的原始输出；
- 第 4 行：对网络输出进行切片操作，取所有行，每行保留的元素由 target 对应位置的值来决定，例如网络原始输出为：

$$\boldsymbol{x} = \begin{bmatrix} \begin{bmatrix} -0.2 & -0.3 & -0.4 \\ -1.1 & -1.2 & -1.3 \end{bmatrix} \end{bmatrix} \tag{252}$$

正确值为：

$$\boldsymbol{x} = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{253}$$

则其结果为：

$$\boldsymbol{x} = \begin{bmatrix} -0.2 & -1.2 \end{bmatrix} \tag{254}$$

- 第 6 行：定义求精度方法，参数：out 为网络的原始输出，yb 为正确值，对于比例中迷你批次大小为 64 时：$out \in R^{64\times 10}$ 其中最大的元素所在的下标代表类别，而 $yb \in R^{64}$ 用 0～9 的数字代表类别；
- 第 11 行：对网络原始输出求出每行最大元素的下标组成新的数组；
- 第 12 行：逐元素与正确值数组进行比较，相等为 1，不相等为 0，然后求整个结果数组的平均值即为正确率；
- 第 11～15 行：载入 MNIST 数据集并载入第一个迷你批次样本；
- 第 16 行：取出第一个迷你批次的正确值；
- 第 17 行：调用前向传播过程求出网络的原始输出；
- 第 19 行：调用代价函数求出代价函数的值；
- 第 20 行：求出本迷你批次上的精度（正确率）；
- 第 19 行：打印代价函数的值和精度；

运行结果如下所示：

Figure 84: 代价函数值

```
loss=2.3955512046813965; accuracy=0.109375
```

由上图看出，我们的精度为 10% 左右，这是合理的，因为我们用随机数初始化的网络参数，我们只能得到自然概率的精度。

5.4.3 使用 DataLoader 重构

在上面的代码中，我们手工将数据集载入并进行迷你批次划分，我们可以将这部分工作用 PyTorch 的 DataLoader 来完成这部分工作。我们首先来改造数据集载入函数：

```
1    def load_dataset(self):
2        MlpMnistApp.PATH.mkdir(parents=True, exist_ok=True)
3        URL = "http://deeplearning.net/data/mnist/"
4        FILENAME = "mnist.pkl.gz"
5
6        if not (MlpMnistApp.PATH / FILENAME).exists():
7                content = requests.get(URL + FILENAME).content
8                (MlpMnistApp.PATH / FILENAME).open("wb")\
9                            .write(content)
10       with gzip.open((MlpMnistApp.PATH / FILENAME).as_posix(), "rb") as
     f:
11           ((X_train, y_train), (X_valid, y_valid),
12                       (X_test, y_test)) = pickle.load(f, encoding="
    latin-1")
13       X_train, y_train, X_validate, y_validate, \
14                   X_test, y_test = map(
15           torch.tensor, (X_train, y_train, X_valid, y_valid,
16           X_test, y_test)
17       )
18       batch_size = 64
```

```
19          train_ds = TensorDataset(X_train, y_train)
20          train_loader = DataLoader(train_ds, batch_size=batch_size,
21                      shuffle=True)
22          validate_ds = TensorDataset(X_validate, y_validate)
23          validate_loader = DataLoader(validate_ds, batch_size=batch_size,
24                      shuffle=True)
25          test_ds = TensorDataset(X_test, y_test)
26          test_loader = DataLoader(test_ds, batch_size=batch_size,
27                      shuffle=False)
28          return train_loader, validate_loader, test_loader
```

Listing 57: 重构的 MNIST 数据集载入方法

程序前半部分没有变化，只是后面有些改变，我们这里只讲后面改变的代码：

- 第 18 行：定义迷你批次大小；
- 第 19 行：生成训练数据集的 TensorDataset 对象；
- 第 20、21 行：生成训练数据集的 DataLoader 对象，这里指定迷你批次大小，同时指定需要将样本顺充打乱重排；
- 第 22 行：生成验证数据集的 TensorDataset 对象；
- 第 23、24 行：生成证验数据集的 DataLoader 对象，这里指定迷你批次大小，同时指定需要将样本顺充打乱重排；
- 第 25 行：生成测试数据集的 TensorDataset 对象；
- 第 26、27 行：生成测试数据集的 DataLoader 对象，这里指定迷你批次大小，测试数据集不需要打乱重排；
- 第 28 行：返回训练数据集、验证数据集、测试数据集的 DataLoader 对象；

我们还需要修改训练函数中获取迷你批次的方法：

```
1       def run(self):
2           train_loader, validate_loader, test_loader \
3                       = self.load_dataset()
4           model = MlpMnistModel()
5           learning_rate = 0.1
6           epochs = 5
7           loss_func = model.nll
8           for epoch in range(epochs):
9               for xb,yb in train_loader:
10                  pred = model.forward(xb)
11                  loss = loss_func(pred, yb)
12                  accuracy = model.accuracy(pred, yb)
13                  loss.backward()
14                  with torch.no_grad():
15                      model.weights -= model.weights.grad * learning_rate
16                      model.bias -= model.bias.grad * learning_rate
17                      model.weights.grad.zero_()
18                      model.bias.grad.zero_()
19                  #print('    {0}: loss={1}; accuracy{2}'.format(i, loss,
    accuracy))
```

```
20              print('{0}: loss={1}; accuracy={2}'.format(epoch, loss,
    accuracy))
```

Listing 58: 重构训练方法

这里可以直接从 train_loader 中获取迷你批次，而不用我们手动计算索引载入了。

5.4.4 抽象出代价函数

接下来我们把代价函数抽像出来，使我们可以在实际应用中切换不同的代价函数，我们定义如下代价函数（app/pytorch/book/chp004/e3/aqp_loss.py）：

```
1  import torch.nn.functional as F
2
3  class AqpLoss(object):
4      @staticmethod
5      def nll(output, target):
6          y_hat = F.softmax(output, dim=1)
7          return -y_hat[range(target.shape[0]), target].mean()
```

Listing 59: 重构训练方法

我们在训练函数中可以指定这个函数为代价函数：

```
1      def run(self):
2          train_loader, validate_loader, test_loader \
3                      = self.load_dataset()
4          model = MlpMnistModel()
5          learning_rate = 0.1
6          epochs = 5
7          #criterion = F.cross_entropy
8          criterion = AqpLoss.nll
9          for epoch in range(epochs):
10             for xb,yb in train_loader:
11                 pred = model.forward(xb)
12                 loss = criterion(pred, yb)
13                 accuracy = model.accuracy(pred, yb)
14                 loss.backward()
15                 with torch.no_grad():
16                     model.weights -= model.weights.grad * learning_rate
17                     model.bias -= model.bias.grad * learning_rate
18                     model.weights.grad.zero_()
19                     model.bias.grad.zero_()
20                 #print('    {0}: loss={1}; accuracy{2}'.format(i, loss,
    accuracy))
21             print('{0}: loss={1}; accuracy={2}'.format(epoch, loss,
    accuracy))
```

Listing 60: 重构训练方法

在第 7 行和第 8 行，我们可以根据需要使用内置的交叉熵代价函数，也可以使用我们刚刚定义的负对数似然函数，二者的效果是相同的。
我们使用新的代价函数时，我们就不需要在模型类中使用激活函数了，如下所示：

```
class MlpMnistModel(object):
    def __init__(self):
        self.name = ''
        self.weights = torch.randn(784, 10) / math.sqrt(784)
        self.weights.requires_grad_()
        self.bias = torch.zeros(10, requires_grad=True)

    def forward(self, xb):
        return xb @ self.weights + self.bias
```

Listing 61: 重构训练方法

5.4.5 重构模型类

接下来我们将继承 torch.nn.Module 类，如下所示：

```
class MlpMnistModel(torch.nn.Module):
    def __init__(self):
        super(MlpMnistModel, self).__init__()
        self.name = ''
        self.weights = torch.nn.Parameter(torch.randn(784, 10) / math.
    sqrt(784))
        self.bias = torch.nn.Parameter(torch.zeros(10))

    def forward(self, xb):
        return xb @ self.weights + self.bias

    def accuracy(self, out, yb):
        preds = torch.argmax(out, dim=1)
        return (preds == yb).float().mean()
```

Listing 62: 重构训练方法

代码解读如下所示：

- 第 1 行：继承自 torch.nn.Module；
- 第 3 行：调用基类的构造函数；
- 第 5 行：将连接权值定义为基类的模型参数；
- 第 6 行：将偏置值定义为基类的模型参数；

在训练函数中，我们需要做如下修改：

```
    def run(self):
        train_loader, validate_loader, test_loader \
                    = self.load_dataset()
        model = MlpMnistModel()
        learning_rate = 0.1
        epochs = 5
        criterion = F.cross_entropy
        #criterion = AqpLoss.nll
        for epoch in range(epochs):
```

```
10            for xb,yb in train_loader:
11                pred = model.forward(xb)
12                loss = criterion(pred, yb)
13                accuracy = model.accuracy(pred, yb)
14                loss.backward()
15                with torch.no_grad():
16                    for p in model.parameters():
17                        p -= p.grad * learning_rate
18                    model.zero_grad()
19                #print('    {0}: loss={1}; accuracy{2}'.format(i, loss,
    accuracy))
20            print('{0}: loss={1}; accuracy={2}'.format(epoch, loss,
    accuracy))
```

Listing 63: 重构训练方法 2

主要改动在 15~18 行，我们就不再需要手工对每个参数进行更新和清空微分值了。

5.4.6 抽象出优化器

下面我们来设计优化器，优化器的任务就是根据网络参数的微分，以合适的学习率来更新参数，并在每遍训练时清零网络参数微分。优化器可以根据实际情况，调整学习率，从而实现更快的收敛速度。我们在这里只实现最简单的优化器，即固定的学习率。代码如下所示（app/pytorch/book/chp004/e3/aqp_base_optim.py）：

```
1  #
2  import torch
3
4  class AqpBaseOptim(object):
5      def __init__(self, parameters, lr=0.1):
6          self.name = ''
7          self.params = []
8          for p in parameters:
9              self.params.append(p)
10         self.lr = lr
11
12     def step(self):
13         with torch.no_grad():
14             for p in self.params:
15                 p -= p.grad * self.lr
16
17     def zero_grad(self):
18         for p in self.params:
19                 p.grad.zero_()
```

Listing 64: 最简优化器示例

代码解读如下所示：

- 第 5 行：定义构造函数，参数：parameters 为网络参数，lr 是初始学习率；
- 第 7~9 行：由于 parameters 是 generator 类型，只能遍历一次，因此我们需要对其进行缓存，保存在 self.params；

- 第 12~15 行：根据网络参数微分更新参数；
- 第 17~19 行：清空网络参数的微分；

调用此优化器的代码如下所示：

```
1     def run(self):
2         train_loader, validate_loader, test_loader \
3                     = self.load_dataset()
4         model = MlpMnistModel()
5         learning_rate = 0.1
6         epochs = 5
7         criterion = F.cross_entropy
8         #criterion = AqpLoss.nll
9         #optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate
      )
10        optimizer = AqpBaseOptim(model.parameters(), lr=learning_rate)
11        for epoch in range(epochs):
12            for xb,yb in train_loader:
13                pred = model.forward(xb)
14                loss = criterion(pred, yb)
15                accuracy = model.accuracy(pred, yb)
16                loss.backward()
17                optimizer.step()
18                optimizer.zero_grad()
19                #print('    {0}: loss={1}; accuracy{2}'.format(i, loss,
      accuracy))
20            print('{0}: loss={1}; accuracy={2}'.format(epoch, loss,
      accuracy))
```

Listing 65: 重构训练方法 4

在上面的程序中，第 9 行是调用系统的优化器，第 10 行是调用我们自己的优化器，我们可以看到，这两个方法基本差不多。

5.4.7 验证数据集

在前面的程序中，我们实际上没使用验证样本集，在实际应用中，我们需要在每遍训练完训练数据集之后，需要在验证样本集上求出精度和代价函数值，然后就可以利用上节介绍的 Early Stopping 算法，避免过拟合（Overfitting）了。
我们首先需要改造验证数据集载入过程（app/pytorch/book/chp004/e3/mlp_mnist_app.py.load_dataset）：

```
1     def load_dataset(self):
2         ......
3         batch_size = 64
4         train_ds = TensorDataset(X_train, y_train)
5         train_loader = DataLoader(train_ds, batch_size=batch_size,
6                     shuffle=True)
7         validate_ds = TensorDataset(X_validate, y_validate)
8         validate_loader = DataLoader(validate_ds, batch_size=batch_size
      *2,
```

```
                    shuffle=False)
        test_ds = TensorDataset(X_test, y_test)
        test_loader = DataLoader(test_ds, batch_size=batch_size*2,
                    shuffle=False)
        ......
```

Listing 66: 重构验证样本集载入方法

由于验证样本集不用参加训练过程，因此就不需要为其进行重新随机排序，同时验证样本集不需要求反向传播过程，因此其内存的需求量只为训练过程的一半，因此迷你批次大小应该为训练数据集迷你批次的 2 倍，对于测试数据集也需要按同样方法处理。

我们在每遍训练结束时，计算在验证样本集上的代价函数值，我们可以用这个值来做 Early Stopping 算法，同时在训练结束之后，我们可以计算在测试数据集上的精度，程序如下所示（app/pytorch/book/chp004/e3/mlp_mnist_app.py.run）：

```
    def run(self):
        train_loader, validate_loader, test_loader \
                    = self.load_dataset()
        model = MlpMnistModel()
        learning_rate = 0.1
        epochs = 5
        criterion = F.cross_entropy
        #criterion = AqpLoss.nll
        #optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate
    )
        optimizer = AqpBaseOptim(model.parameters(), lr=learning_rate)
        for epoch in range(epochs):
            for xb,yb in train_loader:
                pred = model.forward(xb)
                loss = criterion(pred, yb)
                accuracy = model.accuracy(pred, yb)
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()
            model.eval()
            with torch.no_grad():
                valid_loss = sum(criterion(model(xb), yb) for xb, yb in
    validate_loader)
            print('{0}: loss={1}'.format(epoch, valid_loss / len(
    validate_loader)))
        accuracy = self.evaluate(model, test_loader)
        print('accuracy in test dataset: {0}'.format(accuracy))

    def evaluate(self, model, test_loader):
        accuracy = sum(model.accuracy(model(xb), yb) for xb, yb in
    test_loader)
        return accuracy / len(test_loader)
```

Listing 67: 重构验证样本集载入方法

程序大部分代码我们之前已经讲解过，我们这里只讲解变化的代码：

- 第 20 行：因为求验证数据集上的代价函数，不属于反向传播算法的一部分，因此需要将其排除在反向传播之外；
- 第 21 行：计算在整个验证数据集上代价函数之和，方法为：从验证数据集依次取出迷你批次，然后计算在此迷你批次的代价函数值，形成一个列表，最后用 sum 求和；
- 第 22 行：将验证数据集上代价函数之和除以验证样本集批次数之和可以得到代价函数的平均值，我们打印每遍训练结束后的代价函数平均值；
- 第 23 行：求测试数据集上的精度；
  - 第 26 行：定义求精度方法；
  - 第 27 行：依次取出测试数据集上的迷你批次，求出迷你批次上的精度，并形成一个列表，最后用 sum 函数来求和；
  - 第 28 行：将精度之和除以测试数据集上的迷你批次数，求出精度的平均值并返回；
- 第 24 行：打印测试数据集上的精度

5.4.8 GPU 加速

到目前为止，我们已经讲解完成了一个完整的深度学习算法框架，大家如果想研究新的激活函数，新的优化器算法，新的网络结构，都可以找到替换 PyTorch 缺省实现方式，进行算法研究。这里我们讲解做深度学习研究中非常重要的一点，就是 GPU 加速。
加入 GPU 加速后训练方法如下所示：

```
1    def run(self):
2        print('has GPU:{0}'.format(torch.cuda.is_available()))
3        dev = torch.device("cuda") if torch.cuda\
4                    .is_available() else torch\
5                    .device("cpu")
6        print('It will run on {0}!'.format(dev))
7        train_loader, validate_loader, test_loader \
8                    = self.load_dataset()
9        model = MlpMnistModel()
10       model = model.to(dev)
11       learning_rate = 0.1
12       epochs = 5
13       criterion = F.cross_entropy
14       #criterion = AqpLoss.nll
15       #optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate
     )
16       optimizer = AqpBaseOptim(model.parameters(), lr=learning_rate)
17       for epoch in range(epochs):
18           for xb,yb in train_loader:
19               xb = xb.to(dev)
20               yb = yb.to(dev)
21               pred = model.forward(xb)
22               loss = criterion(pred, yb)
23               #accuracy = model.accuracy(pred, yb)
24               loss.backward()
25               optimizer.step()
```

```
26                 optimizer.zero_grad()
27             model.eval()
28             with torch.no_grad():
29                 valid_loss = sum(criterion(model(xb.to(dev)), yb.to(dev))
    for xb, yb in validate_loader)
30             print('{0}: loss={1}'.format(epoch, valid_loss / len(
    validate_loader)))
31         accuracy = self.evaluate(dev, model, test_loader)
32         print('accuracy in test dataset: {0}'.format(accuracy))
```

Listing 68: GPU 加速方法

对变化的代码解读如下所示：

- 第 2 行：我们检查计算机中是否有 NVIDIA 系列的 CPU 设备；
- 第 3 行：如果有 GPU 取 GPU，若没有才用 CPU；
- 第 32 行：将模型移到 GPU 中；
- 第 41、42 行：将迷你批次的样本集和标签集移到 GPU 中；
- 第 29 行：计算验证数据集上的代价函数值也需要将迷你批次的样本集和标签集移到 GPU 中；

5.5 总结

在本章中，我们系统讲解了多层感知器模型的数学原理，并以 Numpy 为例，纯手工完成了一个特别初级的多层感知器模型，接着我们用标准的 PyTorch 方式，实现了多层感知器模型，大家可以看到 PyTorch 框架，极大的简化了我们的代码编写过程。但是如果我们要研究深度学习算法，PyTorch 这种封装方式就不是很方便了。所以我们在第三部分，讲解了怎样将 PyTorch 缺省实现，替换为我们自己的实现的方法，并在最后讲解了怎样将代码转到 GPU 上运行的方法，可以极大加快训练学习过程。但是需要注意的是，将数据在 CPU 和 GPU 之间移动是非常耗费时间的，需要仔细权衡后再确实实现方法。

第 6 章卷积神经网络

Abstract

在本章中我们将讨论 PyTorch 的基础概念张量（Tensor），包括创建、基本属性、基本操作，同时会稍微涉及一下底层原理，是后续学习的基础。

6 卷积神经网络概述

第 7 章递归神经网络

Abstract

在本章中我们将讨论 PyTorch 的基础概念张量（Tensor），包括创建、基本属性、基本操作，同时会稍微涉及一下底层原理，是后续学习的基础。

7 递归神经网络概述

References

闫涛，周琦. 深度学习算法实践 ---基于 *Theano* 和 *TensorFlow*. 电子工业出版社, 北京市丰台区, 2017.

最老程序员. Pytorch 学习笔记 1---张量. *blog.csdn.net*, yt7589, 2019. URL https://blog.csdn.net/Yt7589/article/details/103348336.

Luca Antiga Eli Stevens. *Deep Learning with PyTorch*. Manning, www.manning.com, 2019.

facebook. pytorch tutorial tensor. *pytorch.org*, 1603.02754, 2019. URL https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html.

Jeremy Howard. What is torch.nn really? *pytorch.org*, tutorials, 2019. URL https://pytorch.org/tutorials/beginner/nn_tutorial.html.