深度学习框架与网络结构设计

荣耀学院

2022 年 07 月

第一版序言

目 录

第一章　一个最简单的极值问题求解

用 Pytorch 求解一个最简单的极值问题

$$y = \frac{1}{2}(x-2)^2$$

```
import torch
import numpy as np

# x是一个张量Tensor，只有一个元素
x = torch.tensor(np.random.normal(0, 0.01, (1, 1)), dtype=torch.float32)
# 需要计算x的梯度，以便对x进行优化
x.requires_grad_(requires_grad=True)

# 学习速率
eta = 0.4
for i in range(15):
    print('x=', x.data.item())
    y = (x-2)**2/2
    # 反向计算梯度
    y.backward()
    # 根据梯度，对x进行优化
    x.data -= eta*x.grad
    # 对x的梯度进行清零，以便进入下一轮优化
    x.grad.data.zero_()
```

运行结果如下：

```
x= 0.007588282693177462
x= 0.8045529723167419
x= 1.2827317714691162
x= 1.5696390867233276
x= 1.7417834997177124
x= 1.8450701236724854
x= 1.9070420265197754
x= 1.9442251920700073
x= 1.9665350914001465
x= 1.9799211025238037
```

```
x= 1.987952709197998
x= 1.9927716255187988
x= 1.9956629276275635
x= 1.997397780418396
x= 1.9984387159347534
```

代码很少。只有一个地方需要解释一下：为什么 “y = (x-2)**2/2” 放在循环里？这跟常规用法似乎不一样。注意，在 Pytorch，凡是看到表达式，大脑里跳出来的都应该是 “计算图”：前向计算结果，反向计算梯度。熟悉自动微分，就知道 “y = (x-2)**2/2” 是前向计算，计算图的前一半，“y.backward()” 反向计算梯度，计算图的后一半。每次循环，计算图都要前向计算一次，再反向计算一次。所以 “y = (x-2)**2/2” 必须放在循环里。

第二章　向量表示的极值问题求解

假设有一个向量 $[x_0, x_1]$，求如下函数的极值：

$$y = x_0^2 + x_1^2 + 4x_0 + 5x_1 + 3$$

```
import torch
import numpy as np

x = torch.tensor(np.random.normal(0, 0.01, (1, 2)), dtype=torch.float32)
x.requires_grad_(requires_grad=True)

eta = 0.4
for i in range(10):
    print('x=', x)
    y = torch.mm(x, x.t())+4*x[0,0]+5*x[0,1]+3
    y.backward()
    x.data -= eta*x.grad
    x.grad.data.zero_()
```

运行结果如下：

```
x= tensor([[-0.0040,  0.0005]], requires_grad=True)
x= tensor([[-1.6008, -1.9999]], requires_grad=True)
x= tensor([[-1.9202, -2.4000]], requires_grad=True)
x= tensor([[-1.9840, -2.4800]], requires_grad=True)
x= tensor([[-1.9968, -2.4960]], requires_grad=True)
x= tensor([[-1.9994, -2.4992]], requires_grad=True)
x= tensor([[-1.9999, -2.4998]], requires_grad=True)
x= tensor([[-2.0000, -2.5000]], requires_grad=True)
x= tensor([[-2.0000, -2.5000]], requires_grad=True)
x= tensor([[-2.0000, -2.5000]], requires_grad=True)
```

由此可知，在 $[x_0, x_1] = [-2, -2.5]$ 的时候，函数取得极小值。

其中，torch.mm(x, x.t()) 是矩阵相乘。x 是一个 1×2 的向量，x.t() 是它的转置，是一个 2×1 的向量，因此它们的乘积是一个标量。

注意：不要把向量、矩阵、张量看得很神秘，没有什么神迷的，它们只是数据的存放方式而已。分析问题的时候，把它们视为一堆有排列规则的标量就行了，无论它们形式如何奇怪，本质仍然是标量计算。优化问题目标函数的值一定是标量，比如多输出模型会把所有输出的误差平方和累加起来作为目标函数。因此求解梯度必然是标量对标量、向量、矩阵、张量的求导，百分百不会出现非标量对它们求导，比如绝不可能出现向量对矩阵求导。

深度学习模型无论多复杂，也无非是一个函数，参数多了一些，但求解方式跟这两章给出的例子在本质上是完全一样的。

第三章　多个训练样本的线性回归问题求解

线性回归模型，$y = \boldsymbol{w}\boldsymbol{x} + b$。

y 必然是标量。b 也是标量。$\boldsymbol{w}$ 和 $\boldsymbol{x}$ 是向量，为了保证它们相乘的结果是标量，$\boldsymbol{w}$ 必然是 $1 \times d$，$\boldsymbol{x}$ 必然是 $d \times 1$，d 是维数。d 是不确定的，如果 $\boldsymbol{x}$ 是二维数据，$d = 2$，如果是 $\boldsymbol{x}$ 是三维数据，$d = 3$。

用最简单的情况演示求解：假如是二元线性回归，$d = 2$，此时线性回归问题有三个未知数 ($\boldsymbol{w}$ 有两个未知数，b 是一个未知数)，有三个样本即可求得精确解。作为对比，用三个样本拟合 $\boldsymbol{w}$ 和 b。

设定 $\boldsymbol{w}$、b 和 $\boldsymbol{x}$，生成样本数据：

```
import torch

w = torch.tensor([3.1, 4.2], dtype=torch.float32).unsqueeze(dim=0)
b = torch.tensor([0.5], dtype=torch.float32)
x = torch.tensor([[1.1, 4.6, 8.9], [2.3, 5.7, 10.1]], dtype=torch.float32)
y_target = torch.mm(w, x)+b
print('y_target = ', y_target)
```

运行结果：

```
y_target =  tensor([[13.5700, 38.7000, 70.5100]])
```

在演示程序使用这些样本计算 $\boldsymbol{w}$ 和 b:

```
import torch

x = torch.tensor([[1.1, 4.6, 8.9], [2.3, 5.7, 10.1]], dtype=torch.float32)
y_target = torch.tensor([13.5700, 38.7000, 70.5100], dtype=torch.float32)

w = torch.tensor([0.001, 0.003], dtype=torch.float32).unsqueeze(dim=0)
w.requires_grad_(requires_grad=True)
b = torch.tensor([0.005], dtype=torch.float32)
b.requires_grad_(requires_grad=True)

eta = 0.0001
for i in range(30000):
    print('-'*20)
    print('w =', w)
    print('b =', b)
    y = torch.mm(w, x) + b
    print('y =', y)
    loss = (y - y_target) ** 2 / 2
    print('loss = ', loss)
    sum_err = torch.sum(loss)
    print('sum_err = ', sum_err)
    sum_err.backward()
    print('w.grad=', w.grad)
    w.data -= eta*w.grad
    b.data -= eta*b.grad
    w.grad.data.zero_()
    b.grad.data.zero_()
```

运行结果：

```
--------------------
w = tensor([[0.0010, 0.0030]], requires_grad=True)
b = tensor([0.0050], requires_grad=True)
y = tensor([[0.0130, 0.0267, 0.0442]], grad_fn=<AddBackward0>)
loss =  tensor([[  91.8961,  747.8121, 2482.7148]], grad_fn=<DivBackward0>)
sum_err =  tensor(3322.4231, grad_fn=<SumBackward0>)
w.grad= tensor([[-819.9555, -963.3235]])
--------------------
w = tensor([[0.0830, 0.0993]], requires_grad=True)
b = tensor([0.0173], requires_grad=True)
y = tensor([[0.3370, 0.9652, 1.7592]], grad_fn=<AddBackward0>)
loss =  tensor([[  87.5557,  711.9559, 2363.3374]], grad_fn=<DivBackward0>)
sum_err =  tensor(3162.8491, grad_fn=<SumBackward0>)
w.grad= tensor([[-800.0184, -939.9072]])
--------------------
w = tensor([[0.1630, 0.1933]], requires_grad=True)
b = tensor([0.0292], requires_grad=True)
y = tensor([[0.6532, 1.8810, 3.4325]], grad_fn=<AddBackward0>)
loss =  tensor([[  83.4221,  677.8205, 2249.6970]], grad_fn=<DivBackward0>)
sum_err =  tensor(3010.9395, grad_fn=<SumBackward0>)
w.grad= tensor([[-780.5660, -917.0601]])
--------------------
...
--------------------
w = tensor([[3.2567, 4.0443]], requires_grad=True)
b = tensor([0.6755], requires_grad=True)
y = tensor([[13.5597, 38.7087, 70.5072]], grad_fn=<AddBackward0>)
loss =  tensor([[5.2953e-05, 3.7459e-05, 3.8350e-06]], grad_fn=<DivBackward0>)
sum_err =  tensor(9.4248e-05, grad_fn=<SumBackward0>)
w.grad= tensor([[ 0.0038, -0.0023]])
```

计算到最后，$\boldsymbol{w}$ 和 b 比较接近数据生成时的参数，不完全一致，此时 sum_err 已经相当小了，再计算意义不大。

观察结果，有一点要注意：$w.grad$ 和 $b.grad$ 的值，在开始的时候很大，因此 η 的值必须足够小，否则 $\boldsymbol{w}$ 和 b 计算结果不稳定。动态调整 η 是最好的，有兴趣可以改写代码。

有了自动微分，没啥搞不定的问题。

可以试试神经网络了。

第四章　多层感知机

一个神经网络，需要知道的结构参数是这样的：一个输入层，若干个隐层，一个输出层，每一层跟其它层之间的连接方式，每一层的激活函数，目标函数。定义这些参数，神经网络就确定了。然后就是优化问题。

这一章先手工定义一个神经网络，再用 Pytorch 的辅助更快速地定义这个神经网络。

手工定义神经网络并训练:

```
import torch
import numpy as np

def relu(x):
    return torch.max(input=x, other=torch.tensor(0.0))

def net(x, w1, b1, w2, b2):
    o = relu(torch.matmul(x, w1) + b1)
    return torch.matmul(o, w2) + b2

# 每层神经元数量
n_inputs, n_hiddens, n_outputs = 2, 3, 1

# 各层之间的连接权重和偏差
w1 = torch.tensor(np.random.normal(0, 0.01, (n_inputs, n_hiddens)), dtype=
                                        torch.float)
# 隐层共享一个偏差
b1 = torch.zeros(1, dtype=torch.float)
w2 = torch.tensor(np.random.normal(0, 0.01, (n_hiddens, n_outputs)), dtype=
                                        torch.float)
# 输出层偏差设定可以由n_outputs数量决定，有必要的话也可以设定为共享1个
b2 = torch.zeros(n_outputs, dtype=torch.float)

params = [w1, b1, w2, b2]
for param in params:
    param.requires_grad_(requires_grad=True)

# 样本
x = torch.tensor([2.1, 3.2], dtype=torch.float32).unsqueeze(dim=0)
y_target = torch.tensor([0.7], dtype=torch.float32).unsqueeze(dim=0)

# 学习速率
eta = 0.01

# 训练
for i in range(50):
    print('-'*20)
    print('w1 =', w1)
    print('b1 =', b1)
    y = net(x, w1, b1, w2, b2)
    sum_err = torch.sum((y-y_target)**2/2)
    print('sum_err = ', sum_err)
    sum_err.backward()
    for param in params:
        param.data -= eta*param.grad.data
        param.grad.zero_()
```

输出结果：

```
--------------------
w1 = tensor([[-0.0017, -0.0037, -0.0118],
        [ 0.0041, -0.0064, -0.0009]], requires_grad=True)
b1 = tensor([0.], requires_grad=True)
sum_err =  tensor(0.2450, grad_fn=<SumBackward0>)
--------------------
w1 = tensor([[-0.0018, -0.0037, -0.0118],
        [ 0.0040, -0.0064, -0.0009]], requires_grad=True)
b1 = tensor([-2.9994e-05], requires_grad=True)
sum_err =  tensor(0.2402, grad_fn=<SumBackward0>)
--------------------
...
--------------------
w1 = tensor([[-0.0030, -0.0037, -0.0118],
        [ 0.0021, -0.0064, -0.0009]], requires_grad=True)
b1 = tensor([-0.0006], requires_grad=True)
sum_err =  tensor(0.0915, grad_fn=<SumBackward0>)
```

可以看到输出误差变小了。

使用 Pytorch 的辅助函数定义神经网络，设置两个隐层。代码量简洁多了。

```
import torch
from torch import nn

# 每层的神经元数量
n_inputs, n_hiddens_1,n_hiddens_2, n_outputs = 2, 3, 4, 1
# 定义网络结构：层属性和激活函数都有
net = nn.Sequential(
    nn.Linear(n_inputs, n_hiddens_1),
    nn.ReLU(),
    nn.Linear(n_hiddens_1, n_hiddens_2),
    nn.ReLU(),
    nn.Linear(n_hiddens_2, n_outputs),
)
# 初始化参数
for params in net.parameters():
    torch.nn.init.normal_(params, mean=0, std=0.01)
# 优化器
optimizer = torch.optim.SGD(net.parameters(), lr=0.5)

# 样本
x = torch.tensor([2.1, 3.2], dtype=torch.float32).unsqueeze(dim=0)
y_target = torch.tensor([0.7], dtype=torch.float32).unsqueeze(dim=0)

# 训练
for i in range(5):
    print('-' * 20)
    y = net(x)
    print('y = ', y)
    l = (y - y_target) ** 2 / 2
    l.backward()
    print('l = ', l)
    optimizer.step()
    optimizer.zero_grad()
```

输出结果：

```
--------------------
y =  tensor([[0.0003]], grad_fn=<AddmmBackward0>)
l =  tensor([[0.2448]], grad_fn=<DivBackward0>)
--------------------
y =  tensor([[0.3502]], grad_fn=<AddmmBackward0>)
l =  tensor([[0.0612]], grad_fn=<DivBackward0>)
--------------------
...
--------------------
y =  tensor([[0.6563]], grad_fn=<AddmmBackward0>)
l =  tensor([[0.0010]], grad_fn=<DivBackward0>)
```

第五章　AlexNet：第一个深度卷积神经网络

卷积神经网络是为了解决图像分类问题发明的。

“卷积”，用卷积核从二维图像上提取出各种特征。

LeNet 是第一个卷积神经网络。

“深度”，意思是神经网络的层数多。层数越多，网络性能越好。只有 ReLU 函数能支持大层数，因此“深度卷积神经网络”的激活函数一定只能是 ReLU 函数。

AlexNet 是第一个深度卷积神经网络[1]。

AlexNet 的第 1 层，是卷积层。卷积层的输入是一个图像。

图像由一个个像素组成。像素是有坐标的，假设图像的宽高值是 300×200，那么图像左上角像素的坐标是 $[0, 0]$，右下角像素的坐标是 $[299, 199]$，其它像素坐标比照这两个像素可以得出。如果图像是灰度的，那么只有一个通道 channel，通道又叫图层，灰度图的像素数值一般在 $[0, 255]$ 之间，0 是最暗的，255 是最亮的，光最强。如果图像是彩图，那么通常有三个通道，也可能有四个通道。彩图的格式很多，从实际意义来说，三个通道大多数情况下是 RGB 红绿蓝三色，每一种颜色对应一个通道。

卷积核是一个小窗口。比如第一个卷积层的卷积核是正方的，宽和高都覆盖 11 个像素，也就是说，卷积核有 $11 \times 11 = 121$ 个参数。运算的时候，卷积核从左上角 $[0, 0]$ 开始，覆盖了 11×11 个像素，计算出一个值，把这个值放到输出里，然后向右按照步长 stride 平移，再计算下一个值，直到平移到最右侧。到了最右侧之后，再向下移动一个步长，再从下一轮的行头开始计算，如次反复，直到计算完全部像素。

第一个卷积层的步长是 4 个像素，stride=4。

[1]代码参考: https://github.com/ShusenTang/Dive-into-DL-PyTorch/blob/master/docs/chapter05_CNN/5.6_alexnet.md

对图像做卷积，为了不漏失特征，需要对图像做 pandding。比如，从左上角 $[0,0]$ 做卷积，卷积核是 11×11，那么，在 $[0,0]$ 和 $[10,10]$ 之间的正方形区域内，特征没有被充分提取，因为卷积核不能走到负数坐标。所以需要做 paddning，在四个边上，每一边都增加若干个像素的空白边，增加的像素至少是 1 个像素，最多是 11 个像素，超过 11 个像素没有意义，超过卷积核尺寸了，卷积核只会白跑。

每一个卷积核，对整个输入图像做卷积操作，都会生成一个输出 channel 通道。每个输出 channel 通道，都对应一个卷积核，它们是一一对应关系。

第一个卷积层的输出是 96 个 channel，因此有 96 个卷积核。

第一个卷积层的参数是 $(1,96,11,4)$，1 是输入的 channel 数量，96 是输出的 channel 数量，11 是卷积核尺寸，4 是卷积核移动步长。

第二层是最大值池化层，参数 $(3,2)$，卷积核是 3×3 的，移动步长是 2。

池化层几个作用：第一，它本质上能是前一层的缩小版，所以前一层的特征如果有平移和变形，缩小了之后，这个平移和变形就没有了或者影响变小了，也就是让卷积层抽取的特征具有平移不变性、旋转不变性和尺度不变性；第二，减少数据维度，减少数据量，也就减少计算成本；第三，防止过拟合。

随后各层，是卷积层和池化层交替。

最后，几个全连接层，Dropout 层，输出层。

Dropout 层的作用，训练时，随机让一些神经元不参与训练，减小过拟合。

代码如下：

```
import torch
from torch import nn

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 96, 11, 4),
            nn.ReLU(),
            nn.MaxPool2d(3, 2),
            nn.Conv2d(96, 256, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(3, 2),
            nn.Conv2d(256, 384, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(384, 384, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(384, 256, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(3, 2)
        )

        self.fc = nn.Sequential(
            nn.Linear(256*5*5, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 10),
        )

    def forward(self, img):
        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
        return output

net = AlexNet()
lr= 0.0001
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

# 只用一个样本演示训练效果, batch_size=1, channel=1, width=224, height=224
x = torch.randn(1, 1, 224, 224)
y_target = torch.tensor([1,0,0,0,0,0,0,0,0,0], dtype=torch.float32).unsqueeze(
                                        dim=0)
```

```
for i in range(50):
    print('-' * 20)
    x = x.to(device)
    y_target = y_target.to(device)
    y = net(x)
    print('y = ',y)
    err_sum = torch.sum((y - y_target) ** 2 / 2)
    print('err_sum = ', err_sum)
    err_sum.backward()
    optimizer.step()
    optimizer.zero_grad()
```

输出结果：

```
--------------------
y =  tensor([[-0.0116,  0.0010, -0.0002,  0.0285,  0.0028,  0.0224,  0.0068,
                                        0.0066,
         -0.0165, -0.0067]], grad_fn=<AddmmBackward0>)
err_sum =  tensor(0.5125, grad_fn=<SumBackward0>)
--------------------
y =  tensor([[ 0.0324, -0.0323,  0.0126,  0.0174,  0.0005, -0.0017,  0.0134,
                                        0.0138,
          0.0036, -0.0084]], grad_fn=<AddmmBackward0>)
err_sum =  tensor(0.4691, grad_fn=<SumBackward0>)
--------------------
...
--------------------
y =  tensor([[ 0.8332, -0.0268, -0.0203, -0.0074,  0.0280, -0.0115,  0.0469,
                                        0.0158,
          0.0244,  0.0645]], grad_fn=<AddmmBackward0>)
err_sum =  tensor(0.0186, grad_fn=<SumBackward0>)
```

观察结果，可以看到 err_sum 在逐渐变小，且 y 值逐渐接近目标值，符合预期。

代码里的 “x = torch.randn(1, 1, 224, 224)”。第一个参数 1 是 batch_size，第二个参数是 1，表示图像是 1 个通道，后面两个 224 是图像的宽和高的像素数。四个参数加起来的意思就是生成一幅一个通道宽高都是 224 像素的图像。如果修改成 “x = torch.randn(3, 1, 224, 224)”，就是生成三幅一个通道宽高都是 224 像素的图像，那么对应的 y_target 和后面计算出来的 y 也都要略做修改。

第六章　批量归一

深度模型在训练的过程中，因为深度比较大，所以各隐层的输出，越接近输出层，波动的越大。ReLU 函数的输出值域是 $[0,+\infty)$，因此可以足够大。波动大，不利于学习。

所以，每隔几个隐层，设定一个批量归一层，把隐层的输出值的波动压缩一下，方式类似统计学的归一。

批量归一可以相当好地改进梯度消失和梯度爆炸问题。

设 x_i(i=0,m-1) 是批量归一层的一个批量输入，其中，m 是这个批量的样本数量，每个 x_i 是前一层的输出，批量归一层的步骤如下：

1. 求均值：

$$\mu_B = \frac{1}{m}\sum_{i=0}^{m-1} x_i$$

2. 求方差

$$\sigma_B^2 = \frac{1}{m}\sum_{i=0}^{m-1} x_i$$

3. 归一化

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

其中，ϵ 是为了确保分母为正的足够小的正实数，比如 0.0001，0.001。

4. 做线性变换后输出：

$$y = \gamma\hat{x_i} + \beta$$

γ 和 β 叫可学习重构参数。一个朴素的理解是这样的：前三个步骤将 x_i 的波动幅度降下去了，同时也改变了数据分布，那么再增加一个线性变换，也

许可以更好地使用归一数据，至少不会更差。

在具体实践的时候，会再增加一个平滑参数：这一次计算出来的均值，需要跟上一次计算出来的均值，做一下平滑，作为这一次的最终结果，方差也是如此。

批量归一，Pytorch 已经有封装类了，使用方式如下：

```
import torch
from torch import nn

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class AlexNetImproved(nn.Module):
    def __init__(self):
        super(AlexNetImproved, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 96, 11, 4),
            nn.ReLU(),
            nn.BatchNorm2d(96),
            nn.MaxPool2d(3, 2),
            nn.Conv2d(96, 256, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(3, 2),
            nn.Conv2d(256, 384, 3, 1, 1),
            nn.ReLU(),
            nn.BatchNorm2d(384),
            nn.Conv2d(384, 384, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(384, 256, 3, 1, 1),
            nn.ReLU(),
            nn.BatchNorm2d(256),
            nn.MaxPool2d(3, 2)
        )

        self.fc = nn.Sequential(
            nn.Linear(256*5*5, 4096),
            nn.ReLU(),
            nn.BatchNorm1d(4096),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 10)
        )

    def forward(self, img):
        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
```

```
        return output

net = AlexNetImproved()
lr= 0.0001
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

# 样本, batch_size=2, channel=1, widthe=224, height=224
x = torch.randn(2, 1, 224, 224)
y_target = torch.tensor([[1,0,0,0,0,0,0,0,0,0],[0,1,0,0,0,0,0,0,0,0]], dtype=
                                        torch.float32).unsqueeze(dim=0)

for i in range(20):
    print('-' * 20)
    x = x.to(device)
    y_target = y_target.to(device)
    y = net(x)
    err_sum = torch.sum((y - y_target) ** 2 / 2)
    print('err_sum = ', err_sum)
    err_sum.backward()
    optimizer.step()
    optimizer.zero_grad()
```

这里使用了二维 BatchNorm2d 和一维 BatchNorm1d 的批量归一，传入的参数是上一层网络的输出维数。

注意，BatchNorm1d 有一点特殊，它要求 *batch_size* 大于 1，因为等于 1 的时候，计算均值和方差是没有意义的，所以可以看到 “x = torch.randn(2, 1, 224, 224)”，对应的 *y_target* 也做了修改。

第七章　深度颠峰：ResNet

深度神经网络，层数越多，性能越好，但当层数增加到一定程度的时候，性能会下降：训练误差和预测误差都会升高（双高）。

这不是过拟合 (Overfitting)，过拟合是训练误差降低，预测误差升高。

这也不是梯度消失和梯度爆炸：批量归一可以相当好地解决梯度消失和梯度爆炸问题，即使给网络增加足够多的批量归一层，双高问题仍然在。

有意思的是，给双高的网络，再增加一层，性能会变得更差，不能维持原来的性能，也就是说，增加的这一层不能维持“什么都不做”的状态，因为什么都不做不会让网络性能更差。

“什么都不做”叫恒等映射 (identity mapping)

如果深度神经网络有恒等映射能力，那么至少可以做到增加层数的时候，性能不会下降。

但深度神经网络是没有恒等映射能力的–原因很简单–举个例子，深度神经网络的输入是一个 28×28 像素的数字 9 图像，输出是数字 9 的二进制表示 [0,1,0,0,0,0,0,0,0,0]，那么从输出的二进制是无论如何都无法恢复输入图像的，因此深度神经网络不具有恒等映射能力。假设 $\boldsymbol{y}$ 和 $\boldsymbol{x}$ 是同样的维数，深度神经网络的输出 $\boldsymbol{y} = f(\boldsymbol{x})$。$f(\boldsymbol{x})$ 等价于神经网络，那么必然有 $f(\boldsymbol{x}) \neq \boldsymbol{x}$，也就是 $\boldsymbol{y} \neq \boldsymbol{x}$。

既然如次，那就把条件放宽一些：给深度神经网络增加层数，不再试图寻求 $\boldsymbol{y} = \boldsymbol{x}$，而是 $\boldsymbol{y} = g(\boldsymbol{x}) + \boldsymbol{x}$，在这个等式，$\boldsymbol{x}$ 是输入值，如果让输出值 $\boldsymbol{y}$ 拟合 $\boldsymbol{x}$，本质上就是让 $g(\boldsymbol{x})$ 尽可能接近零。

用 BP 算法略微推导一下，就会发现输出加上 $g(\boldsymbol{x})$ 对网络的权重和偏差的迭代确实产生影响，这个策略让深度学习网络部分地具有了恒等映射。

这个结构暂定名“恒等映射层”。

给深度神经网络增加层数，只增加” 恒等映射层”。

经过试验发现，一个权重层 + 一个 ReLU 层 + 一个权重层组成一个恒等映射层比较合理。

由此，ResNet 把层数提升到前所未有的高度[1]，层数多达 152 层，改进版更是可以多到上千层。

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

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

def resnet_block(in_channels, out_channels, num_residuals, first_block=False):
    if first_block:
        assert in_channels == out_channels
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(in_channels, out_channels, use_1x1conv=True,
                                                    stride=2))
        else:
            blk.append(Residual(out_channels, out_channels))
    return nn.Sequential(*blk)

class GlobalAvgPool2d(nn.Module):
    def __init__(self):
        super(GlobalAvgPool2d, self).__init__()
    def forward(self, x):
        return F.avg_pool2d(x, kernel_size=x.size()[2:])

class FlattenLayer(torch.nn.Module):
    def __init__(self):
        super(FlattenLayer, self).__init__()
    def forward(self, x):
        return x.view(x.shape[0], -1)

class Residual(nn.Module):
    def __init__(self, in_channels, out_channels, use_1x1conv=False, stride=1)
                                            :
        super(Residual, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
                                                padding=1, stride=stride)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
                                                padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(in_channels, out_channels, kernel_size=1,
                                                    stride=stride)
        else:
```

[1]代码参考 https://github.com/ShusenTang/Dive-into-DL-PyTorch/blob/master/docs/chapter05_CNN/5.11_resnet.md

```
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.relu(Y + X)

net = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

net.add_module("resnet_block1", resnet_block(64, 64, 2, first_block=True))
net.add_module("resnet_block2", resnet_block(64, 128, 2))
net.add_module("resnet_block3", resnet_block(128, 256, 2))
net.add_module("resnet_block4", resnet_block(256, 512, 2))

net.add_module("global_avg_pool", GlobalAvgPool2d())
net.add_module("fc", nn.Sequential(FlattenLayer(), nn.Linear(512, 10)))

lr = 0.001
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

# 样本, batch_size=1, channel=1, widthe=224, height=224
x = torch.randn(1, 1, 224, 224)
y_target = torch.tensor([1,0,0,0,0,0,0,0,0,0], dtype=torch.float32).unsqueeze(
                                        dim=0)

for i in range(5):
    print('-' * 20)
    x = x.to(device)
    y_target = y_target.to(device)
    y = net(x)
    print('y = ',y)
    err_sum = torch.sum((y - y_target) ** 2 / 2)
    print('err_sum = ', err_sum)
    err_sum.backward()
    optimizer.step()
    optimizer.zero_grad()
```

输出结果：

```
--------------------
y =  tensor([[ 0.1225,  0.5027, -0.2381,  0.4419, -0.2694,  0.6617, -1.0418,
                                          0.1252,
          0.0777, -0.6003]], grad_fn=<AddmmBackward0>)
err_sum =  tensor(1.6263, grad_fn=<SumBackward0>)
--------------------
y =  tensor([[ 1.3223, -0.3957,  0.3317, -0.3211,  0.3904, -0.3438,  0.3937, -
                                          0.4108,
         -0.3094,  0.2254]], grad_fn=<AddmmBackward0>)
err_sum =  tensor(0.6072, grad_fn=<SumBackward0>)
--------------------
...
--------------------
y =  tensor([[ 1.2204, -0.2426,  0.0784, -0.1663,  0.0028, -0.0660,  0.4819,
                                          0.0140,
          0.0535, -0.2519]], grad_fn=<AddmmBackward0>)
err_sum =  tensor(0.2222, grad_fn=<SumBackward0>)
```

ResNet 之后，出现了 DenseNet。ResNet 每一层的输出，跨了一层跟后面一层的输出做连接。DenseNet 更夸张，每一层的输出，跟后面“所有”层的输出都做连接，性能也更好。

第八章　循环神经网络