# 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 $1 \times 2$ 的向量，x.t() 是它的转置，是一个 $2 \times 1$ 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 $\eta$ 的值必须足够小，否则 $\boldsymbol{w}$ 和 $b$ 计算结果不稳定。动态调整 $\eta$ 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。