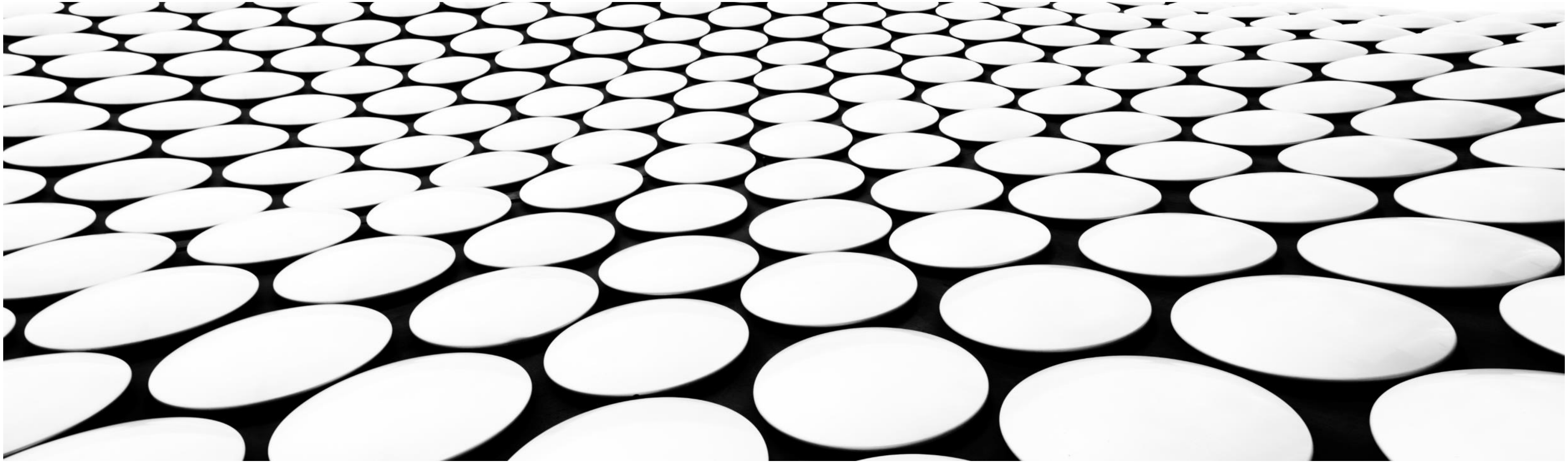
深度学习

邱怡轩



今天的主题

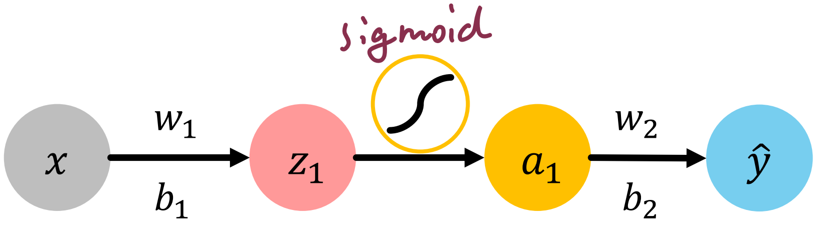
- 反向传播算法（续）
- PyTorch 自动微分

回到神经网络

- 利用之前介绍的方法，考虑一个简单的前馈神经网络
- $a_0 = x$，$z_1 = w_1 a_0 + b_1$，$a_1 = \sigma(z_1)$

Sigmoid函数

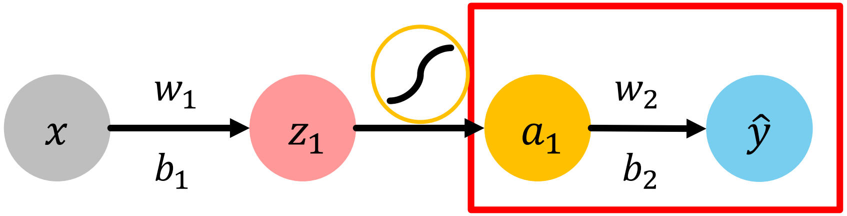
- $\hat{y} = z_2 = w_2 a_1 + b_2$
- $l = (y - \hat{y})^2$
- 需要计算 $\frac{\mathrm{d}l}{\mathrm{d}w_1}$，$\frac{\mathrm{d}l}{\mathrm{d}b_1}$，$\frac{\mathrm{d}l}{\mathrm{d}w_2}$，$\frac{\mathrm{d}l}{\mathrm{d}b_2}$



一个数据点

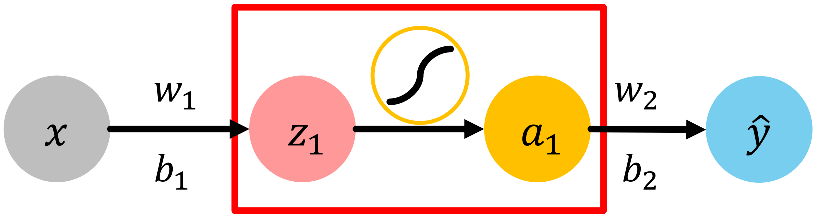
回到神经网络

- $\hat{y} = z_2$ 的上游导数
- $\dfrac{\mathrm{d}l}{\mathrm{d}z_2} = -2(y - z_2)$
- z_2 对 a_1, w_2, b_2 的局部导数
- $\dfrac{\mathrm{d}z_2}{\mathrm{d}a_1} = w_2, \quad \dfrac{\mathrm{d}z_2}{\mathrm{d}w_2} = a_1, \quad \dfrac{\mathrm{d}z_2}{\mathrm{d}b_2} = 1$
- 因此 $\dfrac{\mathrm{d}l}{\mathrm{d}a_1} = \dfrac{\mathrm{d}l}{\mathrm{d}z_2} \cdot w_2, \quad \dfrac{\mathrm{d}l}{\mathrm{d}w_2} = \dfrac{\mathrm{d}l}{\mathrm{d}z_2} \cdot a_1, \quad \dfrac{\mathrm{d}z_2}{\mathrm{d}b_2} = \dfrac{\mathrm{d}l}{\mathrm{d}z_2}$



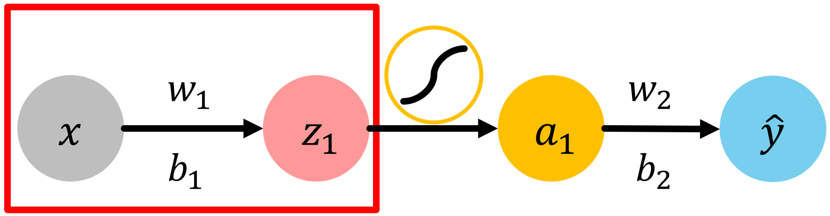
回到神经网络

- a_1 的上游导数 $\frac{\mathrm{d}l}{\mathrm{d}a_1}$ 已经在上一层计算好
- a_1 对 z_1 的局部导数
- $\frac{\mathrm{d}a_1}{\mathrm{d}z_1} = \sigma'(z_1) = \sigma(z_1)[1-\sigma(z_1)] = a_1(1-a_1)$
- 因此 $\frac{\mathrm{d}l}{\mathrm{d}z_1} = \frac{\mathrm{d}l}{\mathrm{d}a_1} \cdot a_1(1-a_1)$



回到神经网络

- z_1 的上游导数 $\frac{\mathrm{d}l}{\mathrm{d}z_1}$ 已经在上一层计算好
- z_1 对 w_1, b_1 的局部导数
- $\frac{\mathrm{d}z_1}{\mathrm{d}w_1} = x, \quad \frac{\mathrm{d}z_1}{\mathrm{d}b_1} = 1$
- 因此 $\frac{\mathrm{d}l}{\mathrm{d}w_1} = \frac{\mathrm{d}l}{\mathrm{d}z_1} \cdot x, \quad \frac{\mathrm{d}l}{\mathrm{d}b_1} = \frac{\mathrm{d}l}{\mathrm{d}z_1}$



扩展

- 这一方法可以很容易扩展到任意层数的情形
- 但每层有多个神经元的情况怎么办?
- 此时 W_i 是矩阵，b_i 是向量

先上结论

规则0

- 参数导数的维度和参数的维度保持一致

x x

规则1

- 对于逐元素计算的激活函数，$a = \sigma(z)$
- $a = (a^1, \ldots, a^m)^T,\ z = (z^1, \ldots, z^m)^T,\ a^i = \sigma(z^i)$
- 导数也可逐元素计算

- 假设 a 的上游导数为 $\frac{\mathrm{d}l}{\mathrm{d}a} = \left(\frac{\mathrm{d}l}{\mathrm{d}a^1}, \ldots, \frac{\mathrm{d}l}{\mathrm{d}a^m}\right)^T$，则
- $\frac{\mathrm{d}l}{\mathrm{d}z} = \left(\frac{\mathrm{d}l}{\mathrm{d}a^1} \cdot \sigma'(z^1), \ldots, \frac{\mathrm{d}l}{\mathrm{d}a^m} \cdot \sigma'(z^m)\right)^T = \frac{\mathrm{d}l}{\mathrm{d}a} \circ \sigma'(z)$

简写

两个向量逐元素相乘

规则2

- 对于线性变换，$z = Wa + b$，$W \in \mathrm{R}^{n \times m}$
- $a = (a^1, \dots, a^m)^T$，$z = (z^1, \dots, z^n)^T$
- 假设 z 的上游导数为 $\frac{\mathrm{d}l}{\mathrm{d}z} = \left(\frac{\mathrm{d}l}{\mathrm{d}z^1}, \dots, \frac{\mathrm{d}l}{\mathrm{d}z^n}\right)^T$
- 那么
- $\frac{\mathrm{d}l}{\mathrm{d}W} = \frac{\mathrm{d}l}{\mathrm{d}z} a^T$，$\frac{\mathrm{d}l}{\mathrm{d}b} = \frac{\mathrm{d}l}{\mathrm{d}z}$，$\frac{\mathrm{d}l}{\mathrm{d}a} = W^T \frac{\mathrm{d}l}{\mathrm{d}z}$

规则2

- 对于线性变换，$z = Wa + b$，$W \in \mathrm{R}^{n\times m}$
- $a = (a^1, \dots, a^m)^T$，$z = (z^1, \dots, z^n)^T$
- 假设 z 的上游导数为 $\frac{\mathrm{d}l}{\mathrm{d}z} = \left(\frac{\mathrm{d}l}{\mathrm{d}z^1}, \dots, \frac{\mathrm{d}l}{\mathrm{d}z^n}\right)^T$
- 那么
- $\underbrace{\boxed{\frac{\mathrm{d}l}{\mathrm{d}W}}}_{n\times m} = \underbrace{\boxed{\frac{\mathrm{d}l}{\mathrm{d}z}}}_{n\times 1}\underbrace{\boxed{a^T}}_{1\times m}$，$\frac{\mathrm{d}l}{\mathrm{d}b} = \frac{\mathrm{d}l}{\mathrm{d}z}$，$\frac{\mathrm{d}l}{\mathrm{d}a} = W^T\frac{\mathrm{d}l}{\mathrm{d}z}$

规则2

- 对于线性变换，$z = Wa + b$，$W \in \mathrm{R}^{n\times m}$
- $a = (a^1, \ldots, a^m)^T$，$z = (z^1, \ldots, z^n)^T$
- 假设 z 的上游导数为 $\frac{\mathrm{d}l}{\mathrm{d}z} = \left(\frac{\mathrm{d}l}{\mathrm{d}z^1}, \ldots, \frac{\mathrm{d}l}{\mathrm{d}z^n}\right)^T$
- 那么
- $\frac{\mathrm{d}l}{\mathrm{d}W} = \frac{\mathrm{d}l}{\mathrm{d}z} a^T$, $\underbrace{\boxed{\frac{\mathrm{d}l}{\mathrm{d}b}}}_{n\times 1} = \underbrace{\boxed{\frac{\mathrm{d}l}{\mathrm{d}z}}}_{n\times 1}$, $\frac{\mathrm{d}l}{\mathrm{d}a} = W^T \frac{\mathrm{d}l}{\mathrm{d}z}$

规则2

- 对于线性变换，$z = Wa + b$，$W \in \mathrm{R}^{n\times m}$
- $a = (a^1, \dots, a^m)^T$，$z = (z^1, \dots, z^n)^T$
- 假设 z 的上游导数为 $\frac{\mathrm{d}l}{\mathrm{d}z} = \left(\frac{\mathrm{d}l}{\mathrm{d}z^1}, \dots, \frac{\mathrm{d}l}{\mathrm{d}z^n}\right)^T$
- 那么
- $\frac{\mathrm{d}l}{\mathrm{d}W} = \frac{\mathrm{d}l}{\mathrm{d}z} a^T$，$\frac{\mathrm{d}l}{\mathrm{d}b} = \frac{\mathrm{d}l}{\mathrm{d}z}$，$\underset{m\times 1}{\boxed{\frac{\mathrm{d}l}{\mathrm{d}a}}} = \underset{m\times n}{\boxed{W^T}}\underset{n\times 1}{\boxed{\frac{\mathrm{d}l}{\mathrm{d}z}}}$

规则3

- 不要照搬数学表达式
- 实际编程中要考虑到数据的存储方式

理论 VS 实际

规则0

理论

- 参数导数的维度和参数的维度保持一致

实际

- 参数导数的维度和参数的维度保持一致

规则1

理论

- 对于逐元素计算的激活函数，导数也可逐元素计算
- $a: [m \times 1],\ z: [m \times 1],\ a_i = \sigma(z_i)$
- $\frac{\mathrm{d}l}{\mathrm{d}z} = \frac{\mathrm{d}l}{\mathrm{d}a} \circ \sigma'(z)$

实际

- 对于逐元素计算的激活函数，导数也可逐元素计算
- $A: [n \times m],\ Z: [n \times m],\ A_{ij} = \sigma(Z_{ij})$
- $\frac{\mathrm{d}l}{\mathrm{d}Z} = \frac{\mathrm{d}l}{\mathrm{d}A} \circ \sigma'(Z)$

规则1.5

$$\underset{2\times1}{1_n}\ \underset{1\times3}{b^T} = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 \end{bmatrix}$$ 加入数据的维度

理论

左乘

- 线性变换，$z = \boxed{W}a + b$，$W \in \mathrm{R}^{p\times m}$
- $a: [m \times 1]$，$z: [p \times 1]$
- $W: [p \times m]$，$b: [p \times 1]$

- a 代表上一层神经元（m个）
- z 代表下一层神经元（p个）

实际

理论w的转置

右乘

☆ softmax 回归

- $Z = A\boxed{W} + \boxed{\mathbf{1}_n}b^T$，$W \in \mathrm{R}^{m\times p}$
- $A: [n \times m]$，$Z: [n \times p]$
- $W: [m \times p]$，$b: [p \times 1]$

- n 代表样本量
- A 的第 i 行代表第 i 个观测在上一层的神经元
- Z 的第 i 行代表第 i 个观测在下一层的神经元

规则2

理论

- $$\underset{p\times m}{\frac{\mathrm{d}l}{\mathrm{d}W}} = \underset{p\times 1}{\frac{\mathrm{d}l}{\mathrm{d}z}}\,\overset{1\times m}{a^T}$$
- $$\underset{p\times 1}{\frac{\mathrm{d}l}{\mathrm{d}b}} = \underset{p\times 1}{\frac{\mathrm{d}l}{\mathrm{d}z}}$$
- $$\underset{m\times 1}{\frac{\mathrm{d}l}{\mathrm{d}a}} = \underset{m\times p}{W^T}\,\underset{p\times 1}{\frac{\mathrm{d}l}{\mathrm{d}z}}$$

实际

- $$\overset{m\times p}{\frac{\mathrm{d}l}{\mathrm{d}W}} = \overset{m\times n}{A^T}\,\overset{n\times p}{\frac{\mathrm{d}l}{\mathrm{d}Z}}$$
- $$\underset{p\times 1}{\frac{\mathrm{d}l}{\mathrm{d}b}} = \underset{p\times n}{\left(\frac{\mathrm{d}l}{\mathrm{d}Z}\right)^T}\,\underset{n\times 1}{\mathbf{1}_n}$$
- $$\underset{n\times m}{\frac{\mathrm{d}l}{\mathrm{d}A}} = \underset{n\times p}{\frac{\mathrm{d}l}{\mathrm{d}Z}}\,\underset{p\times m}{W^T}$$

自动微分与优化

- 参见 `lec5-module.ipynb`

自动微分代码 $f(x,y) = x \cdot \log(x) + \sin(xy)$

```python
import torch
x = torch.tensor([1.0], requires_grad=True)
y = torch.tensor([2.0], requires_grad=True)

f = x * torch.log(x) + torch.sin(x * y)

f.backward()
```

矩阵、向量也可以

backward 进行反向传播

```python
x.grad
y.grad
```

不需要 pytorch 记录导数值时

```python
with torch.no_grad()
    torch.log(x) + 1.0 + y * torch.cos(x * y)
```

问题：行列式为正的矩阵 X，定义 $f(X) = \log\det(X)$，其中 det 为 X 的行列式

那么 $\frac{\partial f}{\partial X}$ 应该是什么？

$\partial f = d(\log\det(X)) = (X^{-1})^T$

dX — dX = $(X$

线性模型自动微分

$\underset{n\times p}{X}\underset{p\times 1}{\beta} + \underset{n\times 1}{\varepsilon}$ 对β优化，对β求导 参数!!!

$Y = X\beta + \varepsilon$

损失函数： torch.mean(torch.square(y-yhat))

(y - yhat)**2

循环

nepoch = 500 迭代次数

learning_rate = 0.01 学习率

Losses = [ ] 损失值

```
for i in range(nepoch):
    loss = loss_fn (bhat, x, y)
    loss.backward()
    losses.append (loss.item())
```

这里获得 loss 为 tensor

求导 获得 bhat.grad

loss 具体值，而不是 tensor

为了画图像

```
if i% 50 ==0:
    print(f"iteration {i}, loss = {loss.item()}, error={torch.mean
                                                    (torch.square(y-yhat))

with torch.no_grad():
    bhat -= learning_rate * bhat.grad

    #清空梯度项
    bhat.grad= None
```

模块化编程

```
import torch.nn as nn

class MyModel(nn.Module)
    def __init__(self, beta_dim):
        super(MyModel, self).__init__()
        self.bhat = nn.Parameter(torch.zeros(beta_dim))
```

参数、自动微分

线性很多层

```
    def forward(self, x):
        yhat = torch.matmul(x, self.bhat)
```

预测值

```
    return yhat

model = MyModel(beta_dim=p)
print(list(model.parameters()))
```

```
nepoch = 500                 迭代次数
learning_rate = 0.01         学习率
Losses = []                  损失值

                    对参数优化
opt = torch.optim.SGD(model.parameters(), lr=learning_rate)

for i in range(nepoch):
    yhat = model(x)      调用forward返回                预测值
    loss = torch.mean(torch.square(y-yhat))            #loss = loss_fn(bhat, x, y)
    losses.append(loss.item())                         损失函数 与分布有关

    opt.zero_grad()                  with torch.no_grad():
    loss.backward()   求导               bhat -= learning_rate * bhat.grad
                                         #清空梯度项
```

```
opt.step()   实现更新beta ——        bhat.grad = None

if  i % 50 == 0:
    print(f"iteration {i}, loss = {loss.item()},
```

$N \geq 0$