计算机视觉

张健

数字媒体研究中心
信息工程学院
北京大学深圳研究生院

2021.10.20

课程内容

- 作业讨论
- 初始神经网络

回顾

多维

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \dots, \frac{\partial f(\boldsymbol{x})}{\partial x_d}\right]^\top$$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f(\boldsymbol{x})$$

标量f对矩阵X的导数

$$\frac{\partial f}{\partial X} = \left[\frac{\partial f}{\partial X_{ij}}\right]$$

- 定义在计算中并不好用
- 用矩阵运算更整洁
- 要找一个从整体出发的算法

矩阵求导基础

一元微积分中的导数（标量对标量的导数）与微分有联系：

$$df = f'(x)dx$$

多元微积分中的梯度（标量对向量的导数）也与微分有联系：

$$df = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} dx_i = \frac{\partial f}{\partial \boldsymbol{x}}^T d\boldsymbol{x}$$

第一个等号是全微分公式，第二个等号表达了梯度与微分的联系

全微分 df 是梯度向量 $\frac{\partial f}{\partial \boldsymbol{x}}$ (n×1)与微分向量 $d\boldsymbol{x}$ (n×1)的内积

受前面一元和多元微积分启发，可以将矩阵导数与微分建立联系：

$$df = \sum_{i=1}^{m}\sum_{j=1}^{n}\frac{\partial f}{\partial X_{ij}}dX_{ij} = \operatorname{tr}\left(\frac{\partial f}{\partial X}^{T} dX\right)$$

其中tr代表迹(trace)是方阵对角线元素之和，满足性质：

对尺寸相同的矩阵A,B， $\operatorname{tr}(A^T B) = \sum_{i,j} A_{ij}B_{ij}$ 即 $\operatorname{tr}(A^T B)$ 是矩阵A,B的**内积**

第一个等号是全微分公式，第二个等号表达了矩阵导数与微分的联系：

全微分 df 是导数 $\dfrac{\partial f}{\partial X}$ (m×n)与微分矩阵 dX (m×n)的内积。

然后通过矩阵微分运算法则可高效快速求解。

常用的矩阵微分的运算法则:

加减法: $d(X \pm Y) = dX \pm dY$；矩阵乘法: $d(XY) = (dX)Y + XdY$；转置: $d(X^T) = (dX)^T$；迹: $d\operatorname{tr}(X) = \operatorname{tr}(dX)$。

逐元素乘法: $d(X \odot Y) = dX \odot Y + X \odot dY$，$\odot$表示尺寸相同的矩阵X,Y逐元素相乘。

逐元素函数: $d\sigma(X) = \sigma'(X) \odot dX$，$\sigma(X) = [\sigma(X_{ij})]$是逐元素标量函数运算, $\sigma'(X) = [\sigma'(X_{ij})]$是逐元素求导数。例如

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}, d\sin(X) = \begin{bmatrix} \cos X_{11} dX_{11} & \cos X_{12} dX_{12} \\ \cos X_{21} dX_{21} & \cos X_{22} dX_{22} \end{bmatrix} = \cos(X) \odot dX$$

矩阵迹的性质：

1. 标量套上迹：$a = \mathrm{tr}(a)$
2. 转置：$\mathrm{tr}(A^T) = \mathrm{tr}(A)$。
3. 线性：$\mathrm{tr}(A \pm B) = \mathrm{tr}(A) \pm \mathrm{tr}(B)$。
4. 矩阵乘法交换：$\mathrm{tr}(AB) = \mathrm{tr}(BA)$，其中 A 与 B^T 尺寸相同。两侧都等于 $\sum_{i,j} A_{ij} B_{ji}$。
5. 矩阵乘法/逐元素乘法交换：$\mathrm{tr}(A^T(B \odot C)) = \mathrm{tr}((A \odot B)^T C)$，其中 A, B, C 尺寸相同。两侧都等于 $\sum_{i,j} A_{ij} B_{ij} C_{ij}$。

矩阵求导基础

例：已知 $Y=XW$ 和 $\frac{\partial f}{\partial Y}$，求 $\frac{\partial f}{\partial X}$， $\frac{\partial f}{\partial W}$.

$$df=\mathrm{tr}\left(\frac{\partial f^T}{\partial Y}dY\right)=\mathrm{tr}\left(\frac{\partial f^T}{\partial Y}d(XW)\right)=\mathrm{tr}\left(\frac{\partial f^T}{\partial Y}dXW\right)$$

$$=\mathrm{tr}\left(W\frac{\partial f^T}{\partial Y}dX\right)=\mathrm{tr}\left(\left(\frac{\partial f}{\partial Y}W^T\right)^T dX\right)$$

$$\frac{\partial f}{\partial X}=\frac{\partial f}{\partial Y}W^T$$

同理： $$\frac{\partial f}{\partial W}=X^T\frac{\partial f}{\partial Y}$$

练习

```
x = torch.tensor(1., requires_grad=True)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)

y = w*x + b

y.backward()

print(w.grad)
print(x.grad)
print(b.grad)
```

```
y = w*x + b
y.backward()

print(w.grad)
print(x.grad)
print(b.grad)
```

```
y = w*x + b
w.grad.zero_()
x.grad.zero_()
b.grad.zero_()

y.backward()
print(w.grad)
print(x.grad)
print(b.grad)
```

作业

```
import torch
torch.manual_seed(0)

x = torch.randn(10,4, requires_grad=True)
W = torch.randn(4,4, requires_grad=True)
y = torch.randn(10,4, requires_grad=True)
```

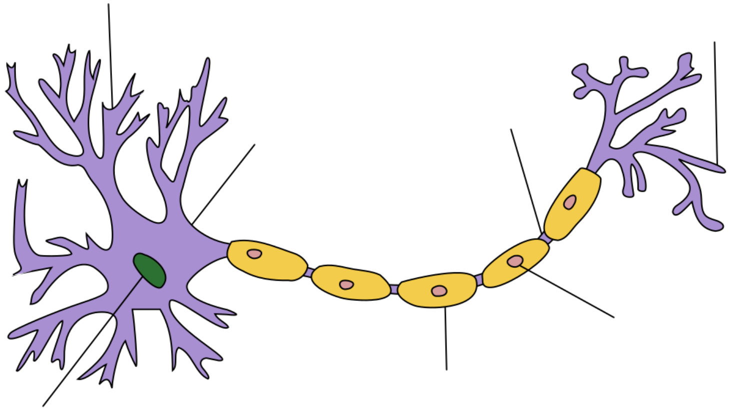
目标函数： $f = ||\max(XW, 0) - Y||_F^2$

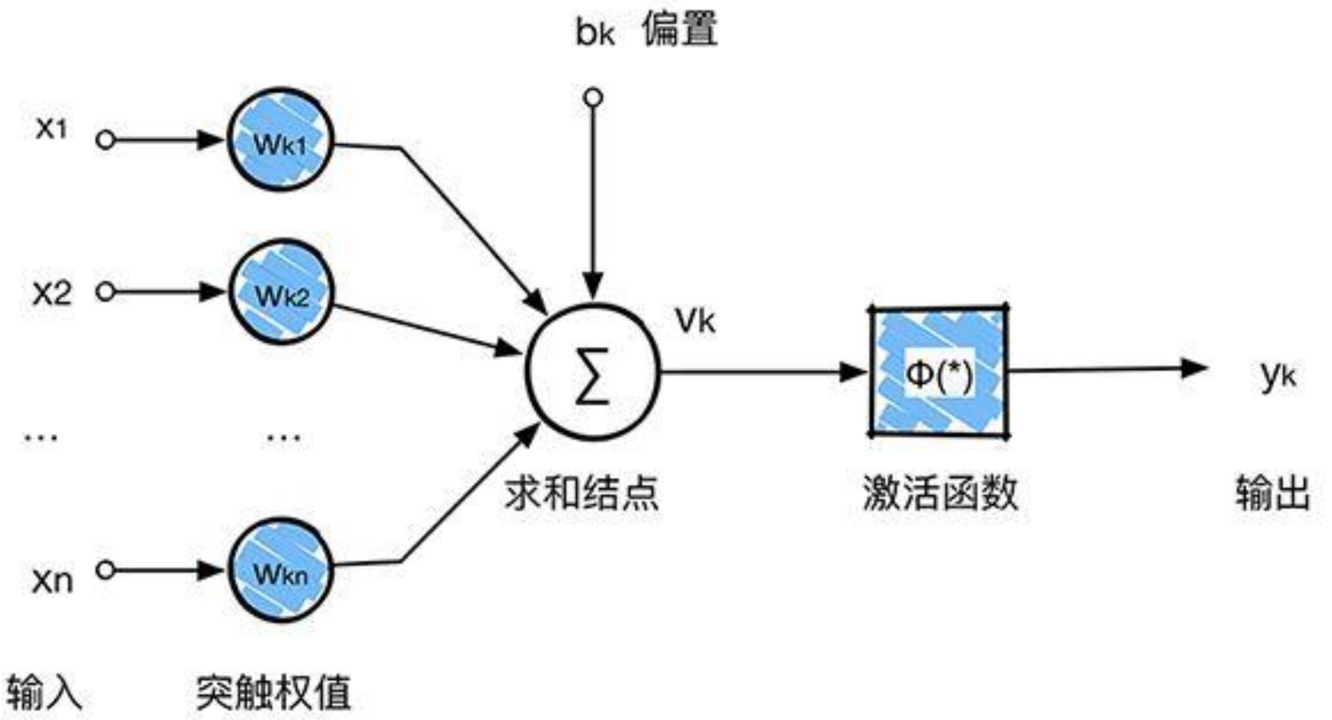
手动写出以下表达式，并用PyTorch进行验证：

$$\frac{\partial f}{\partial W} \qquad \frac{\partial f}{\partial X} \qquad \frac{\partial f}{\partial Y}$$

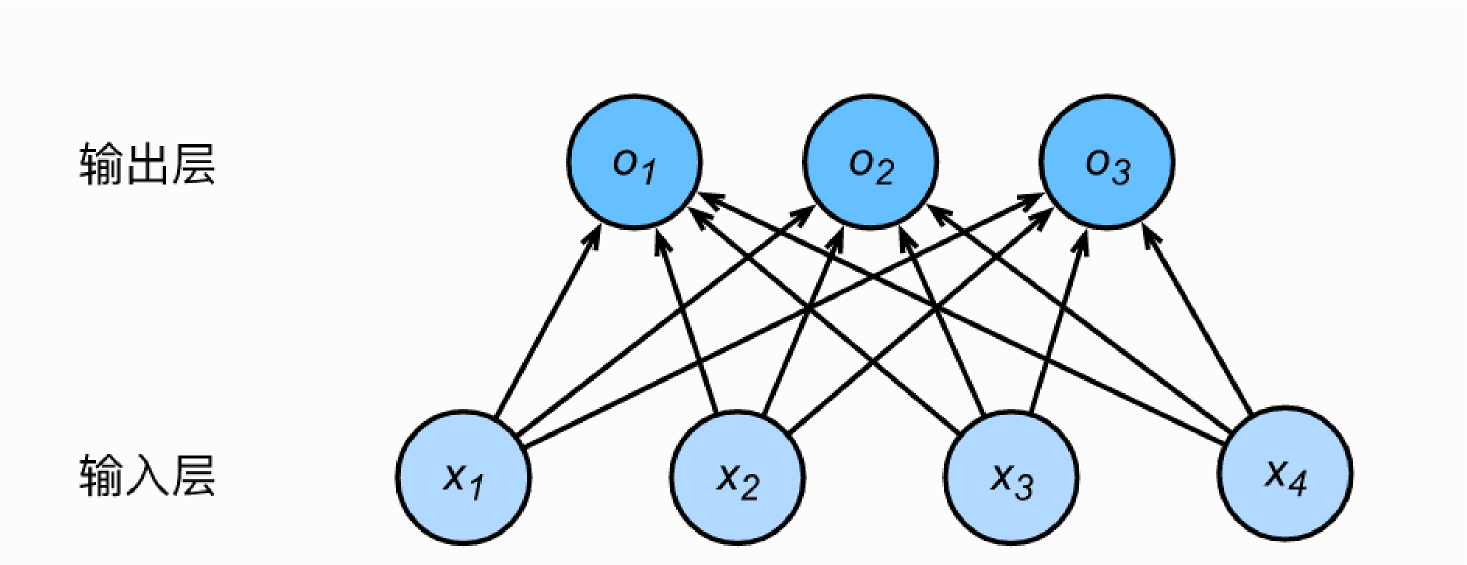
初识神经网络

神经元

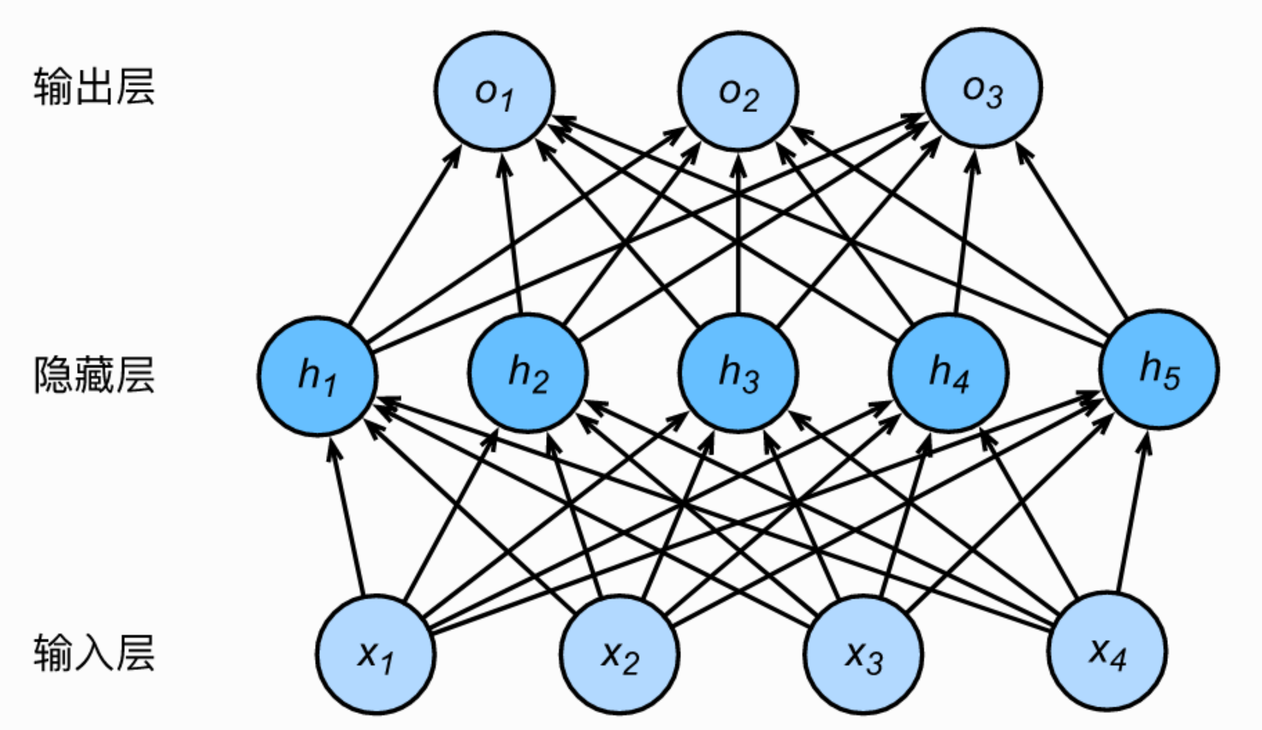




一层全连接网络



两层全连接网络

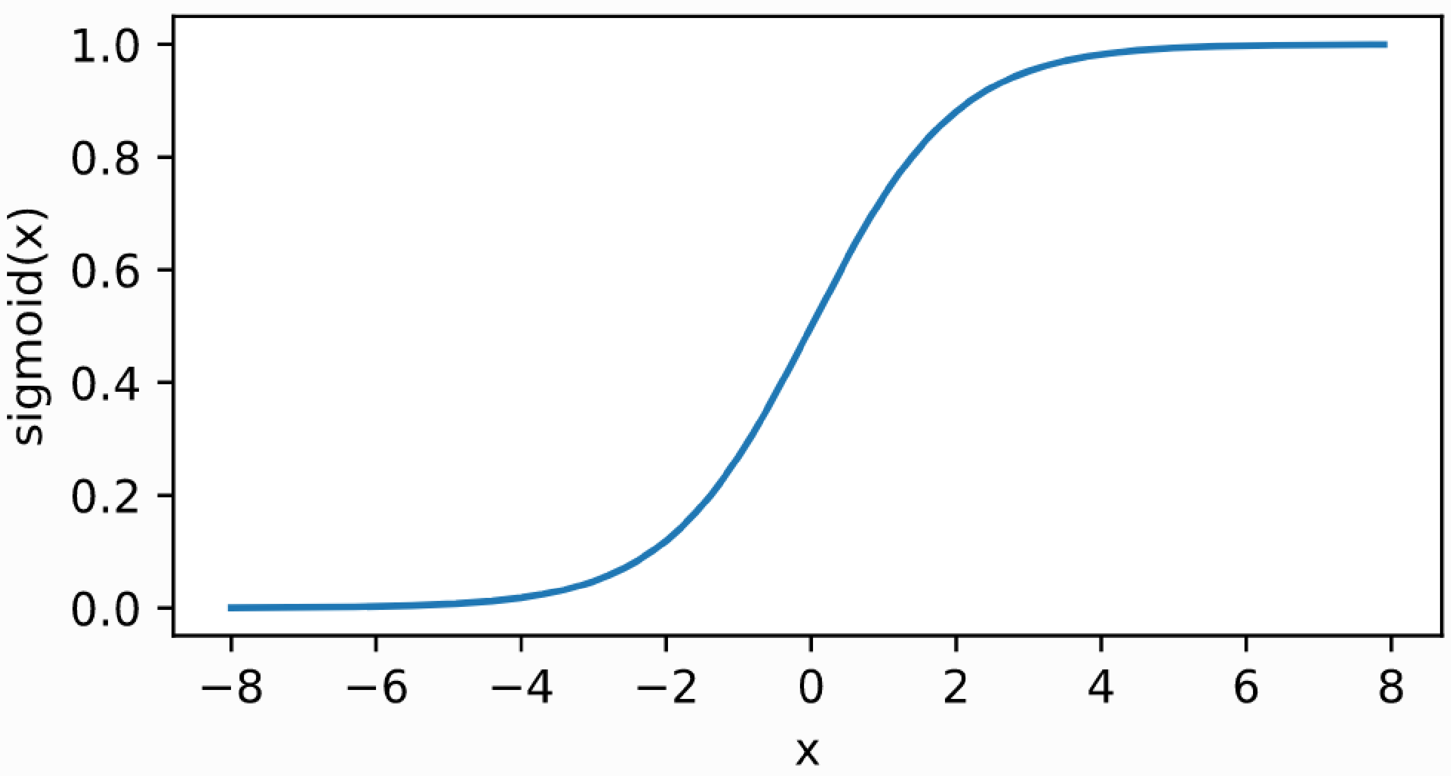


$$
\begin{aligned}
\boldsymbol{H} &= \phi(\boldsymbol{X}\boldsymbol{W}_h + \boldsymbol{b}_h),\\
\boldsymbol{O} &= \boldsymbol{H}\boldsymbol{W}_o + \boldsymbol{b}_o,
\end{aligned}
$$

https://playground.tensorflow.org/

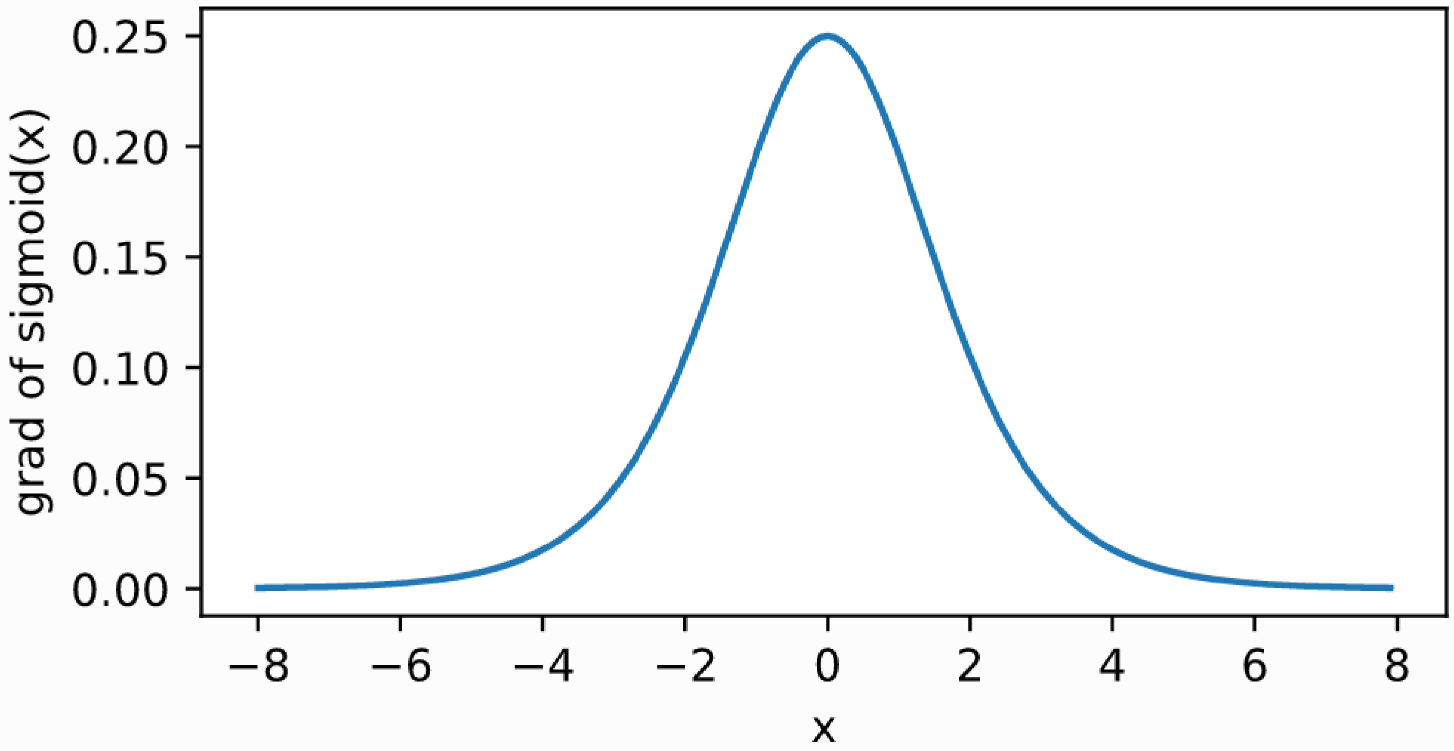
Sigmoid

$$\mathrm{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$



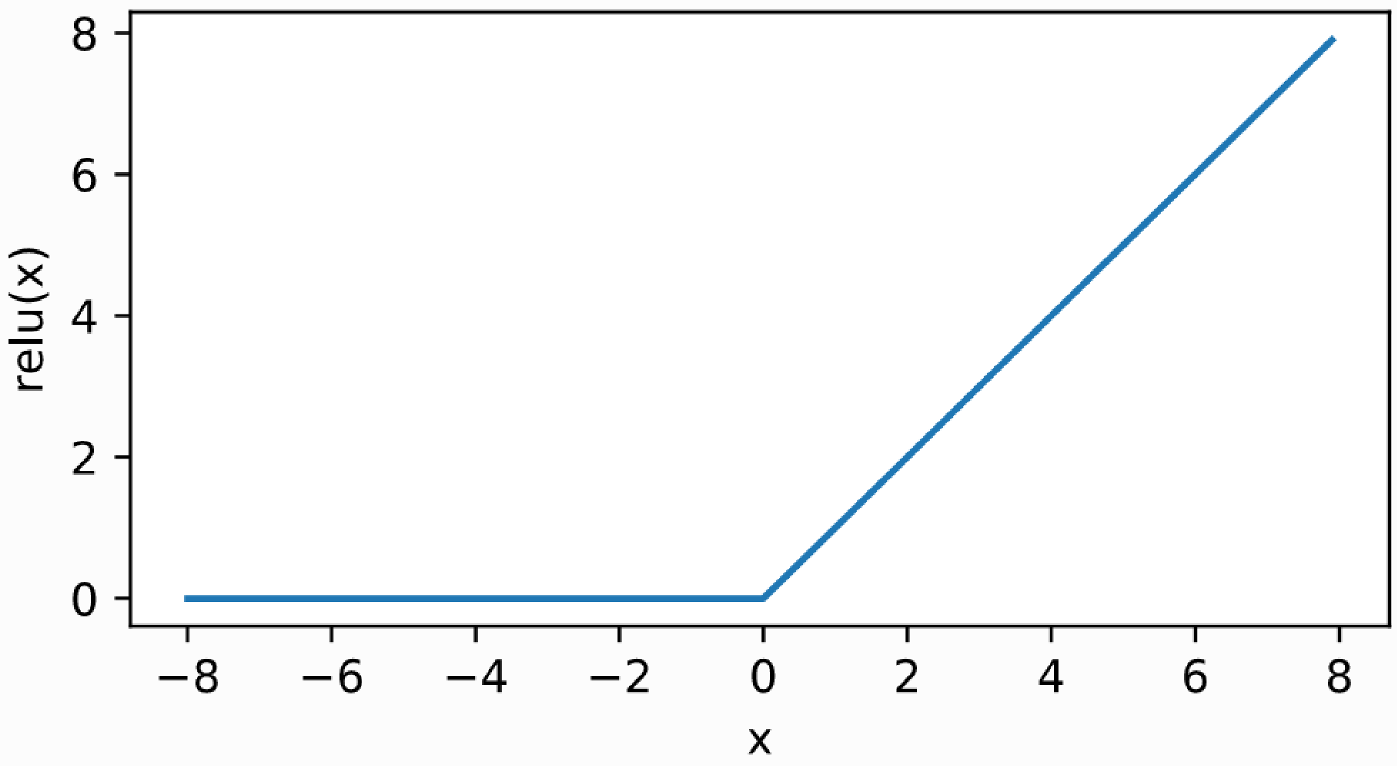
Sigmoid

$$\mathrm{sigmoid}'(x) = \mathrm{sigmoid}(x)\left(1 - \mathrm{sigmoid}(x)\right)$$



ReLU（rectified linear unit）

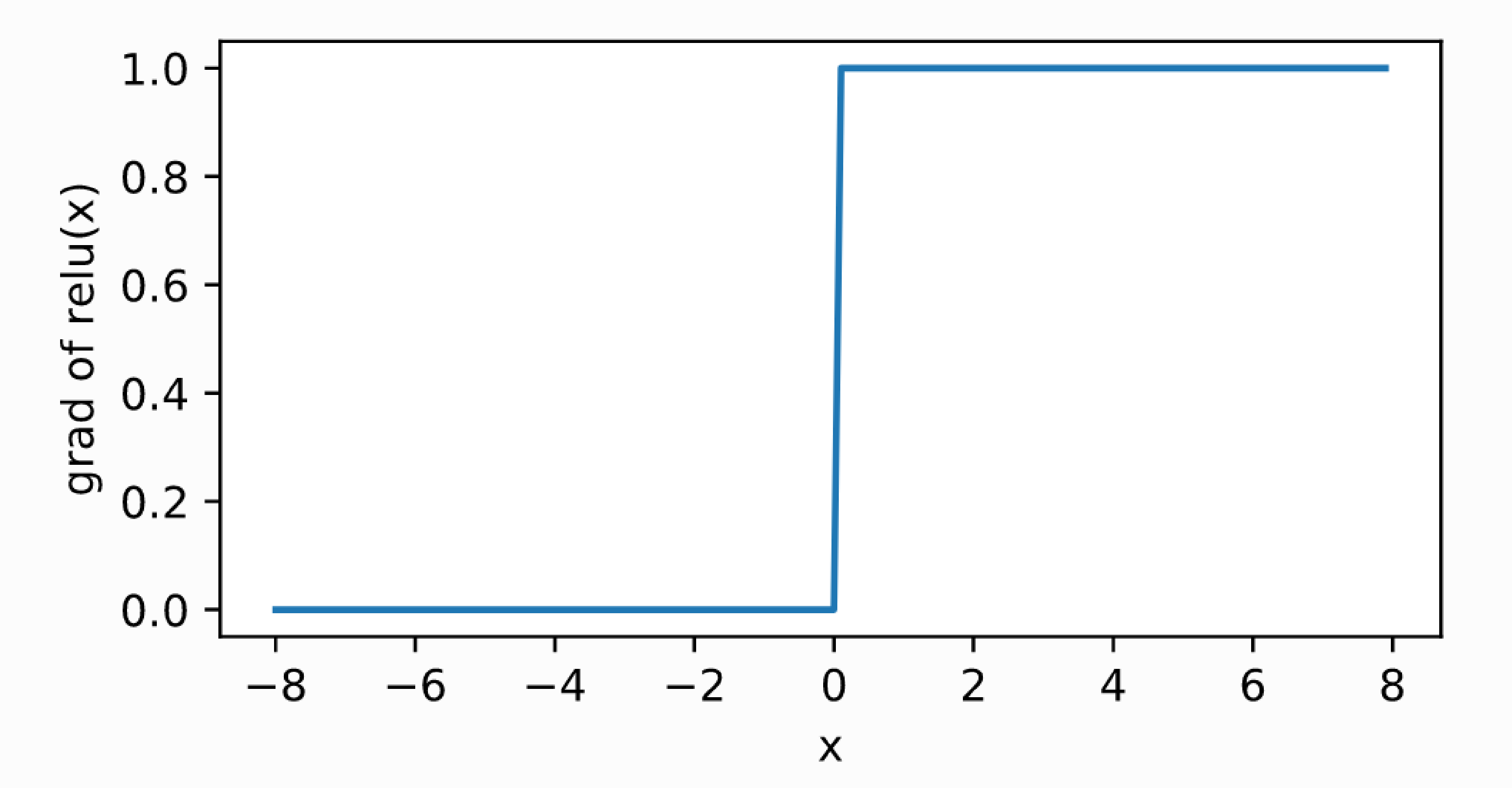
$$\mathrm{ReLU}(x) = \max(x, 0)$$



ReLU（rectified linear unit）

北京大學 PEKING UNIVERSITY

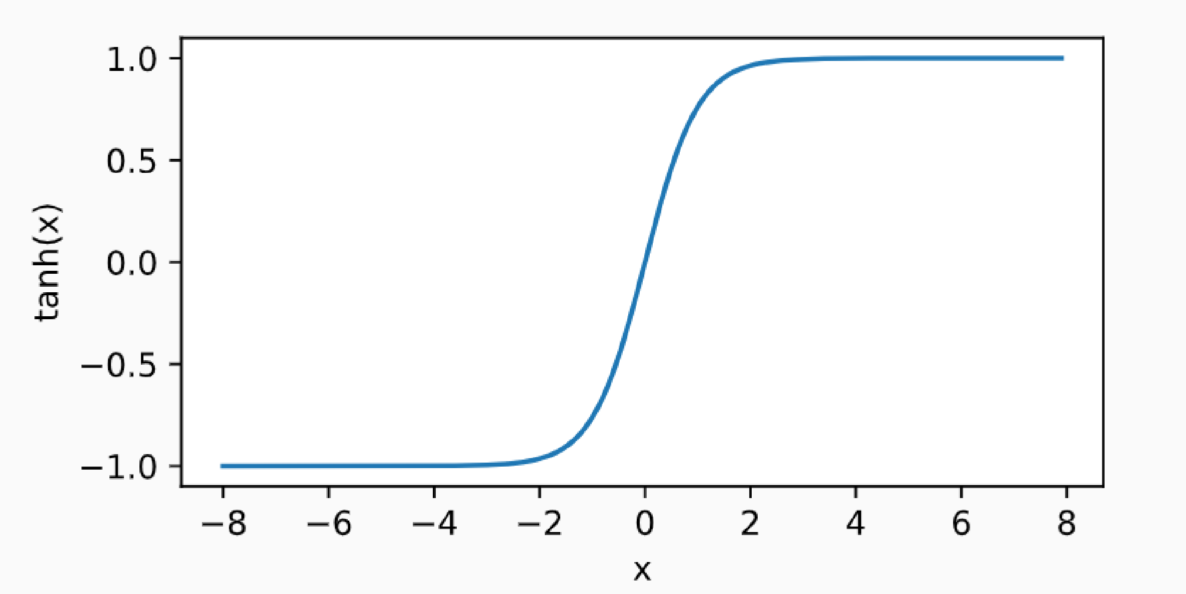
ReLU 函数的导数

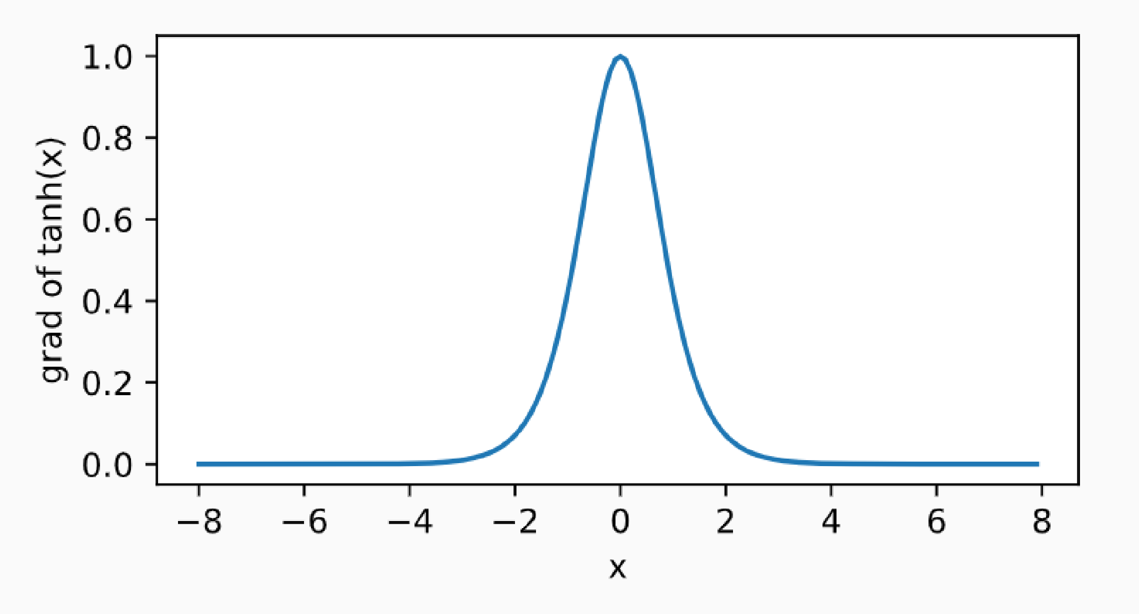


tanh（双曲正切）

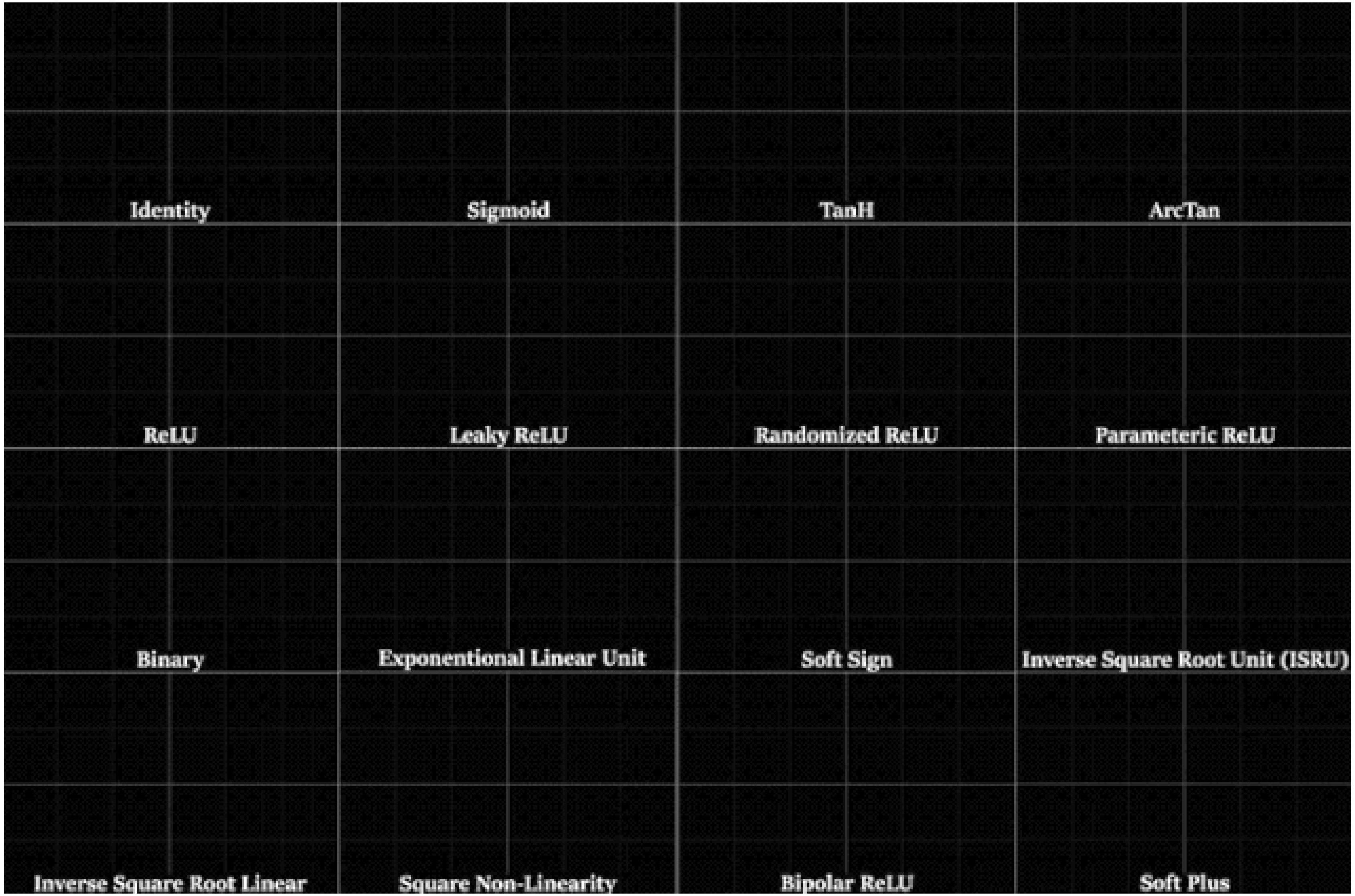
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

$$\tanh'(x) = 1 - \tanh^2(x).$$





tanh（双曲正切)



https://baijiahao.baidu.com/s?id=1653421414340022957&wfr=spider&for=pc

PyTorch直接搭建全连接层

北京大学 PEKING UNIVERSITY

class `torch.nn.Linear`(*in_features, out_features, bias=True*) [source]

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters:

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to False, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, in\_features)$ where $*$ means any number of additional dimensions
- Output: $(N, *, out\_features)$ where all but the last dimension are the same shape as the input.

Variables:

- **weight** – the learnable weights of the module of shape *(out_features x in_features)*
- **bias** – the learnable bias of the module of shape *(out_features)*

PyTorch直接搭建全连接层

全连接层的输入为二维张量

例子1：全连接网络例子

```
import torch.nn as nn
input = torch.randn(10,100) # (BatchSize, length)
fc1 = nn.Linear(100, 200)
output_fc1 = fc1(input)

print("Size of Input is", input.shape)
print("Size of fc1 Output is", output_fc1.shape)

params = list(fc1.parameters())
print("Parameter Number of fc1 is %d " % len(params))

for name, parameters in fc1.named_parameters():
    print(name, ':', parameters.size())
```

```
Size of Input is torch.Size([10, 100])
Size of fc1 Output is torch.Size([10, 200])
The Number of fc1 is 2
weight : torch.Size([200, 100])
bias : torch.Size([200])
```

PyTorch直接搭建全连接层

例子2：全连接网络例子(bias=False)

```
import torch.nn as nn
input = torch.randn(10,100) # (BatchSize, length)
fc1 = nn.Linear(100, 200, bias=False)
output_fc1 = fc1(input)

print("Size of Input is", input.shape)
print("Size of fc1 Output is", output_fc1.shape)

params = list(fc1.parameters())
print("Parameter Number of fc1 is %d " % len(params))

for name, parameters in fc1.named_parameters():
    print(name, ':', parameters.size())
```

```
Size of Input is torch.Size([10, 100])
Size of fc1 Output is torch.Size([10, 200])
The Number of fc1 is 1
weight : torch.Size([200, 100])
```

从Numpy到PyTorch各种两层全连接实现

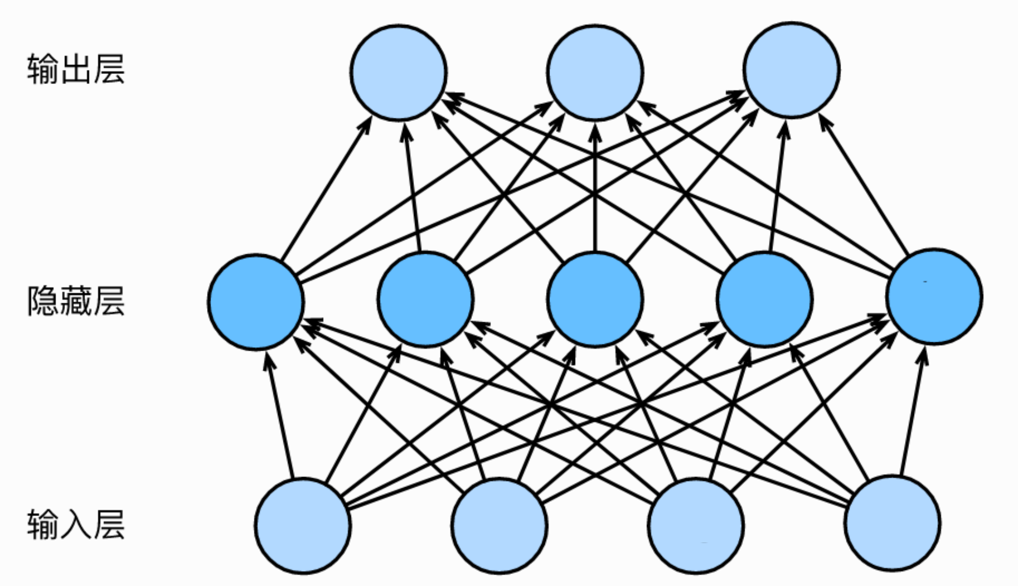
一个全连接ReLU神经网络，一个隐藏层，没有bias。
用来从x预测y，使用L2 Loss。

$$h = XW_1$$

$$h_{\text{relu}} = \max(0, h)$$

$$Y_{\text{pred}} = h_{\text{relu}} W_2$$

$$f = ||Y - Y_{\text{pred}}||_F^2$$



对W1和W2的偏导数怎么求？手动推出！

从Numpy到PyTorch各种两层全连接实现

方案一：Numpy 实现

```
import numpy as np

N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
```

```
for it in range(501):
    # Forward pass
    h = x.dot(w1) # N * H
    h_relu = np.maximum(h, 0) # N * H
    y_pred = h_relu.dot(w2) # N * D_out

    # compute loss
    loss = np.square(y_pred - y).sum()
    if it % 50 == 0:
        print(it, loss)

    # Backward pass
    # compute the gradient
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h<0] = 0
    grad_w1 = x.T.dot(grad_h)

    # update weights of w1 and w2
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

$$h = XW_1$$

$$h_{\text{relu}} = \max(0, h)$$

$$Y_{\text{pred}} = h_{\text{relu}} W_2$$

$$f = ||Y - Y_{\text{pred}}||_F^2$$

演示 W5_PyTorch_Network.ipynb

从Numpy到PyTorch各种两层全连接实现

方案二：PyTorch：Tensor 实现

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

w1 = torch.randn(D_in, H)
w2 = torch.randn(H, D_out)

learning_rate = 1e-6
```

```
for it in range(501):
    # Forward pass
    h = x.mm(w1) # N * H
    h_relu = h.clamp(min=0) # N * H
    y_pred = h_relu.mm(w2) # N * D_out

    # compute loss
    loss = (y_pred - y).pow(2).sum().item()
    if it % 50 == 0:
        print(it, loss)

    # Backward pass
    # compute the gradient
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)

    # update weights of w1 and w2
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

$$h = XW_1$$

$$h_{\text{relu}} = \max(0, h)$$

$$Y_{\text{pred}} = h_{\text{relu}} W_2$$

$$f = ||Y - Y_{\text{pred}}||_F^2$$

从Numpy到PyTorch各种两层全连接实现

方案三：PyTorch：Tensor和Autograd 实现

```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
```

```
for it in range(501):
    # Forward pass
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # compute loss
    loss = (y_pred - y).pow(2).sum()
    if it % 50 == 0:
        print(it, loss.item())

    # Backward pass
    loss.backward()

    # update weights of w1 and w2
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

从Numpy到PyTorch各种两层全连接实现

北京大学 PEKING UNIVERSITY

方案四：PyTorch：Tensors 和 Optim 实现

```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
optimizer = torch.optim.SGD([w1, w2], lr=learning_rate)
```

```
for it in range(501):
    # Forward pass
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # compute loss
    loss = (y_pred - y).pow(2).sum()
    if it % 50 == 0:
        print(it, loss.item())

    # Backward pass
    loss.backward()

    # update weights of w1 and w2
    # with torch.no_grad():
    #     w1 -= learning_rate * w1.grad
    #     w2 -= learning_rate * w2.grad
    #     w1.grad.zero_()
    #     w2.grad.zero_()
    optimizer.step()
    optimizer.zero_grad()
```

从Numpy到PyTorch各种两层全连接实现

方案五：PyTorch：Tensors 和 nn.MSELoss 实现

```
import torch
N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
optimizer = torch.optim.SGD([w1, w2], lr=learning_rate)
loss_fn = nn.MSELoss(reduction='sum')
```

```
for it in range(501):
    # Forward pass
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # compute loss
    # loss = (y_pred - y).pow(2).sum()
    loss = loss_fn(y_pred, y)
    if it % 50 == 0:
        print(it, loss.item())

    # Backward pass
    loss.backward()

    # update weights of w1 and w2
    optimizer.step()
    optimizer.zero_grad()
```

从Numpy到PyTorch各种两层全连接实现

方案六：PyTorch：nn 实现

```
import torch.nn as nn

N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H, bias=True),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out, bias=True),
)

torch.nn.init.normal_(model[0].weight)
torch.nn.init.normal_(model[2].weight)
```

```
# model = model.cuda()
loss_fn = nn.MSELoss(reduction='sum')
learning_rate = 1e-6

for it in range(501):
    # Forward pass
    y_pred = model(x) # model.forward()

    # compute loss
    loss = loss_fn(y_pred, y) # computation graph

    if it % 50 == 0:
        print(it, loss.item())

    # Backward pass
    loss.backward()

    # update weights of w1 and w2
    with torch.no_grad():
        for param in model.parameters(): # param (tensor, grad)
            param -= learning_rate * param.grad

    model.zero_grad()
```

从Numpy到PyTorch各种两层全连接实现

方案七：PyTorch：nn 和 Optim 实现

```
import torch.nn as nn

N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H, bias=False),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out, bias=False),
)

torch.nn.init.normal_(model[0].weight)
torch.nn.init.normal_(model[2].weight)
# model = model.cuda()
```

```
loss_fn = nn.MSELoss(reduction='sum')
# learning_rate = 1e-4
# optimizer = torch.optim.Adam(model.parameters(),
lr=learning_rate)

learning_rate = 1e-6
optimizer = torch.optim.SGD(model.parameters(),
lr=learning_rate)

for it in range(501):
    # Forward pass
    y_pred = model(x) # model.forward()

    # compute loss
    loss = loss_fn(y_pred, y) # computation graph
    if it % 50 == 0:
        print(it, loss.item())

    # Backward pass
    loss.backward()

    # update model parameters
    optimizer.step()
    optimizer.zero_grad()
```

从Numpy到PyTorch各种两层全连接实现

方案八：PyTorch：自定义 nn Modules 实现（显式参数）

```
import torch.nn as nn

N, D_in, H, D_out = 64, 1000, 100, 10

x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        # define the model architecture
        self.W1 = nn.Parameter(nn.init.xavier_normal_(torch.Tensor(D_in, H)))
        self.W2 = nn.Parameter(nn.init.xavier_normal_(torch.Tensor(H, D_out)))

    def forward(self, x):
        y_pred = x.mm(self.W1).clamp(min=0).mm(self.W2)
        return y_pred
```

```
model = TwoLayerNet(D_in, H, D_out)
# loss_fn = nn.MSELoss(reduction='sum')
loss_fn = nn.MSELoss()
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
lr=learning_rate)

for it in range(500):
    # Forward pass
    y_pred = model(x) # model.forward()

    # compute loss
    loss = loss_fn(y_pred, y)
    if it % 50 == 0:
        print(it, loss.item())

    # Backward pass
    loss.backward()

    # update model parameters
    optimizer.step()

    optimizer.zero_grad()
```

从Numpy到PyTorch各种两层全连接实现

方案九：PyTorch：自定义 nn Modules 实现（隐式参数）

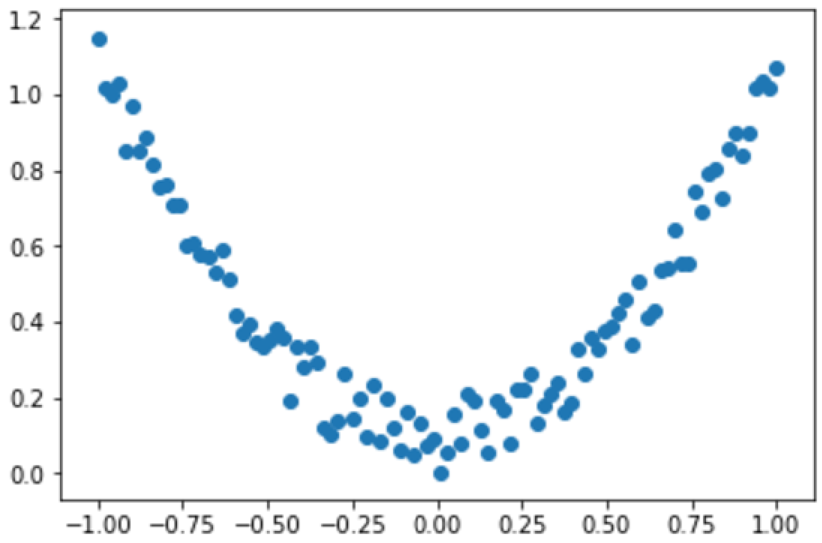
```
import torch.nn as nn
N, D_in, H, D_out = 64, 1000, 100, 10

# 随机创建一些训练数据
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        # define the model architecture
        self.linear1 = torch.nn.Linear(D_in, H, bias=False)
        self.linear2 = torch.nn.Linear(H, D_out, bias=False)

    def forward(self, x):
        y_pred = self.linear2(self.linear1(x).clamp(min=0))
        return y_pred
```

```
model = TwoLayerNet(D_in, H, D_out)
loss_fn = nn.MSELoss(reduction='sum')
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
lr=learning_rate)

for it in range(500):
    # Forward pass
    y_pred = model(x) # model.forward()

    # compute loss
    loss = loss_fn(y_pred, y) # computation graph
    if it % 50 == 0:
        print(it, loss.item())

    # Backward pass
    loss.backward()

    # update model parameters
    optimizer.step()

    optimizer.zero_grad()
```

搭建深度神经网络步骤

- 准备训练数据
- 设计网络架构，构建损失函数
- 批量输入数据，利用反向传播算法训练参数
  - 正向计算损失函数
  - 计算网络参数梯度
  - 利用梯度下降算法更新网络参数

小批量随机梯度下降



$$h = XW_1 + b_1$$
$$h_{\text{sigmoid}} = sigmoid(h)$$
$$Y_{\text{pred}} = h_{\text{sigmoid}} W_2 + b_2$$
$$f = ||Y - Y_{\text{pred}}||_F^2$$

```
torch.manual_seed(1)    # reproducible

x = torch.unsqueeze(torch.linspace(-1, 1, 10000000), dim=1)
y = x.pow(2) + 0.2*torch.rand(x.size())
```

Batch 概念

小批量随机梯度下降

$$f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{x})$$

$$\nabla f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x})$$

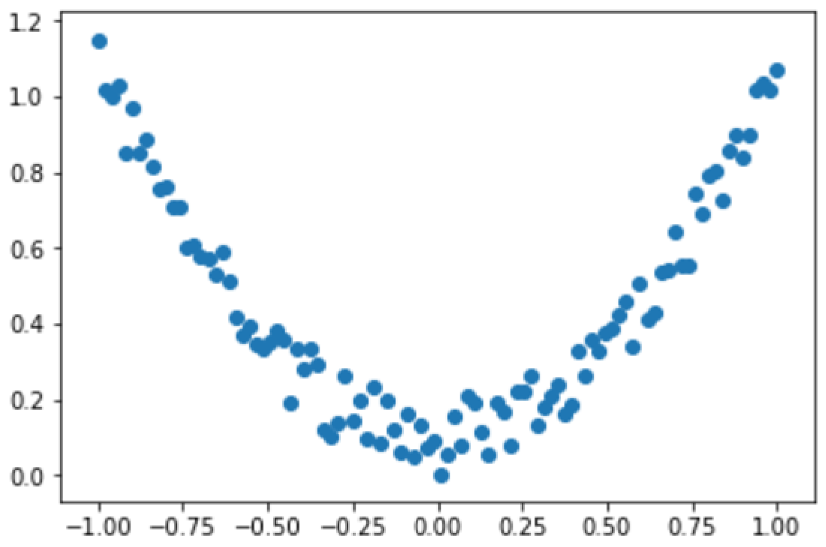
$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f_i(\boldsymbol{x})$$

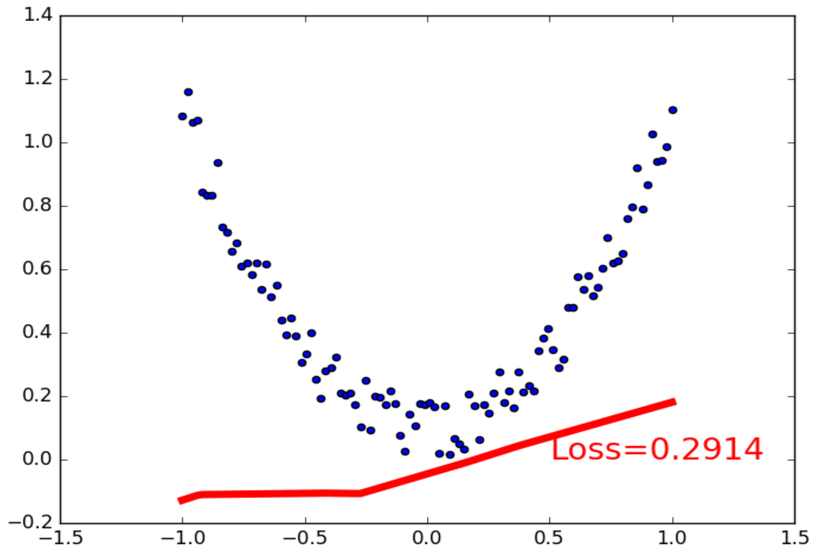
$$\nabla f_{\mathcal{B}}(\boldsymbol{x}) = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \nabla f_i(\boldsymbol{x})$$

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f_{\mathcal{B}}(\boldsymbol{x})$$

W5_Regression_Batch.ipynb

PyTorch搭建两层全连接网络-作业






```
torch.manual_seed(1)    # reproducible

x = torch.unsqueeze(torch.linspace(-1, 1, 100), dim=1)
y = x.pow(2) + 0.2*torch.rand(x.size())
```

1. 补全两层全连接代码 W4_Homework.ipynb
2. 给出变量W1,b1,W2,b2导数表达式

$$h = XW_1 + b_1$$

$$h_{\text{sigmoid}} = sigmoid(h)$$

$$Y_{\text{pred}} = h_{\text{sigmoid}} W_2 + b_2$$

$$f = ||Y - Y_{\text{pred}}||_F^2$$

交流&问题？