智能系统安全实践：手写数字识别

复旦白泽智能

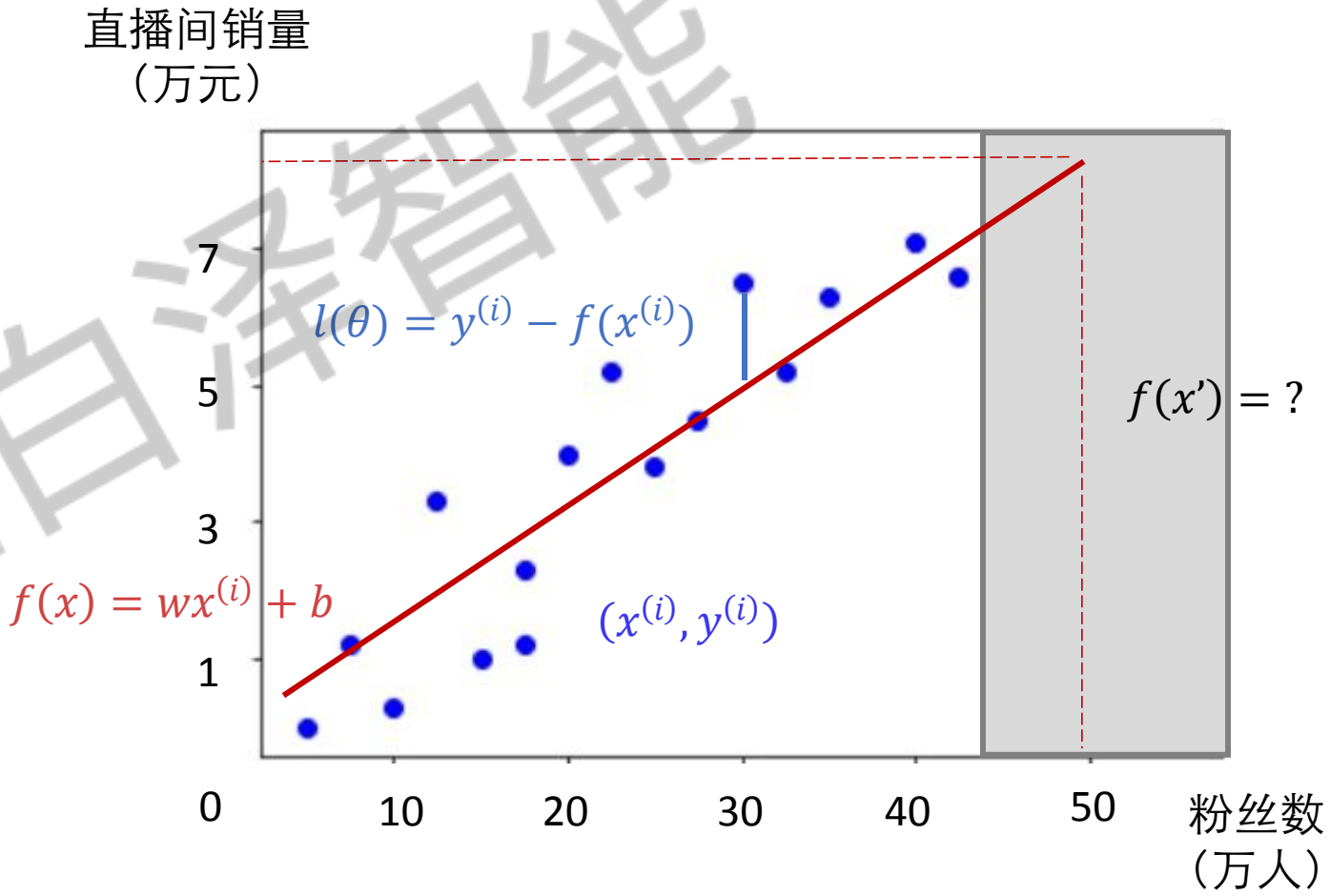
系统软件与安全实验室

大纲

- 理解手写数字识别任务
  - 如何通过有监督学习来解决此任务
- 理解神经网络模型
  - 理解神经网络的基本结构
- 手写数字识别数据集上的实验
  - 基于自动微分实现手写数字识别
  - 基于PyTorch框架实现手写数字识别

有监督学习：回顾

- 有监督学习的过程
  - 准备工作
    - 收集**训练集** $(x^{(i)}, y^{(i)})$
    - 建立**模型** $f(x)$
  - 训练模型
    - 设定**损失函数** $l(\theta)$
    - 通过**梯度下降**最小化 $l(\theta)$
  - 模型预测
    - 在新数据点上计算 $f(x)$



有监督学习：回顾

- 实例: 红酒分类 🍷
  - 准备工作

    收集**训练集** $(x^{(i)}, y^{(i)})$ → 收集每款红酒的属性值和等级

    建立**模型** $f(x)$ → $\boldsymbol{f(x) = Wx + b}$ (多维线性模型)

    p(等级1), p(等级2), ...　　属性1, 属性2, ...
  - 训练模型

    设定**损失函数** $l(\theta)$ → 交叉熵损失函数

    通过**梯度下降**最小化 $l(\theta)$
  - 模型预测

    在新数据点上计算 $f(x)$ → $f_j(x)$最大的j即为预测的等级

属性	值
固定酸度	8.319637
挥发物	0.527821
柠檬酸	0.270976
糖分	2.538806
氯化物	0.087467
游离SO_2	15.874922
总SO_2	46.467792
密度	0.996747
PH值	3.311113
硫酸盐	0.658149
酒精度	10.422983

红酒等级: $\{0, 1, 2\}$

有监督学习：手写数字识别

- 应用: 手写数字识别
  - 准备工作

    收集**训练集** $(x^{(i)}, y^{(i)})$ → **手写数字图像数据集的形式？**

    建立**模型** $f(x)$ → $\boldsymbol{f(x)} = ?$

  - 训练模型

    设定**损失函数** $l(\theta)$ → 交叉熵损失函数

    通过**梯度下降**最小化 $l(\theta)$ → **如何高效更新？**

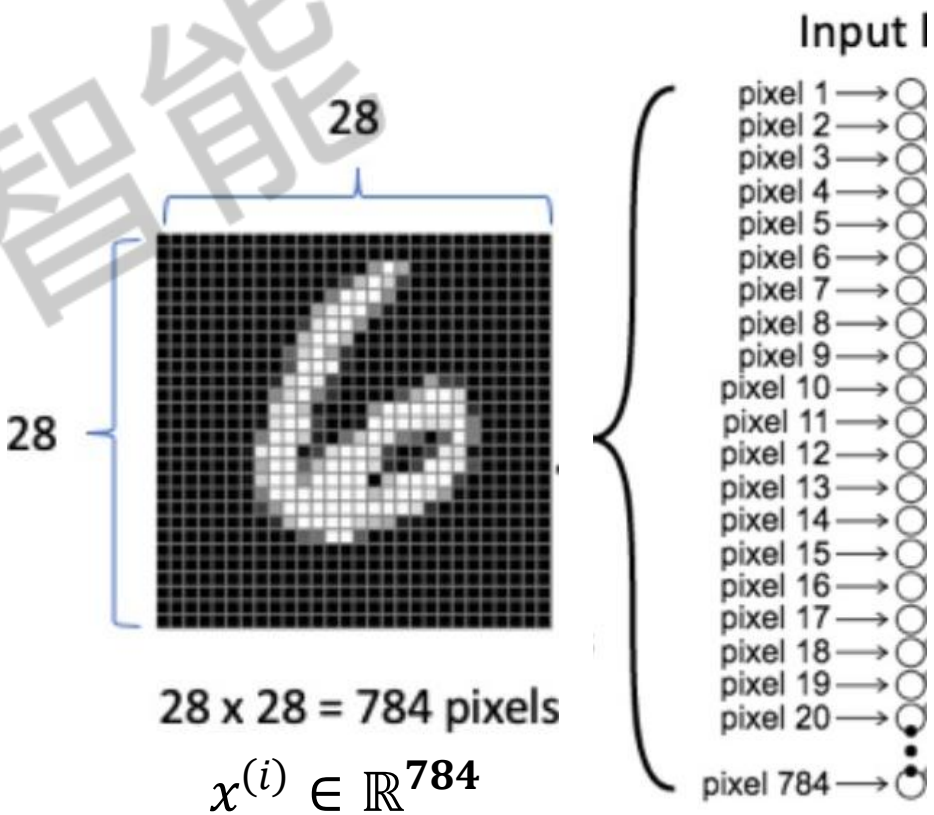
  - 模型预测

    在新数据点上计算 $f(x)$

数字识别: 0-9

手写数字识别：数据集形式

- **手写数字图像数据集**
  - 图像存储为28x28的灰度图
  - 每个像素值0-1，越接近1越白
  - 每个图像 → **784个像素值**
- **输入图像形式**
  - 把28x28的像素值**按行**拼接成一个长向量
  - 每个图像$\boldsymbol{x}^{(i)} \in \mathbb{R}^{784}$，第$k$个维度$x_k^{(i)} \in [0,1]$
- **标签形式**
  - $y^{(i)} \in \{0,1,2,3,4,5,6,7,8,9\}$



$x^{(i)} \in \mathbb{R}^{784}$

$y^{(i)} = 6$

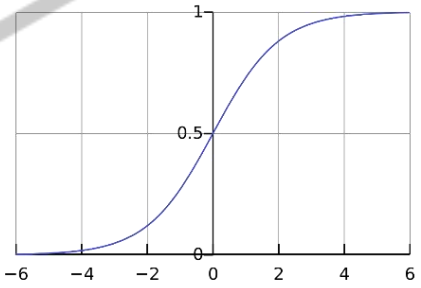
手写数字识别：多层神经网络(MLP)

- 模型定义(以3层为例)
  - 输入层 - $x \in \mathbb{R}^{784}$
  - 隐藏层 - $h \in \mathbb{R}^{100}$ (包含100个神经元)

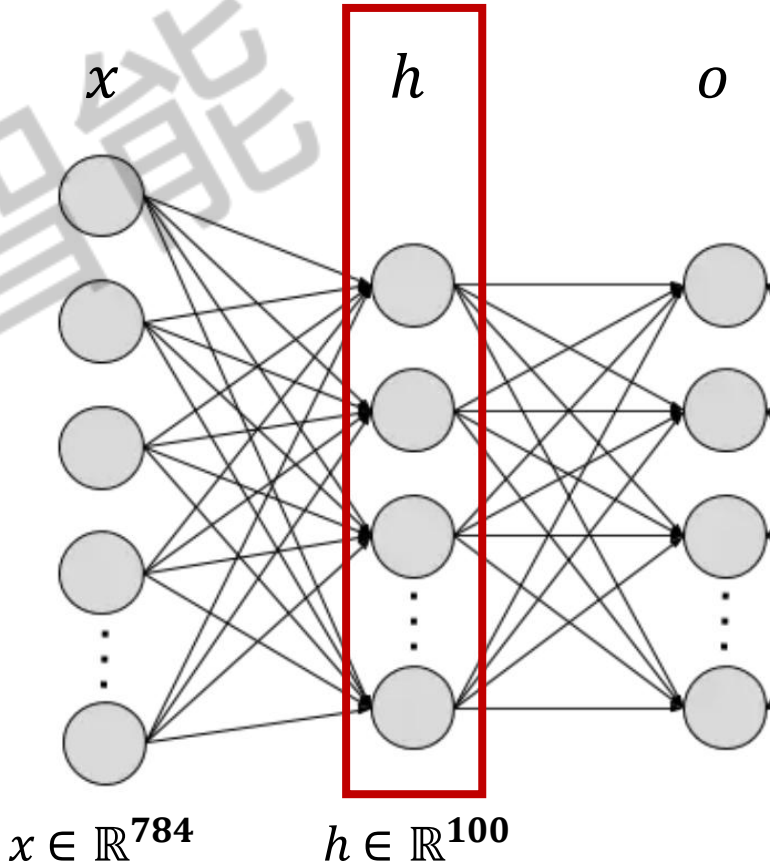
$$h = \sigma(W_h x + b_h)$$，其中$\sigma(h) = \frac{1}{1+e^{-h}}$

$W_h \in \mathbb{R}^{100\times 784}$

$b_h \in \mathbb{R}^{100}$



激活函数不改变维度



手写数字识别：多层神经网络(MLP)

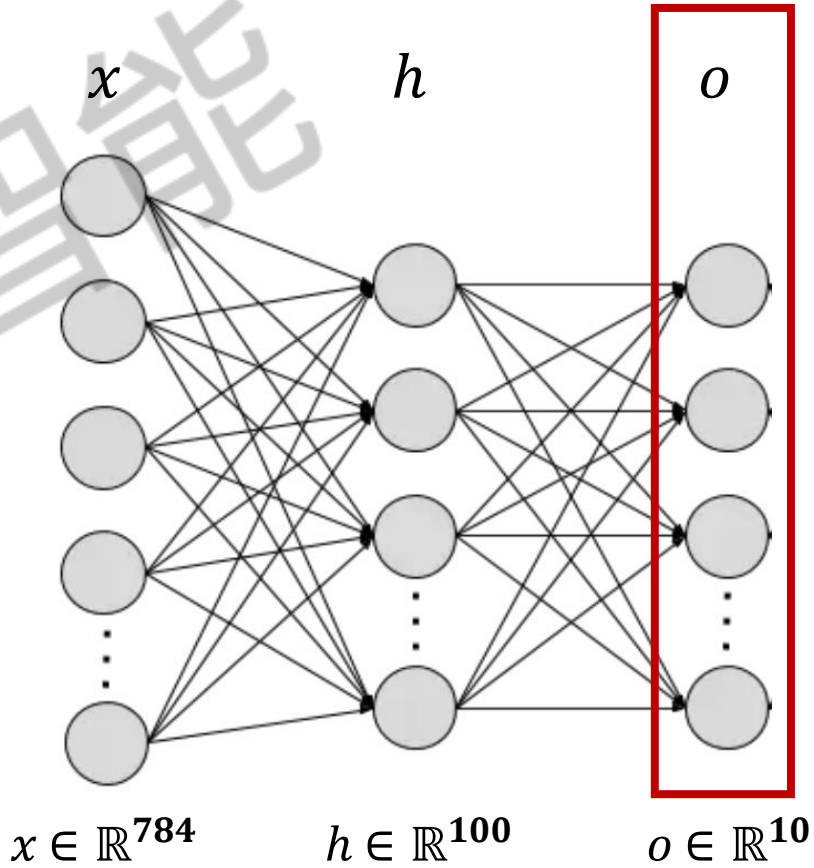
- **模型定义**(以3层为例)
  - 输入层 - $x \in \mathbb{R}^{784}$
  - 隐藏层 - $h \in \mathbb{R}^{100}$ (包含100个神经元)
  - 输出层 - $o \in \mathbb{R}^{10}$

$$\boldsymbol{W_o} \in \mathbb{R}^{10\times 100}$$

$$\boldsymbol{o} = \boldsymbol{W_o h} + \boldsymbol{b_o}$$

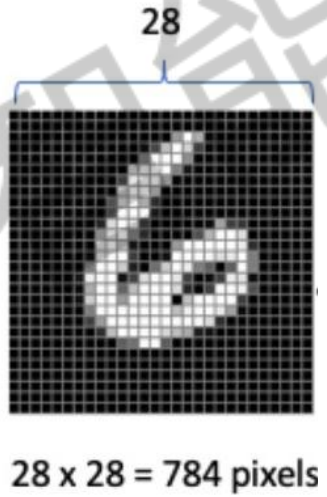
$$\boldsymbol{b_o} \in \mathbb{R}^{10}$$

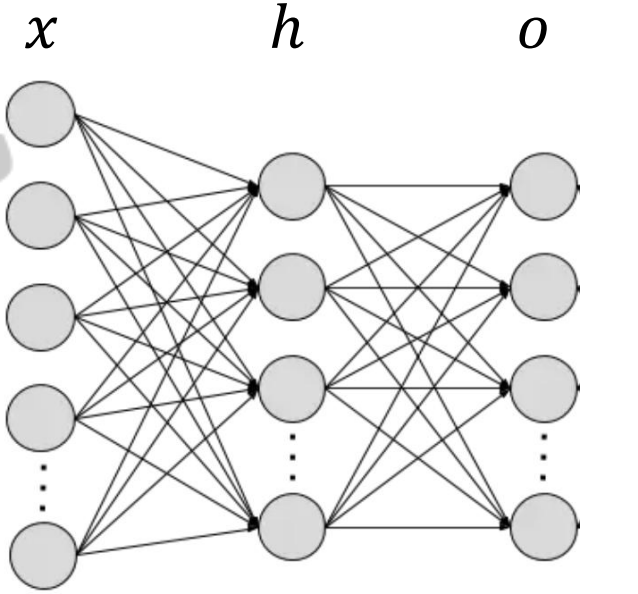
o_j表示模型认为图像是数字j的置信度



手写数字识别：损失函数

- 模型定义
  - $\boldsymbol{h} = \boldsymbol{\sigma}(\boldsymbol{W_h x} + \boldsymbol{b_h})$
  - $\boldsymbol{o} = \boldsymbol{W_o h} + \boldsymbol{b_o}$





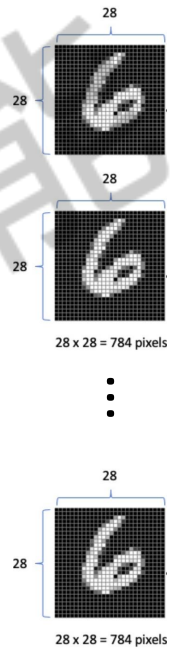
- 损失函数
  - 把标签$y^{(i)}$转换为one-hot形式$\tilde{y}^{(i)}$
  - 把预测结果$f(x^{(i)})$通过Softmax变成$\tilde{f}(x^{(i)})$
  - 计算$\tilde{f}(x^{(i)})$与$\tilde{y}^{(i)}$的交叉熵损失函数 $\ell(\theta) = \sum_{i=1}^{N} CE(f(x^{(i)}), y^{(i)})$

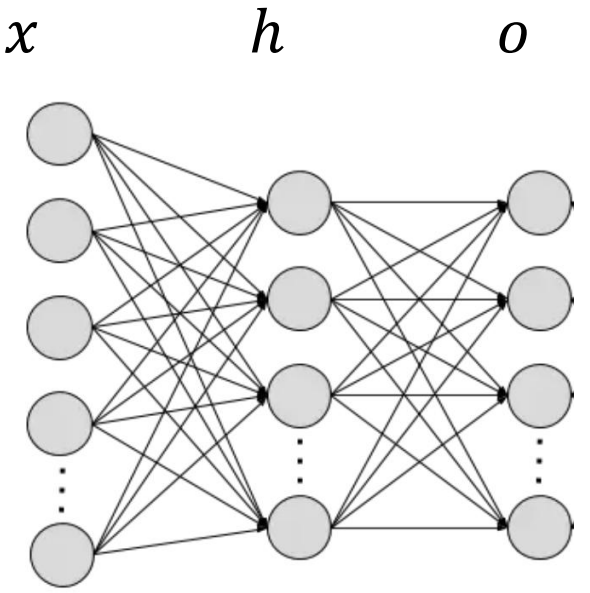
手写数字识别：模型训练

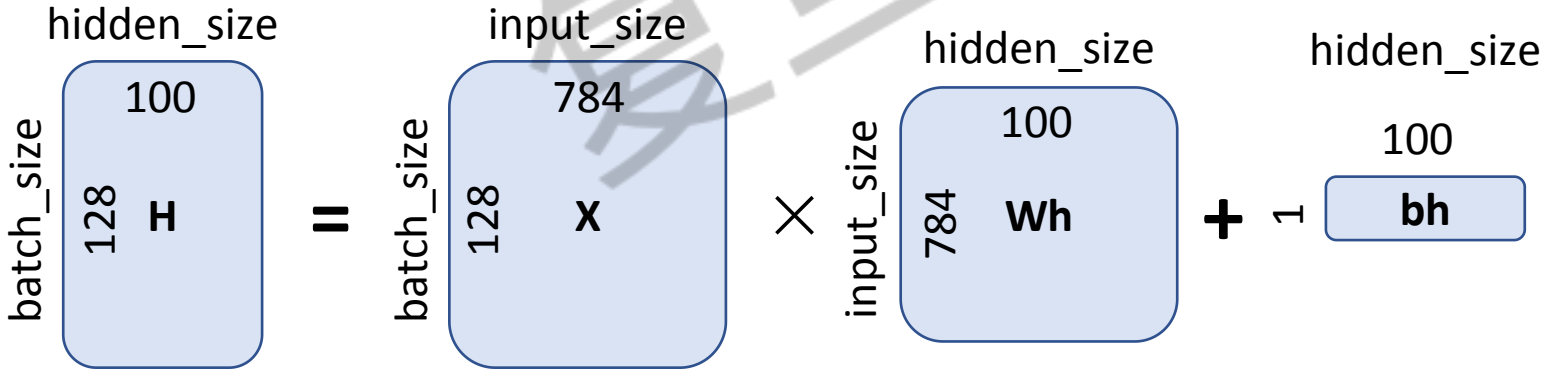
- **模型定义**
  - $\boldsymbol{h} = \sigma(\boldsymbol{W}_h \boldsymbol{x} + \boldsymbol{b}_h)$
  - $\boldsymbol{o} = \boldsymbol{W}_o \boldsymbol{h} + \boldsymbol{b}_o$
- **模型训练**
  - 通过**梯度下降**训练参数$\boldsymbol{W}_h, \boldsymbol{b}_h, \boldsymbol{W}_o, \boldsymbol{b}_o$
  - 计算损失函数 → 计算参数的偏导数 → 更新参数 $\ell(\theta) = \sum_{i=1}^{N} CE(f(x^{(i)}), y^{(i)})$
  - 存在问题: 手写数字识别的数据集中, $N = 60000$
  - 解决方案: 每次随机取一部分数据(batch)计算损失函数，然后更新参数 (Batch Stochastic Gradient Descent, Batch SGD)

手写数字识别：模型训练

- Batch SGD
  - 每次随机取一部分数据(batch)输入模型，进行训练
  - 输入从1x784变成128x784
  - 对整个batch一起计算损失函数，反向更新参数
  - e.g., 隐藏层 $\boldsymbol{h} = \sigma(\boldsymbol{W_h x} + \boldsymbol{b_h})$







有监督学习：手写数字识别

- 回顾整个框架
  - 准备工作

    收集**训练集** $(x^{(i)}, y^{(i)})$ → **图像表示为** $\boldsymbol{x^{(i)}} \in \mathbb{R}^{784}$

    建立**模型** $f(x)$ → **MLP模型结构**

  - 训练模型

    设定**损失函数** $l(\theta)$ → 交叉熵损失函数

    通过**梯度下降**最小化 $l(\theta)$ → **Batch SGD**

  - 模型预测

    在新数据点上计算 $f(x)$

数字识别: 0-9

Q&A

实验部分1

基于自动微分的手写数字识别

手写数字识别

- 解决训练中的问题
  - 前向过程: 矩阵乘法

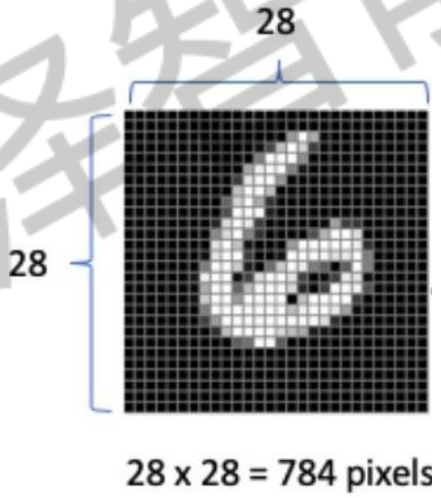
$$\boldsymbol{h} = \sigma(\boldsymbol{W_h x} + \boldsymbol{b_h})$$

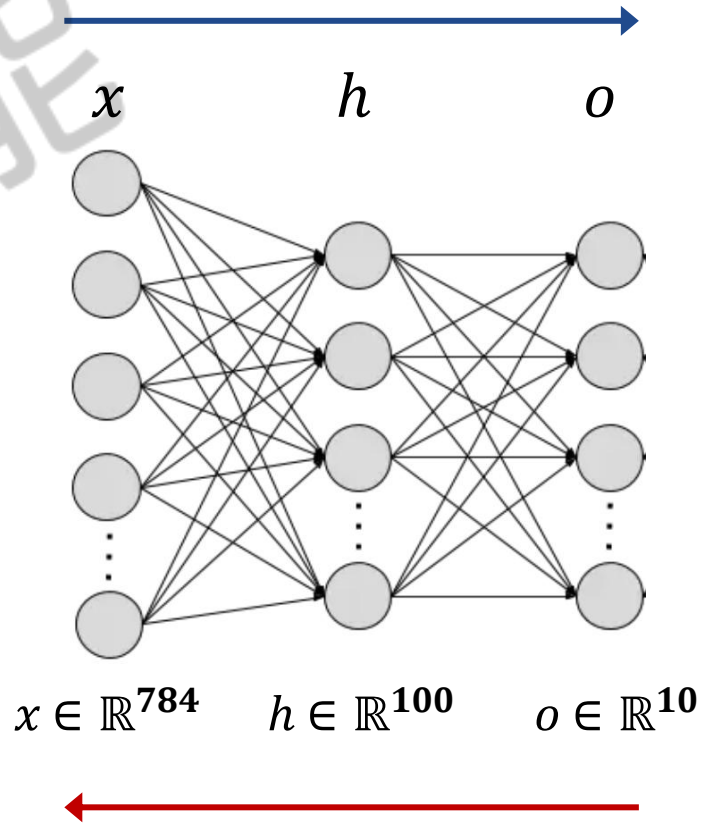
$$\boldsymbol{o} = \boldsymbol{W_o h} + \boldsymbol{b_o}$$

  - 反向传播: 如何对参数求偏导+更新？

$$\boldsymbol{l(\theta)} = \sum_{i=1}^{N} \boldsymbol{CE}(\boldsymbol{f}(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)})$$

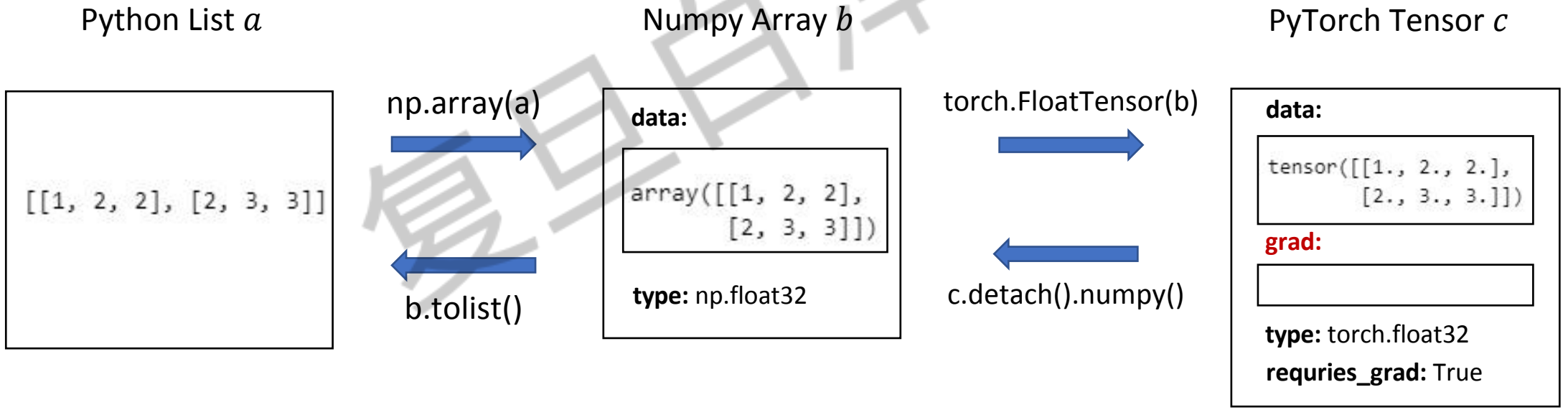
$$\frac{\partial \ell(\boldsymbol{\theta})}{\partial \boldsymbol{W_h}}, \frac{\partial \ell(\boldsymbol{\theta})}{\partial \boldsymbol{b_h}}, \frac{\partial \ell(\boldsymbol{\theta})}{\partial \boldsymbol{W_o}}, \frac{\partial \ell(\boldsymbol{\theta})}{\partial \boldsymbol{b_o}} ?$$





PyTorch：科学计算库

- Pytorch库
  - 对NumPy做了进一步的封装，结合**链式求导**法则，能实现**自动微分**
  - 参数向量以tensor形式存储，pytorch能自动求出梯度



PyTorch：科学计算库

- 自动微分
  - 参数以tensor形式储存
  - 训练时定义损失函数
  - loss.backward()计算每个参数的梯度
  - .grad命令获取某个变量的梯度值

```
import torch
```

```
a = torch.FloatTensor([3.0, 2.0, 3.0])
a.requires_grad = True
a
```

```
tensor([3., 2., 3.], requires_grad=True)
```

```
loss = ((a - 1) **2).sum()
```

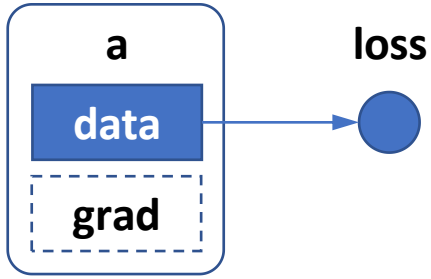
```
loss.backward()
```

```
a.grad
```

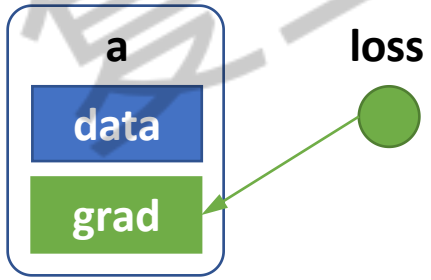
```
tensor([4., 2., 4.])
```

PyTorch： 自动微分

- 正向过程



- 反向传播: 自动微分
  - 调用loss.backward()后




```
import torch
```

```
a = torch.FloatTensor([3.0, 2.0, 3.0])
a.requires_grad = True
a
```

```
tensor([3., 2., 3.], requires_grad=True)
```

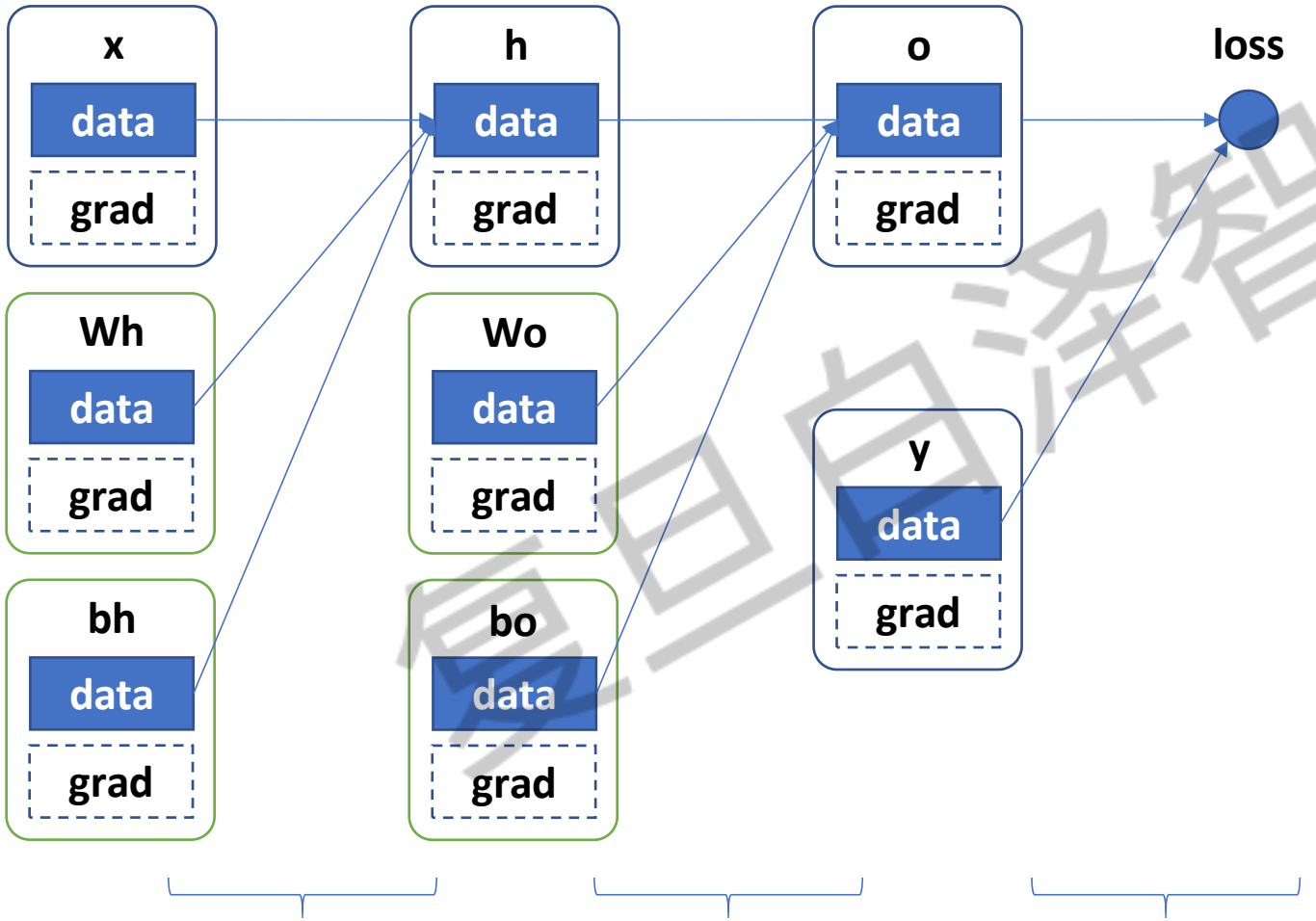
```
loss = ((a - 1) **2).sum()
```

```
loss.backward()
```

```
a.grad
```

```
tensor([4., 2., 4.])
```

PyTorch：自动微分

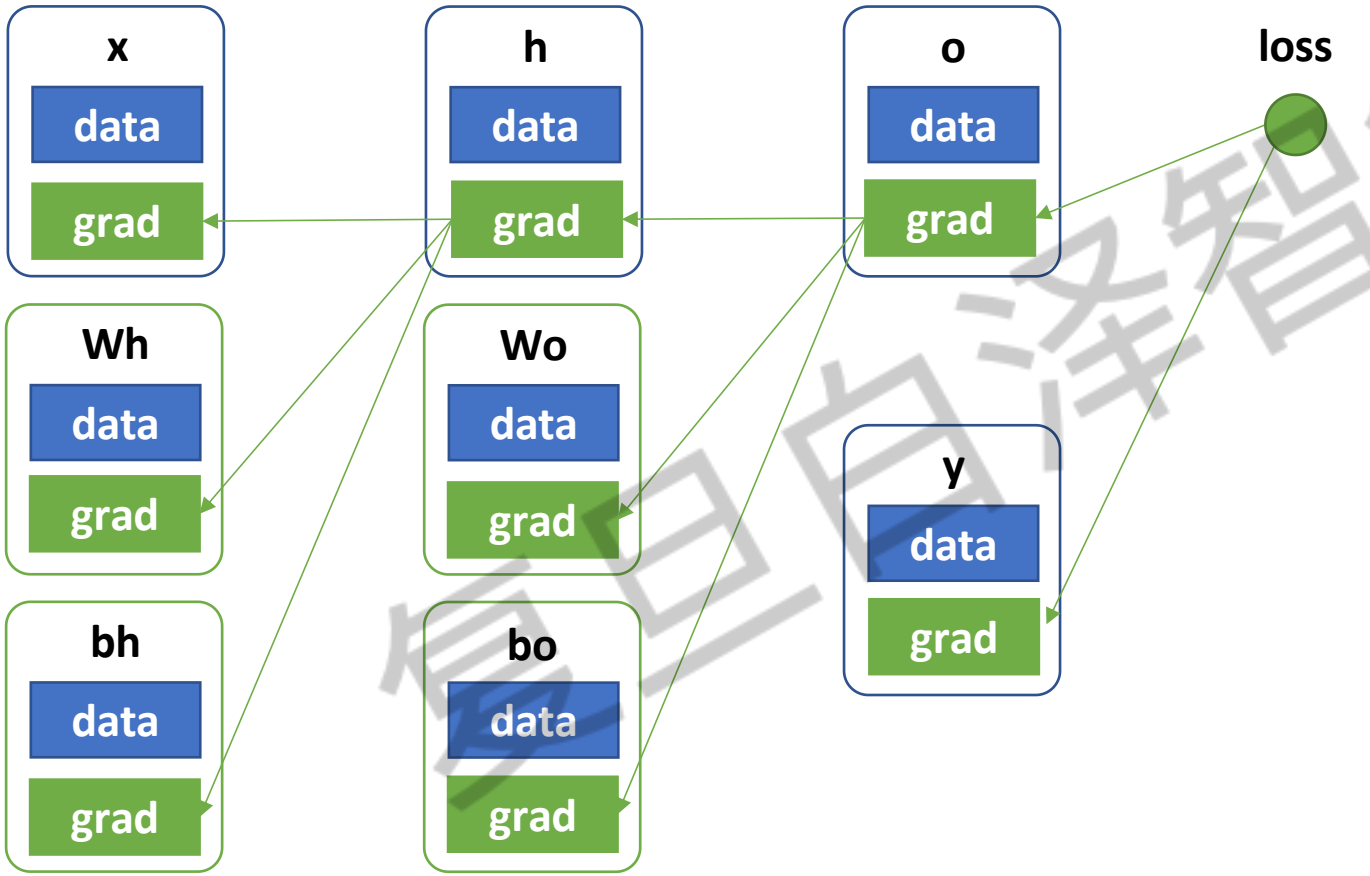



```
Wh = torch.FloatTensor(Wh)
bh = torch.FloatTensor(bh)
Wo = torch.FloatTensor(Wo)
bo = torch.FloatTensor(bo)

Wh.requires_grad = True
bh.requires_grad = True
Wo.requires_grad = True
bo.requires_grad = True
```

```
def forward(x, Wh, bh, Wo, bo):



    return o
```

```
loss_func = torch.nn.CrossEntropyLoss()

loss = loss_func(o, y)
```

PyTorch： 自动微分



① h = x * Wh + bh
隐藏层

② o = h * Wo + bo
输出层

③ loss = $\ell(o, y)$
损失函数

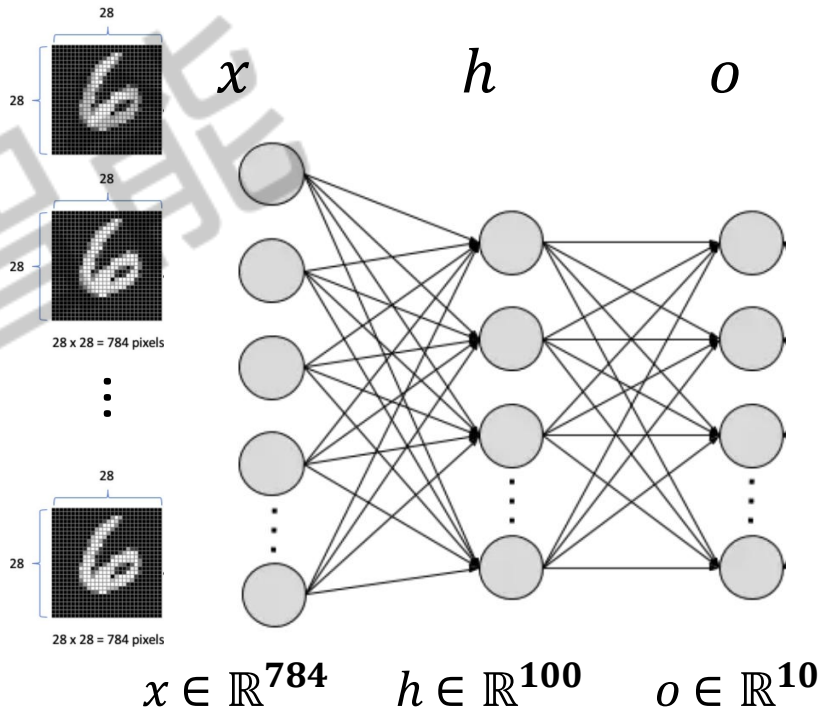
```
Wh = torch.FloatTensor(Wh)
bh = torch.FloatTensor(bh)
Wo = torch.FloatTensor(Wo)
bo = torch.FloatTensor(bo)

Wh.requires_grad = True
bh.requires_grad = True
Wo.requires_grad = True
bo.requires_grad = True
```

```
def forward(x, Wh, bh, Wo, bo):



    return o
```

```
loss.backward()
```

实验任务1：手写数字识别

- **数据形式**
  - 输入一个batch: $x \in \mathbb{R}^{128\times 784}$
  - batch对应的标签: $y = \{0-9\}^{128}$
- **参数初始化**
  - 定义$\boldsymbol{W_h}, \boldsymbol{b_h}, \boldsymbol{W_o}, \boldsymbol{b_o}$并随机初始化
- **前向/反向过程**
  - 根据x, y输出o，计算损失函数
  - 调用loss.backward()计算梯度
  - 利用梯度信息更新参数



实验任务1：手写数字识别

- Week3_Task1_Question.ipynb

  (实验前先解压数据集 MNIST.zip)

- Step1:

  实现模型的正向过程(forward函数)

- Step2:

  调用反向传播，获取参数梯度

  更新模型参数

- Step3:

  在整个数据集上训练模型

  观察准确率

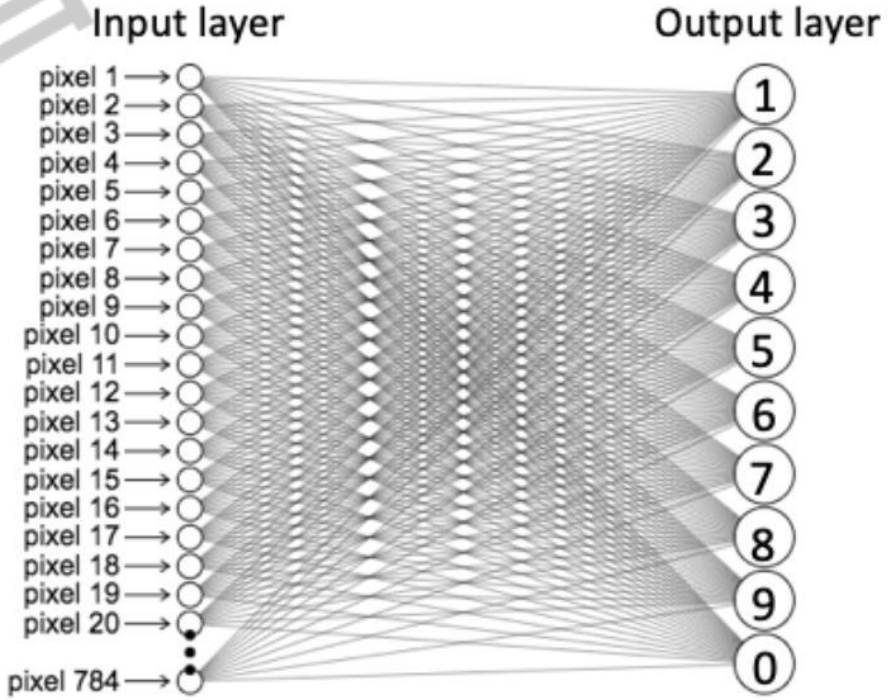
```python
def forward(x, Wh, bh, Wo, bo):

    # TODO: 根据上述函数描述，利用给定的API实现模型前向预测计算

    return o
```

```python
def train(dataloader, lr, Wh, bh, Wo, bo):
    for x, y in dataloader:
        # reshape x from [batch_size, 28, 28] to [batch_size, 784]
        x = x.reshape(-1, 28 * 28)

        # TODO: 根据上述函数描述，利用已经实现好的forward, calc_grad函数，获取参数梯度，实现参数的SGD更新

    return Wh, bh, Wo, bo
```

```python
def predict(x, Wh, bh, Wo, bo):
    # TODO: 根据上述函数描述，利用已经实现好的forward函数和给定的参考API，实现模型的预测

    return prediction


def test(dataloader, Wh, bh, Wo, bo):
    correct = 0
    total = 0
    for x, y in dataloader:
        # reshape x from [batch_size, 28, 28] to [batch_size, 784]
        x = x.reshape(-1, 28 * 28)

        # forward & predict
        prediction = predict(x, Wh, bh, Wo, bo)

        # TODO: 根据上述函数描述，实现模型的预测准确率统计

    return correct / total
```

实验任务1：检查内容

- **默认参数设置**
  - 反向更新时的学习率lr=0.01，迭代轮数epoches=10
  - batch大小=128，对模型参数进行随机初始化
- **检查内容**
  - **默认参数**下的测试集准确度
  - 自行尝试**不同的超参数**，观察结果变化
  - 记录下**3组**不同的(超参数-准确度)数据



实验任务1：Bonus

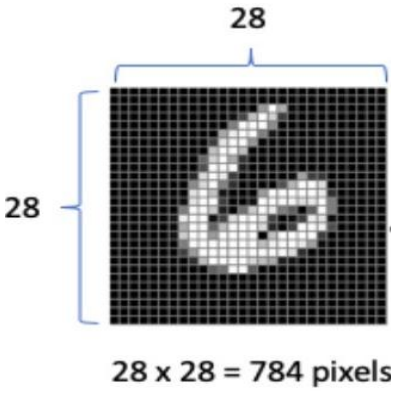
- 梯度计算
  - 代码中使用了Pytorch框架进行自动微分
  - **如何利用链式法则，计算参数W_h, b_h, W_o, b_o的偏导数？**

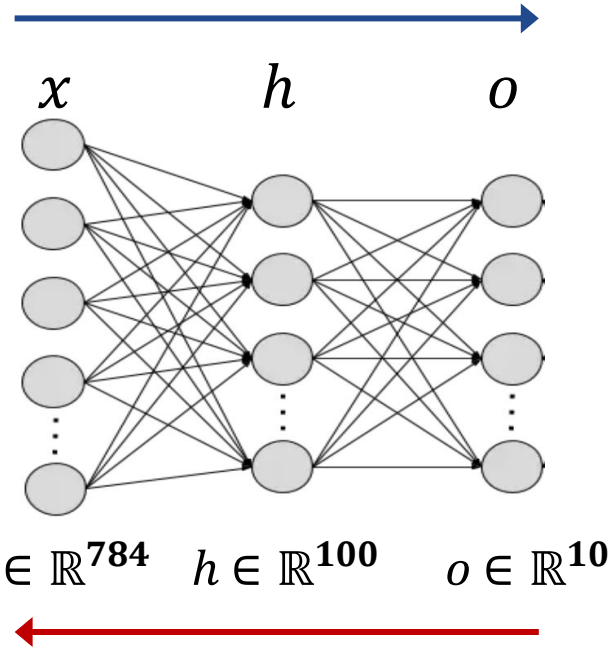
$$h = \sigma(W_h x + b_h)$$

$$o = W_o h + b_o$$

$$l(\theta) = \sum_{i=1}^{N} CE(f(x^{(i)}), y^{(i)})$$

$$\frac{\partial \ell(\theta)}{\partial W_h}, \frac{\partial \ell(\theta)}{\partial b_h}, \frac{\partial \ell(\theta)}{\partial W_o}, \frac{\partial \ell(\theta)}{\partial b_o}?$$





实验任务1：Bonus

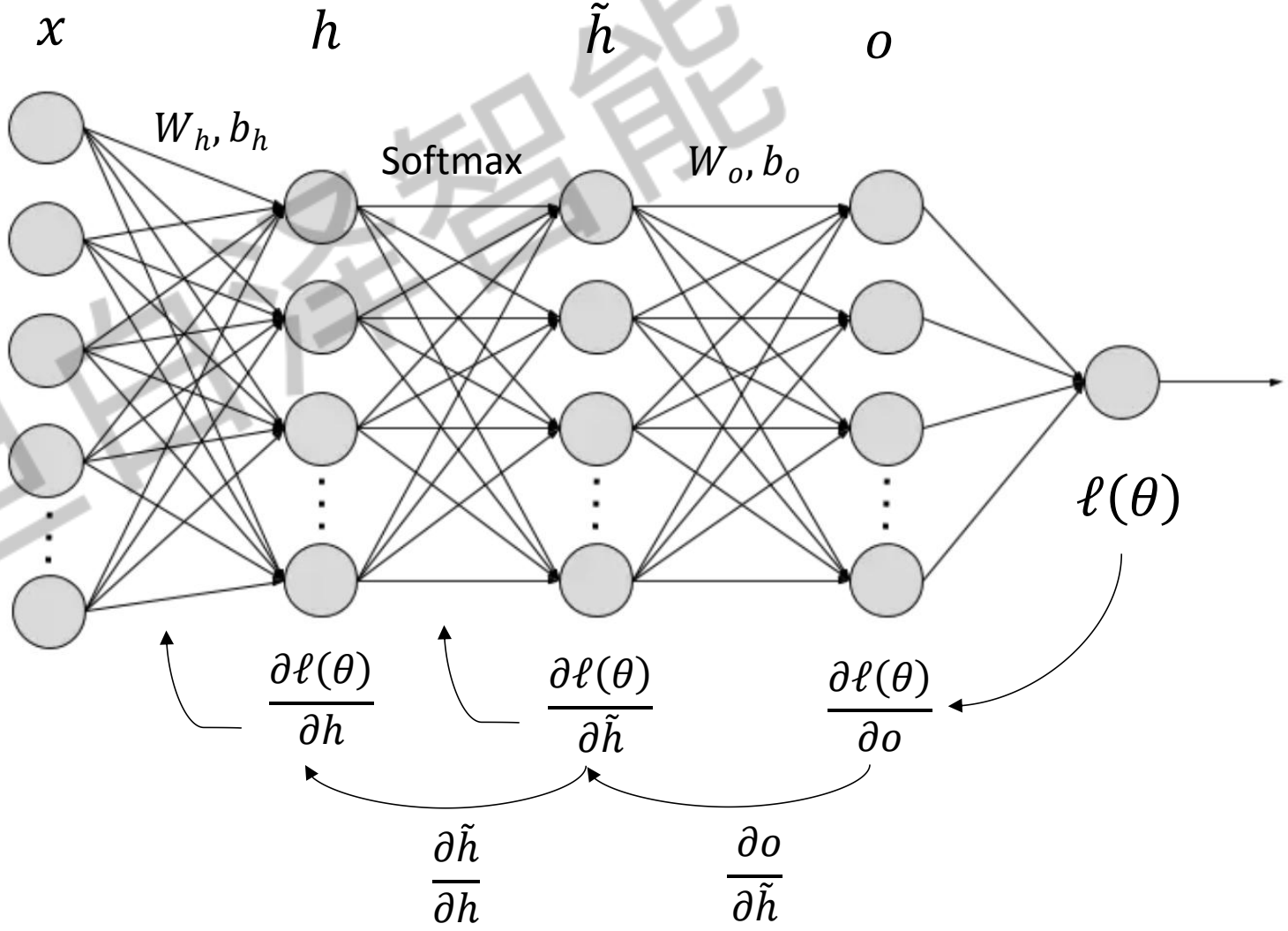
■ 链式法则

$$\frac{\partial \ell(\theta)}{\partial W_o} = \frac{\partial \ell(\theta)}{\partial o} \cdot \frac{\partial o}{\partial W_o}$$

$$\frac{\partial \ell(\theta)}{\partial W_h} = \frac{\partial \ell(\theta)}{\partial h} \cdot \frac{\partial h}{\partial W_h}$$

$$= \frac{\partial \ell(\theta)}{\partial o} \cdot \frac{\partial o}{\partial h} \cdot \frac{\partial h}{\partial W_h}$$

$$= \frac{\partial \ell(\theta)}{\partial o} \cdot \frac{\partial o}{\partial \tilde{h}} \cdot \frac{\partial \tilde{h}}{\partial h} \cdot \frac{\partial h}{\partial W_h}$$



实验任务1：Bonus

- 检查内容
  - 在Week3_Task1_Question.ipynb中自行补充train_handcraft()函数
  - 不使用loss.backward()函数
  - **手动推导W_h, b_h, W_o, b_o的梯度**
  - 通过**矩阵运算**来实现梯度计算

```python
def train_handcraft(x, y, lr):
    global Wh, bh, Wo, bo
    features, prediction = predict(x)
    Wh_grad = torch.zeros(Wh.shape)
    bh_grad = torch.zeros(bh.shape)
    Wo_grad = torch.zeros(Wo.shape)
    bo_grad = torch.zeros(bo.shape)

    for batch in range(batch_size):
        pred = prediction[batch]
        y_vector = np.zeros(bo.shape)
        y_vector[y[batch]] = 1.
        y_vector = torch.Tensor(y_vector)

    # TODO:手动计算当前batch的梯度，并实现对模型参数的梯度更新过程
```

Q&A

实验部分2

基于PyTorch框架的手写数字识别

PyTorch：内置模块

- 除了提供矩阵计算和自动微分，PyTorch还提供**神经网络模块接口**
- **定义神经网络**：继承torch.nn.Module类
  - **重写类函数，实现三个功能**： 定义参数W,b、初始化参数、forward计算
  - **步骤**：
    1. 重写init方法，定义参数和初始化
    2. 重写forward方法，定义前向计算过程
    3. 初始化该类，得到模型实例
    4. 通过model(x)调用forward函数，执行预测

```
class MyLinear(torch.nn.Module):
    def __init__(self, in_dim, out_dim):
        super(MyLinear, self).__init__()
        self.W = ...
        self.b = ...

    def forward(self, x):
        o = torch.mm(self.w, x) + self.b
        return o

linear = MyLinear(764, 100)
o = linear(x)
```

PyTorch：内置模块

- 定义模型时，PyTorch内置了大量计算模块
  - e.g., torch.nn.Linear类 (线性模型)

```
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = torch.nn.Linear(764, 100)
        self.fc2 = ...

    def forward(self, x):
        h = self.fc1(x)
        ...
        return o
```

- torch.nn.Module类还支持嵌套调用
  - 在一个module内使用另一个module
  - 利用上述两点，可以快速搭建模型

- **定义模型后，如何训练模型参数？**

PyTorch：模型优化

- 之前思路 (Task 1)
  - 调用反向传播，获取参数梯度 ⇨ 更新模型参数
  - 通过parameters()函数，可以获取一个Module内所有的模型参数

```
1 mlp = MLP()
2 mlp.parameters()
```

```
<generator object Module.parameters at 0x7f8c678af820>
```

- 利用Optimizer类 (Task 2)
  - PyTorch提供了一种更简洁的方式
  - 如torch.optim.SGD类 (SGD梯度下降策略)
    - PyTorch在类内**帮我们实现了获取参数梯度，并做SGD更新的步骤**

PyTorch：模型优化

■ 定义优化器实现参数更新

■ Optimizer类用法

1. 初始化torch.optim.SGD实例
2. 输入模型参数、定义的学习速率
3. 在每一步更新中：

   先用optimizer.zero_grad()清除上一步的梯度信息

   对梯度进行反向传播，loss.backward()

   调用optimizer.step()完成参数更新

```
optimizer = torch.optim.SGD(mlp.parameters(), lr=lr)

for x, y in dataloader:
    optimizer.zero_grad()

    o = model(x)
    loss = loss_func(o, y)
    loss.backward()

    optimizer.step()
```

小结：PyTorch内置模块

- 三个核心类
  - 继承torch.nn.Module，实现自定义的神经网络模块
  - 初始化torch.nn.loss，用于计算损失函数
  - 初始化torch.optim.SGD类，用于自动SGD更新

- 训练步骤
  - 通过model(x)获取模型输出
  - 计算损失函数，并使用backward()计算梯度
  - 利用optimizer.step()做更新

小结：MLP模型的训练

	NumPy实现	PyTorch自动微分	PyTorch内置模块

模型参数定义	`Wh = np.array(...)`
	`...`
		`Wh = torch.FloatTensor(...)`
		`...`
		`Wh.requires_grad = True`
		`...`
			模型定义

模型前向计算	`def forward(x, ...):`
	`  h = np.matmul(Wh, x) + bh`
	`  ...`
	`  return o`
		`def forward(x, ...):`
		`  h = torch.mm(Wh, x) + bh`
		`  ...`
		`  return o`
			`class MLP(torch.nn.Module):`
			`  def __init__(self, ...):`
			`  super(MLP, self).__init__()`
			`  self.fc1 = ...`
			`  def forward(self, x):`
			`  h = self.fc1(x)`
			`  ...`
			`  return o`

损失函数计算	`def loss_func(o, y):`
	`  loss = ...`
	`  return loss`
		内置模块
		`loss_func = torch.nn.CrossEntropyLoss()`
			`loss_func = torch.nn.CrossEntropyLoss()`

模型梯度计算	`def calc_grad(...):`
	`  loss = ...`
	`  dWh = ...`
	`  ...`
		自动微分
		`def calc_grad(...):`
		`  loss = loss_func(o, y)`
		`  loss.backward()`
		`  dWh = Wh.grad`
		`  ...`
			`mlp = MLP()`
			`optimizer = torch.optim.SGD(mlp.parameters(), lr=lr)`
			`mlp.train()`
			`for x, y in dataloader:`
			`  optimizer.zero_grad()`
			`  o = model(x)`
			`  loss = loss_func(o, y)`
			`  loss.backward()`

SGD更新	`for x, y in dataloader:`
	`  o = forward(x, ...)`
	`  dWh, .. = calc_grad(...)`
	`  Wh -= lr * dWh`
	`  ...`
		`for x, y in dataloader:`
		`  o = forward(x, Wh, bh, Wo, bo)`
		`  dWh, ... = calc_grad(...)`
		`  Wh.data -= lr * dWh`
		`  ...`
			通过step()
			完成SGD

实验任务2：基于Pytorch框架的手写数字识别

- 实现一个4层神经网络
  - Week3_Task2_Question.ipynb
    - 完成init函数中的模型定义
    - 完成forward函数中的正向过程
    - 完成训练过程中的反向梯度传播、参数更新
    - 训练模型，观察准确度，与Task1的结果比较
- 额外尝试
  - 查找PyTorch的API
  - 用torch.optim中其他优化器完成训练

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        # TODO: 定义上文要求的MLP模型结构

    def forward(self, x):
        # TODO: 定义MLP模型的前向过程

        return o
```

```python
for e in range(epoches):
    t = tqdm(train_loader)
    for img, label in t:
        # Forward img and compute loss
        pred = mlp(img)
        loss = criterion(pred, label)

        # TODO: 基于优化器的使用方法，完成反向梯度传播、参数更新

        t.set_postfix(epoch=e, train_loss=loss.item())
```

Q&A