黄冈师范学院
HUANGGANG NORMAL UNIVERSITY

人工智能与机器学习

Artificial Intelligence and Machine Learning

章节：实验8-神经网络的NumPy实现

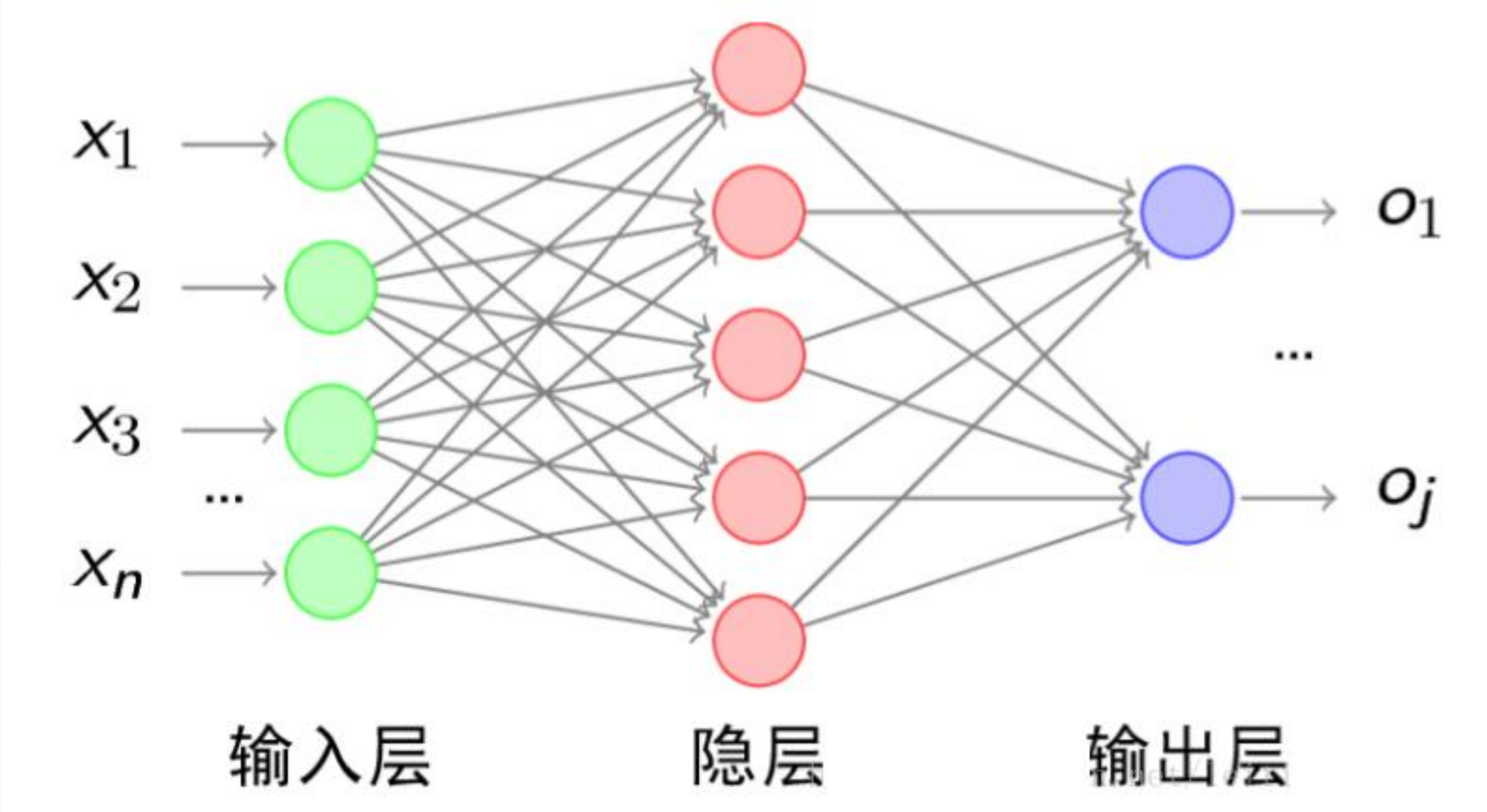
教师：刘重

学院：计算机学院

厚德 博学 力行 致远

一、实验目的

- (1) 巩固神经网络的结构和原理；
- (2) 巩固神经网络的训练方法；
- (3) 练习神经网络的NumPy实现。

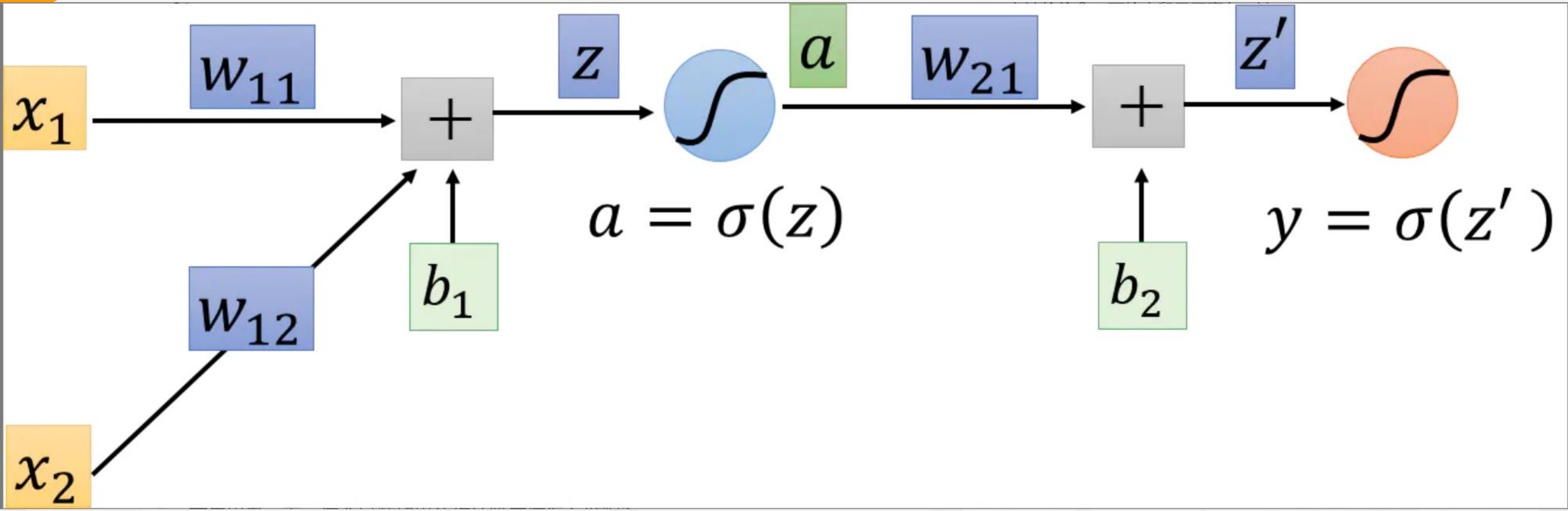


一、神经网络的反向传播算法原理

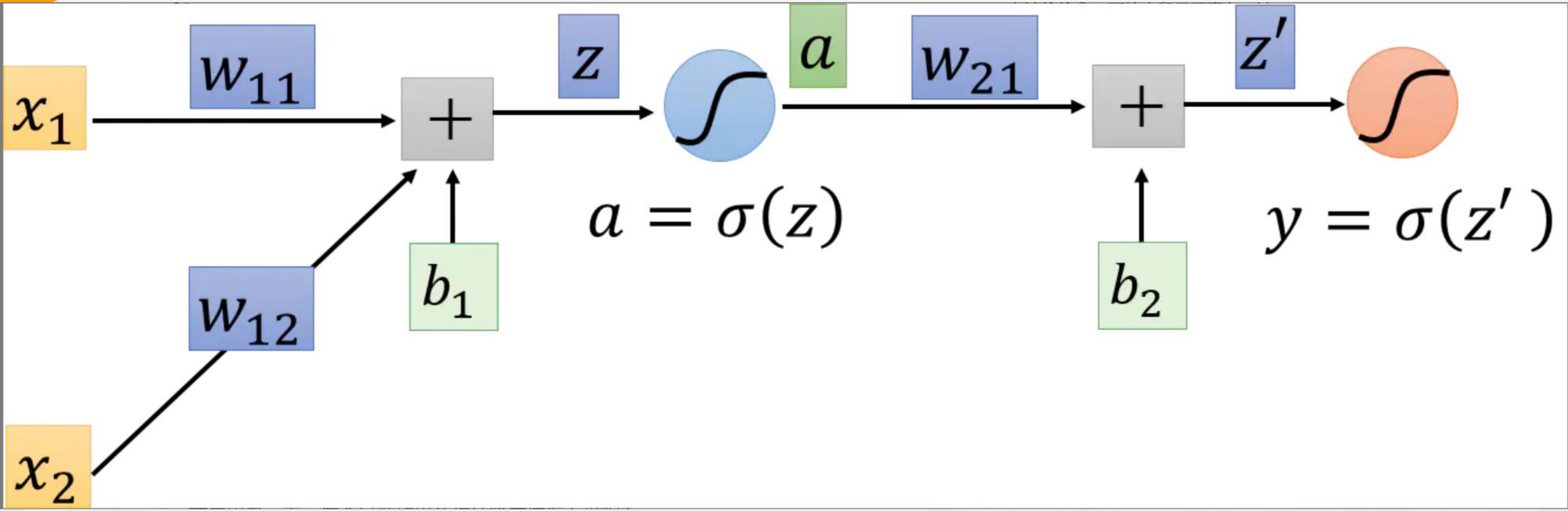
二、基于NumPy的神经网络的实现

1、定义网络结构；2、初始化模型参数；3、定义前向传播过程；4、计算当前损失；5、执行反向传播；6、权重更新函数；7、神经网络模型封装；8、生成模拟数据集；9、模型训练；10、结果的可视化呈现

三、实验结果分析



输入样本为一个二维的向量$\vec{x}$，输出为一个数y，现在假设loss函数是$l = ||(y - \hat{y})||_2^2$，那么要求的便是$\frac{\partial l}{\partial w_i}$以及$\frac{\partial l}{\partial b}$（把$w$和$b$合并成一个向量$W$也是可以的）。



先只看第一层神经元的参数更新（其实对于任意层任意神经元都有下式的关系）：

$$\frac{\partial l}{\partial w} = \frac{\partial l}{\partial z}\frac{\partial z}{\partial w}$$

接下来分别计算$\frac{\partial l}{\partial z}$和$\frac{\partial z}{\partial w}$:

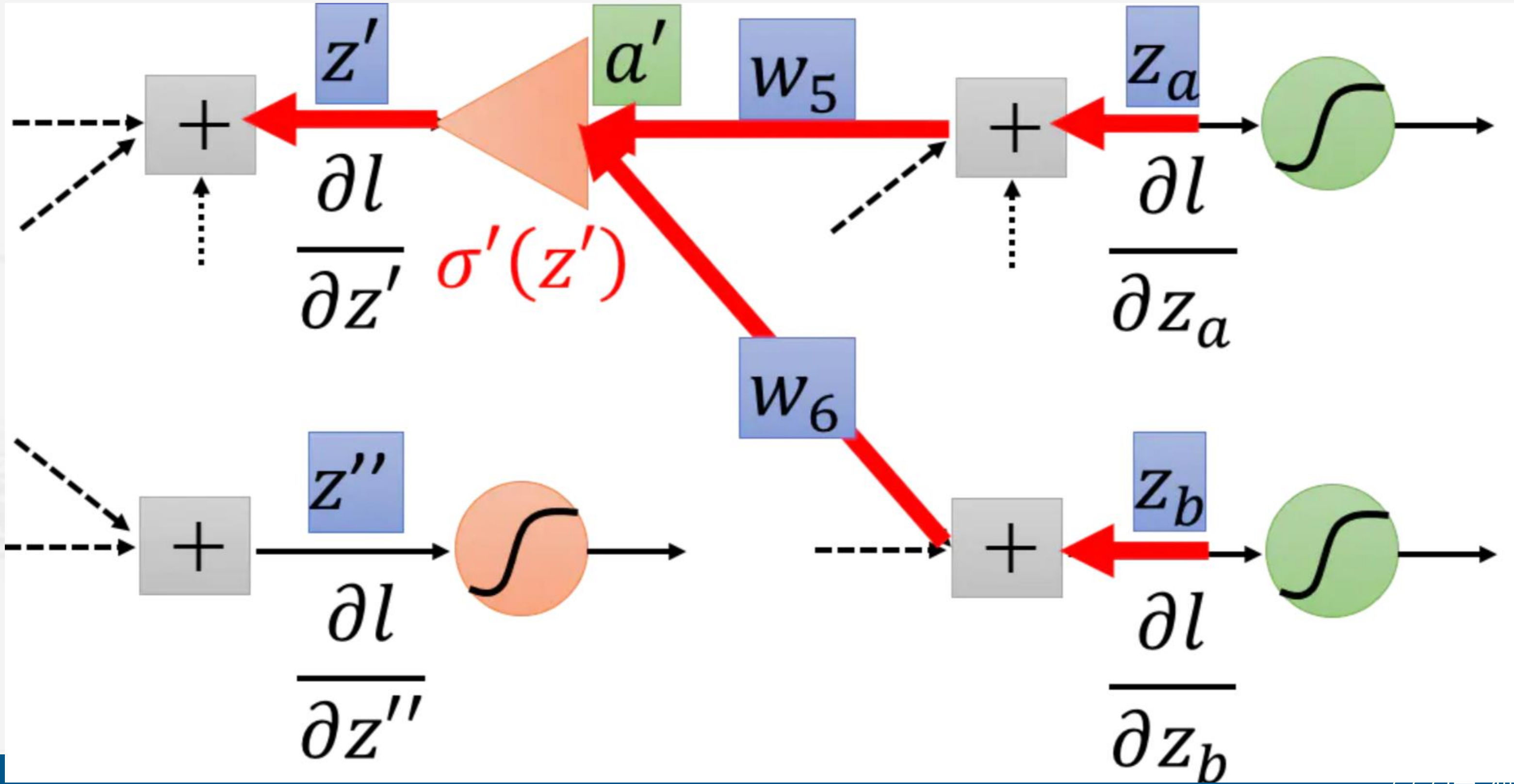
（1）正向计算：$\frac{\partial z}{\partial w_{11}} = x_1$，$\frac{\partial z}{\partial w_{12}} = x_2$，可以看出正向计算有个非常好的性质，$x_1, x_2$就是神经元的值，这都已经在正向传播时计算过了！

（2）反向计算：$\frac{\partial l}{\partial z} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial z}$。这其中，$\frac{\partial a}{\partial z}$很简单，$sigmiod$求导就很容易了，而且$\sigma(x)' = \sigma(x)[1-\sigma(x)]$，又可以用现成的结果了。

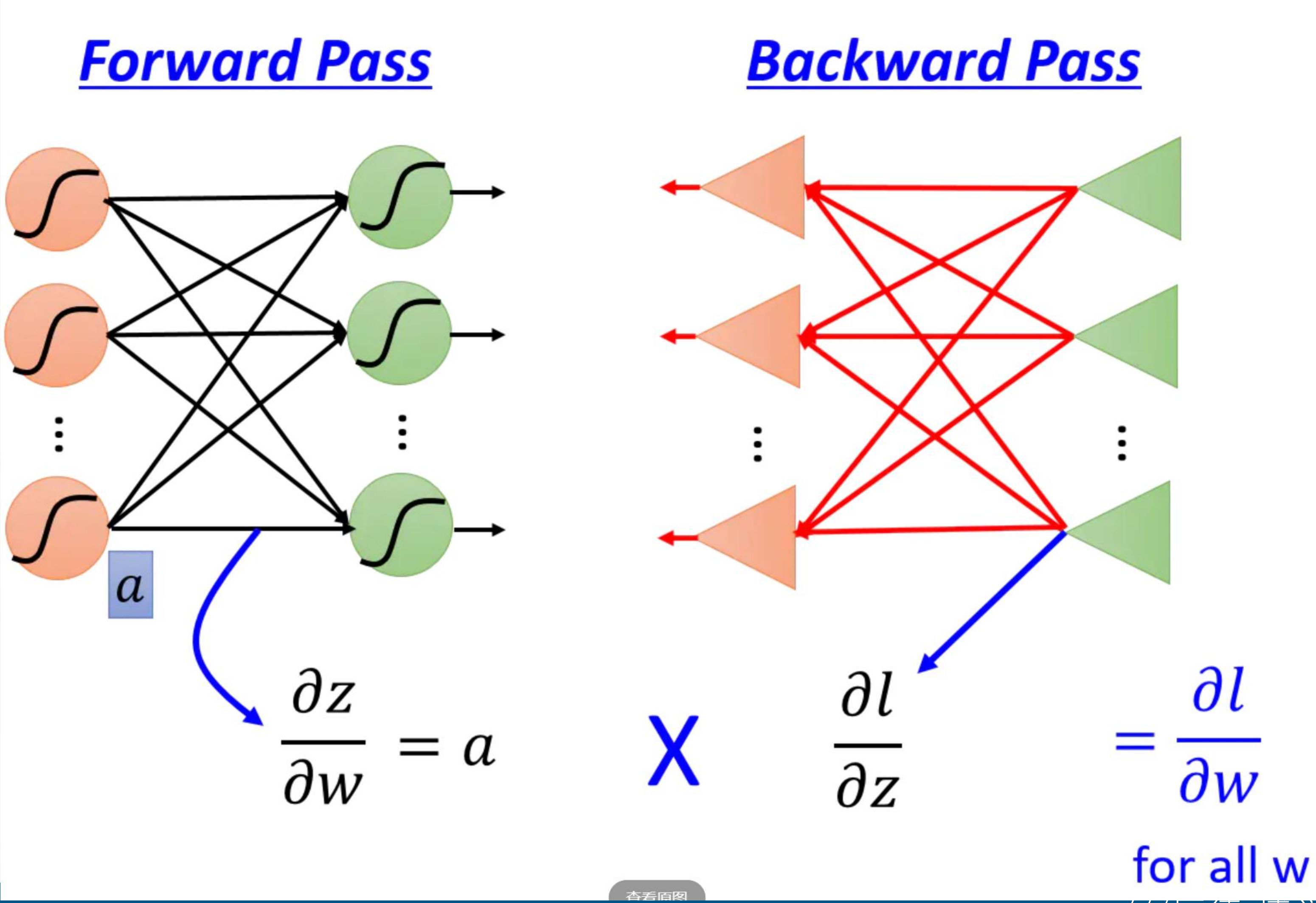
（3）而对于$\frac{\partial l}{\partial a}$的计算比较棘手了，因为：$\frac{\partial l}{\partial a} = \frac{\partial l}{\partial z\prime}\frac{\partial z\prime}{\partial a} = \frac{\partial l}{\partial z\prime}\cdot w_{21}$。其中$w_{21}$就是神经元间的连接权重参数，对于比较复杂的网络，计算$\frac{\partial l}{\partial a}$可以进行递归计算。如果被计算的$\frac{\partial l}{\partial z\prime}$是神经网络的最后一层，问题就变的简单了：$\frac{\partial l}{\partial z\prime} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial z\prime}$。

（4）最终，$\frac{\partial l}{\partial z} = \sigma(z)'\left[\frac{\partial l}{\partial z\prime}\cdot w_{21}\right]$。

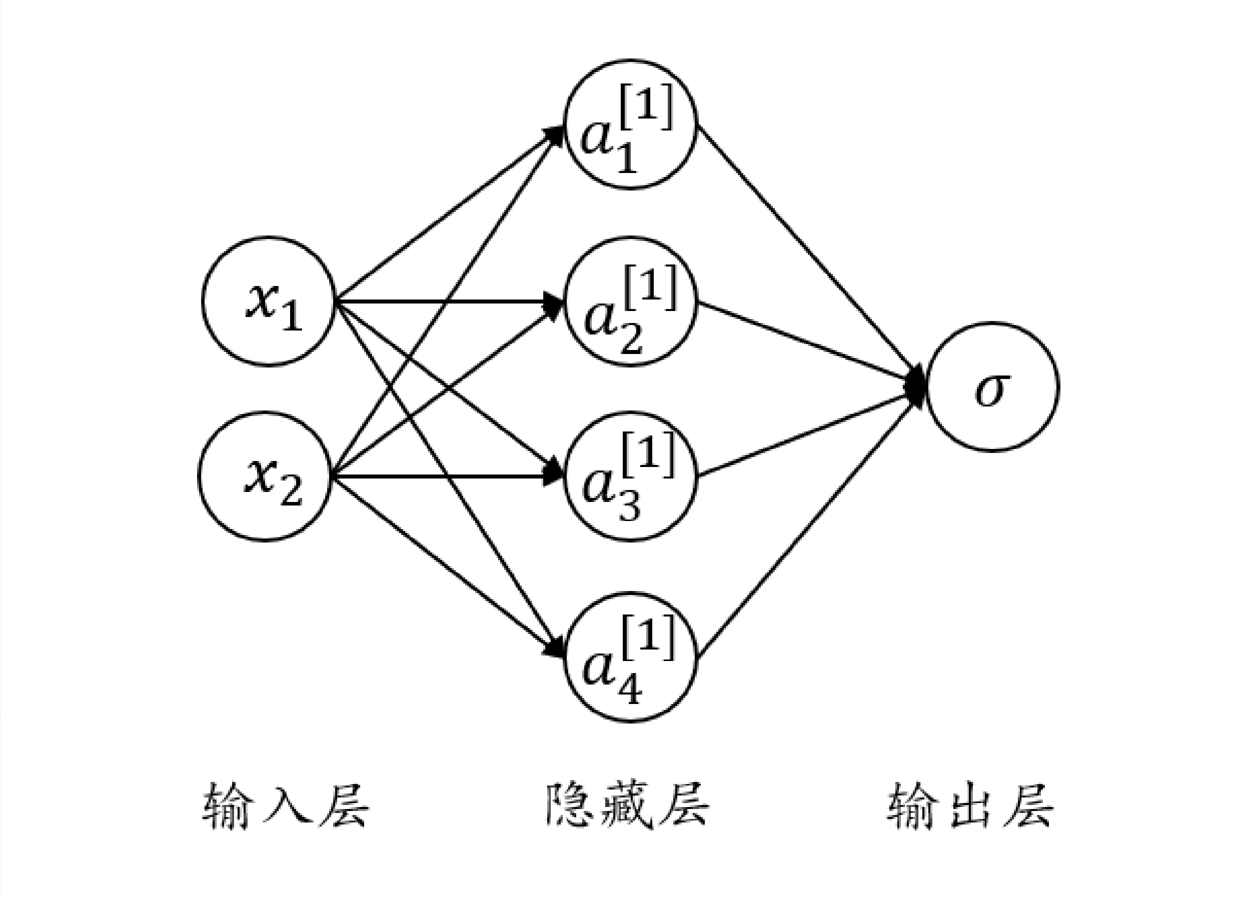
再看一下反向计算 $\frac{\partial l}{\partial z}=\frac{\partial l}{\partial a}\frac{\partial a}{\partial z}$：因为$\frac{\partial a}{\partial z}$比较好求，sigmoid函数在该神经元上的求导就得到了；$\frac{\partial l}{\partial a}$则可以使用递归方法，只要求出output layer的$\frac{\partial l}{\partial z\prime}$即可得到。从下图来看，就是input为$\frac{\partial l}{\partial z\prime}$，然后按照同样的网络反方向计算一次，和正向传播的不同之处仅仅在于sigmoid（非线性转换/激活函数）变成先求导再相乘，结构清晰，计算量也大幅度下降。



- 最后总结一下：正向传播和反向传播 相乘就可以计算出所有的参数更新梯度。



四、问题描述



基于NumPy实现神经网络模型的基本思路包括定义网络结构、初始化模型参数、定义前向传播过程、计算当前损失、执行反向传播、更新权重，以及将全部模块整合成一个完整的神经网络模型。

- （1）定义网络结构。
- 隐藏层有4个神经元，输入输出与具体的训练数据的维度有关

```python
# 定义网路结构
def layer_sizes(X, Y):
    '''
    输入：
    X：训练输入
    y：训练输出
    输出：
    n_x：输入层大小
    n_h：隐藏层大小
    n_y：输出层大小
    '''
    n_x = X.shape[0] # 输入层大小
    n_h = 4 # 隐藏层大小
    n_y = Y.shape[0] # 输出层大小
    return (n_x, n_h, n_y)
```

- （2）初始化模型参数。
- 有了网络结构和大小后，就可以初始化网络权重系数了。
- 假设W1为输入层到隐层的权重数组、b1为输入层到隐层的偏置数组；W2为隐层到输出层的权重数组，b2为隐藏层到输出层的偏置数组

```python
# 初始化神经网络模型参数
def initialize_parameters(n_x, n_h, n_y):
    W1 = np.random.randn(n_h, n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)*0.01
    b2 = np.zeros((n_y, 1))

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

- （3）定义前向传播过程。
- 以tanh函数为隐层激活函数，以sigmoid函数为输出层的激活函数。
- 前向传播计算过程由以下四个公式定义

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1](i)}$$
$$a^{[2](i)} = \tanh(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2](i)}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)})$$

```python
# 定义前向传播过程
def forward_propagation(X, parameters):
    '''
    输入:
    X: 训练输入
    parameters: 初始化模型参数
    输出:
    A2: 模型输出
    caches: 前向传播过程计算的中间值缓存
    '''
    # 获取各参数初始值
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']    
    # 执行前向计算
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)    
    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache
```

- （4）计算当前损失。
- 前向计算输出结果后，需要将其与真实标签做比较，基于损失函数给出当前迭代的损失。基于交叉熵的损失函数定义如下

$$L = -\frac{1}{m}\sum_{i=0}^{m}\left[y^{(i)}\log\left(a^{[2](i)}\right) + \left(1-y^{(i)}\right)log\left(1-a^{[2](i)}\right)\right]$$

```python
# 定义损失函数
def compute_cost(A2, Y, parameters):
    '''
    输入:
    A2: 前向计算输出
    Y: 训练标签
    输出:
    cost: 当前损失
    '''
    # 训练样本量
    m = Y.shape[1]
    # 计算交叉熵损失
    logprobs = np.multiply(np.log(A2),Y) + np.multiply(np.log(1-A2), 1-Y)
    cost = -1/m * np.sum(logprobs)
    # 维度压缩
    cost = np.squeeze(cost)

    assert(isinstance(cost, float))
    return cost
```

- （5）执行反向传播。
- 前向传播和损失计算完成后，神经网络最关键、最核心的部分就是执行反向传播了。损失函数关于各个参数的梯度计算公式如下

$$dz^{[2]} = a^{[2]} - y$$
$$dW^{[2]} = (dz^{[2]})(a^{[1]})^T$$
$$db^{[2]} = dz^{[2]}$$
$$dz^{[1]} = (W^{[2]})^T(dz^{[2]}) * [g^{[1]}(z^{[1]})]$$
$$dW^{[1]} = (dz^{[1]})(x)^T$$
$$db^{[1]} = dz^{[1]}$$

```python
# 定义反向传播过程
def backward_propagation(parameters, cache, X, Y):
    '''
    输入：
    parameters：神经网络参数字典
    cache：神经网络前向计算中间缓存字典
    X：训练输入
    Y：训练输出
    输出：
    grads:权重梯度字典
    '''
    # 样本量
    m = X.shape[1]
    # 获取W1和W2
    W1 = parameters['W1']
    W2 = parameters['W2']
    # 获取A1和A2
    A1 = cache['A1']
    A2 = cache['A2']
    # 执行反向传播（需要在编程之前推导公式）
    dZ2 = A2-Y
    dW2 = 1/m * np.dot(dZ2, A1.T)
    db2 = 1/m * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2)*(1-np.power(A1, 2))
    dW1 = 1/m * np.dot(dZ1, X.T)
    db1 = 1/m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}
    return grads
```

- （6）权重更新函数。
- 反向传播完成后，便可以基于权重梯度来更新权重了。对权重按照负梯度方向不断迭代，也就是梯度下降法，即可一步步达到最优值

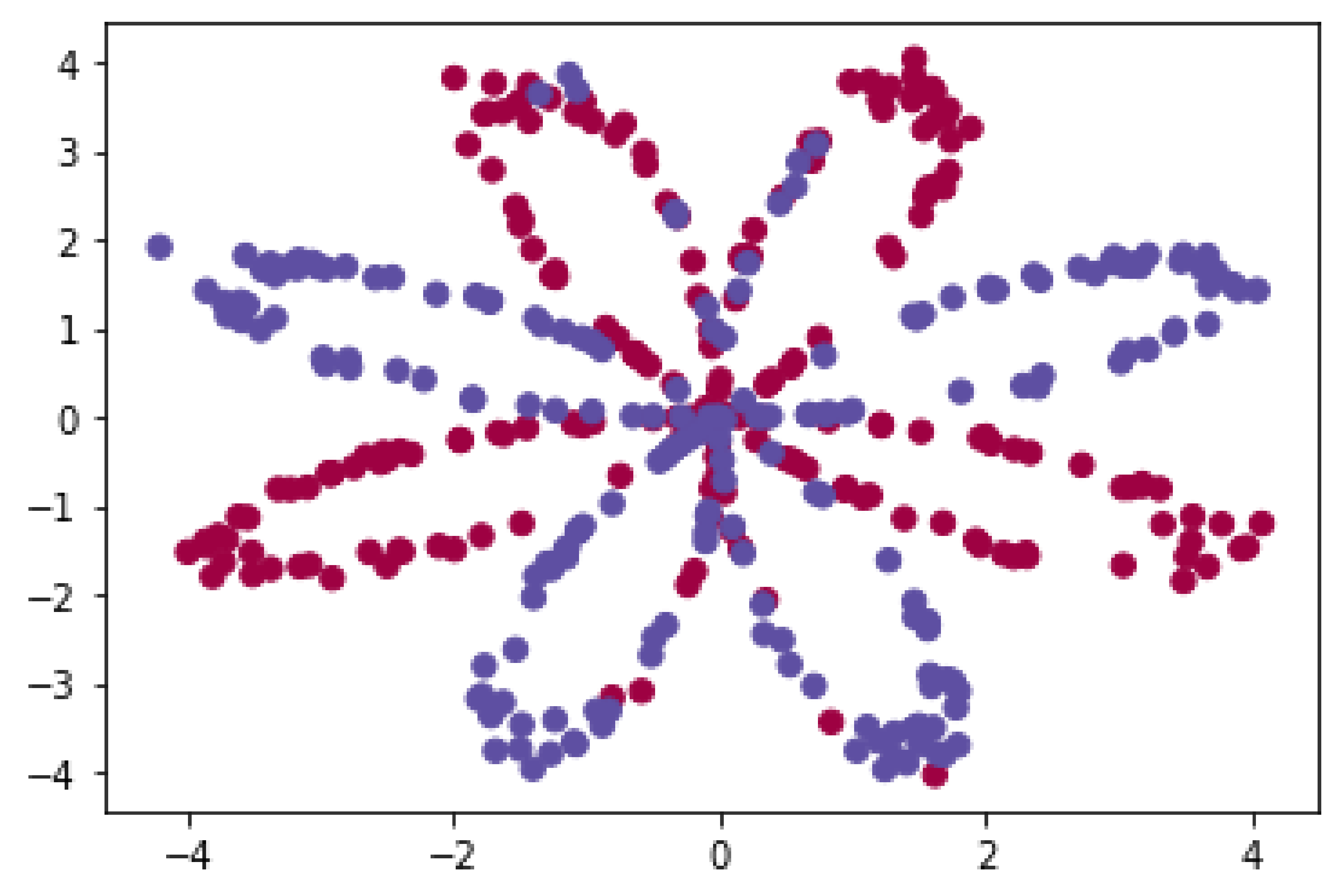
```python
# 权重更新函数
def update_parameters(parameters, grads, learning_rate=1.2):
    '''
    输入：
    parameters：神经网络参数字典
    grads：权重梯度字典
    learning_rate：学习率
    输出：
    parameters：更新后的权重字典
    '''
    # 获取参数
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    # 获取梯度
    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']
    # 参数更新
    W1 -= dW1 * learning_rate
    b1 -= db1 * learning_rate
    W2 -= dW2 * learning_rate
    b2 -= db2 * learning_rate

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}
    return parameters
```

- （7）神经网络模型封装。

```python
# 神经网络模型封装
def nn_model(X, Y, n_h, num_iterations=10000, print_cost=False):
    '''
    输入：X：训练输入 Y：训练输出 n_h：隐藏层节点数 num_iterations：迭代次数
    print_cost：训练过程中是否打印损失
    输出：parameters：神经网络训练优化后的权重系数
    '''
    # 设置随机数种子
    np.random.seed(3)
    # 输入和输出节点数
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]
    # 初始化模型参数
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']
    # 梯度下降和参数更新循环
    for i in range(0, num_iterations):
    # 前向传播计算
        A2, cache = forward_propagation(X, parameters)
        # 计算当前损失
        cost = compute_cost(A2, Y, parameters)
        # 反向传播
        grads = backward_propagation(parameters, cache, X, Y)
        # 参数更新
        parameters = update_parameters(parameters, grads, learning_rate=1.2)
        # 打印损失
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

- （8）生成模拟数据集

```
# 生成样本集
def create_dataset():
    '''
    输入:
    无
    输出:
    X:模拟数据集输入
    Y:模拟数据集输出
    '''
    np.random.seed(1)
    m = 400 # 数据量
    N = int(m/2) # 每个标签的实例数
    D = 2 # 数据维度
    X = np.zeros((m,D)) # 数据矩阵
    Y = np.zeros((m,1), dtype='uint8') # 标签维度
    a = 4 
    
    for j in range(2):
        ix = range(N*j,N*(j+1))
        t = np.linspace(j*3.12,(j+1)*3.12,N) + np.random.randn(N)*0.2 # theta
        r = a*np.sin(4*t) + np.random.randn(N)*0.2 # radius
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        Y[ix] = j
        
    X = X.T
    Y = Y.T

    return X, Y

X, Y = create_dataset()
plt.scatter(X[0, :], X[1, :], c=Y[0], s=40, cmap=plt.cm.Spectral);
```


4
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4

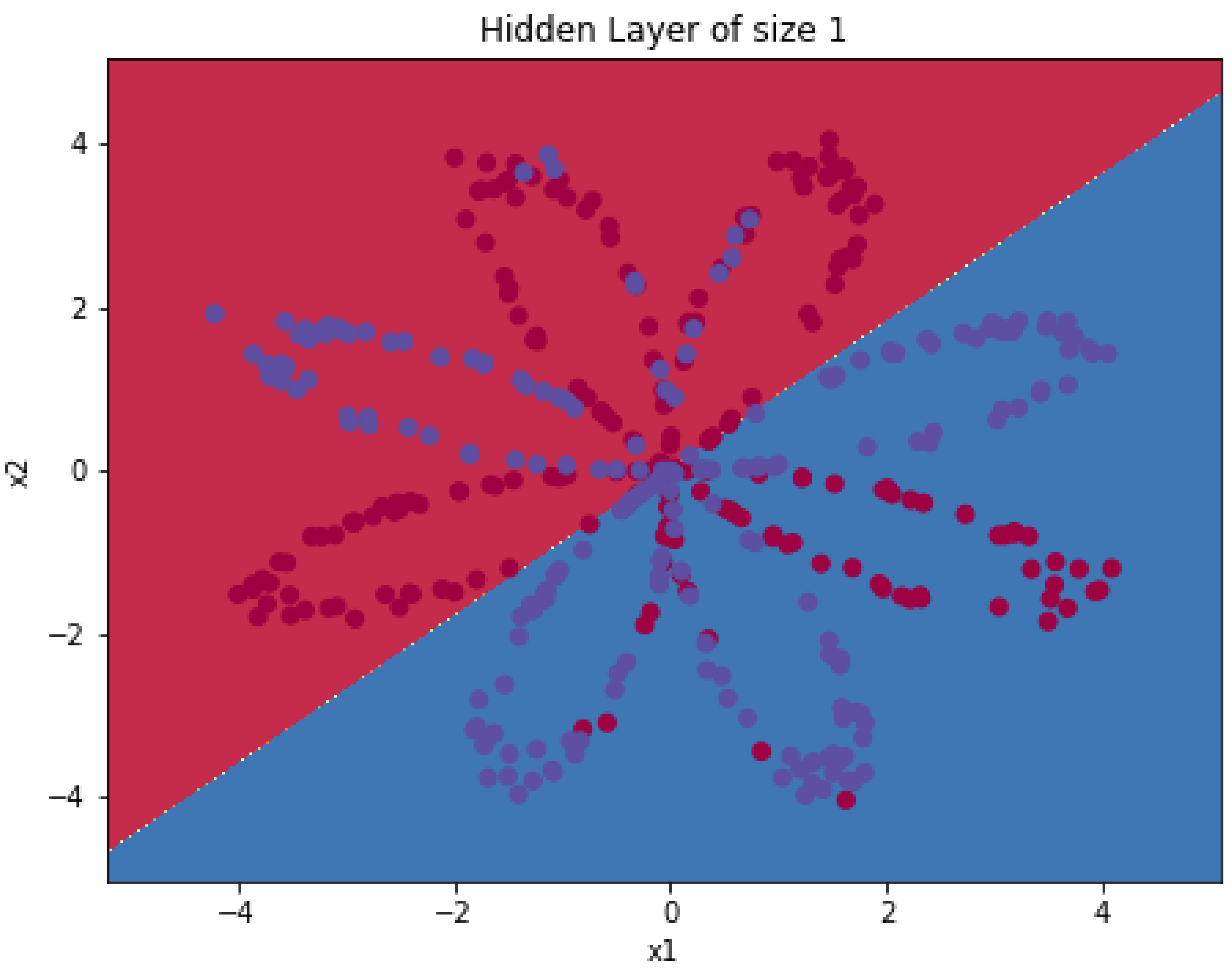
- （9）模型训练

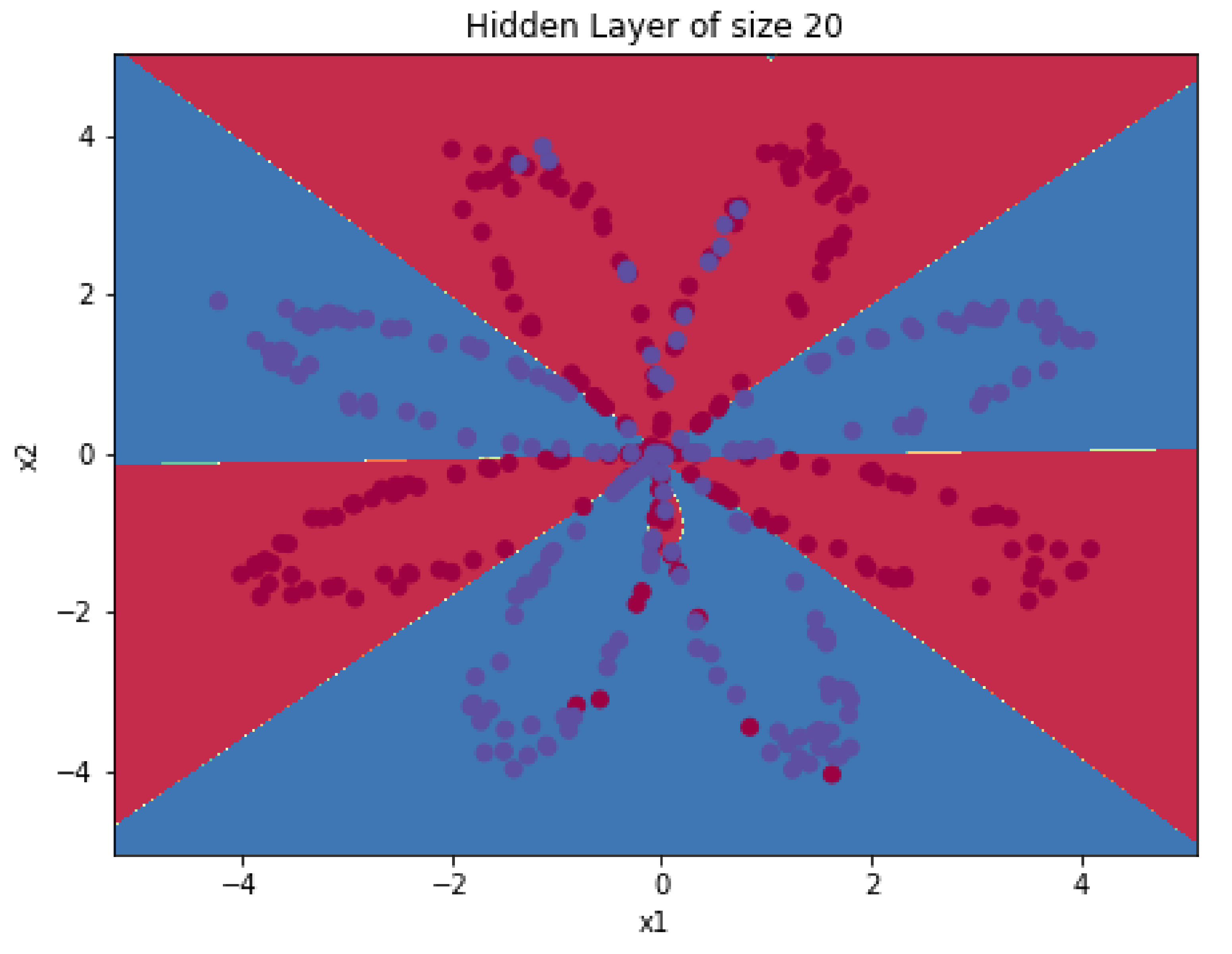
```
# 神经网络模型训练
parameters = nn_model(X, Y, n_h = 4, num_iterations=10000, print_cost=True)

def predict(parameters, X):
    A2, cache = forward_propagation(X, parameters)
    predictions = (A2>0.5)
    return predictions

predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) +
      np.dot(1-Y,1-predictions.T))/float(Y.size)*100) + '%')
```

```
IPdb [1]: !continue
Cost after iteration 0: 0.693162
Cost after iteration 1000: 0.258625
Cost after iteration 2000: 0.239334
Cost after iteration 3000: 0.230802
Cost after iteration 4000: 0.225528
Cost after iteration 5000: 0.221845
Cost after iteration 6000: 0.219094
Cost after iteration 7000: 0.220659
Cost after iteration 8000: 0.219408
Cost after iteration 9000: 0.218485
Accuracy: 90%
```

- （10）画出分类决策边界

```python
def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole grid
    Z = model(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.ylabel('x2')
    plt.xlabel('x1')
    plt.scatter(X[0, :], X[1, :], c=y, cmap=plt.cm.Spectral)
    
plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 10, 20]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 5000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y[0])
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) 
                      + np.dot(1-Y,1-predictions.T))/float(Y.size)*100)
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```


Hidden Layer of size 1
4
2
0
−2
−4
x2
−4
−2
0
2
4
x1


Hidden Layer of size 20
x2
4
2
0
−2
−4
x1
−4
−2
0
2
4

```
Accuracy for 1 hidden units: 67.5 %
Accuracy for 2 hidden units: 67.25 %
Accuracy for 3 hidden units: 90.75 %
Accuracy for 4 hidden units: 90.5 %
Accuracy for 5 hidden units: 91.25 %
Accuracy for 10 hidden units: 91.25 %
Accuracy for 20 hidden units: 91.0 %
```

六、实验报告要求

- 1、实验目的
- 2、实验内容
- 3、实验原理
- 4、实验代码
- 5、运行结果与分析
- 6、实验小结

- 说明：每个学生都要交电子版的实验报告，命名格式：
- 01/02-XXXX（学号）-XXX（姓名）

Q&A

>>>>>>>>>>>>>>>>>> <<<<<<<<<<<<<<<<<<