高级程序设计
---Python与深度学习
9. 感知机与神经网络
9. Perception & neural network

李冰
副研究员
交叉科学研究院

课程内容

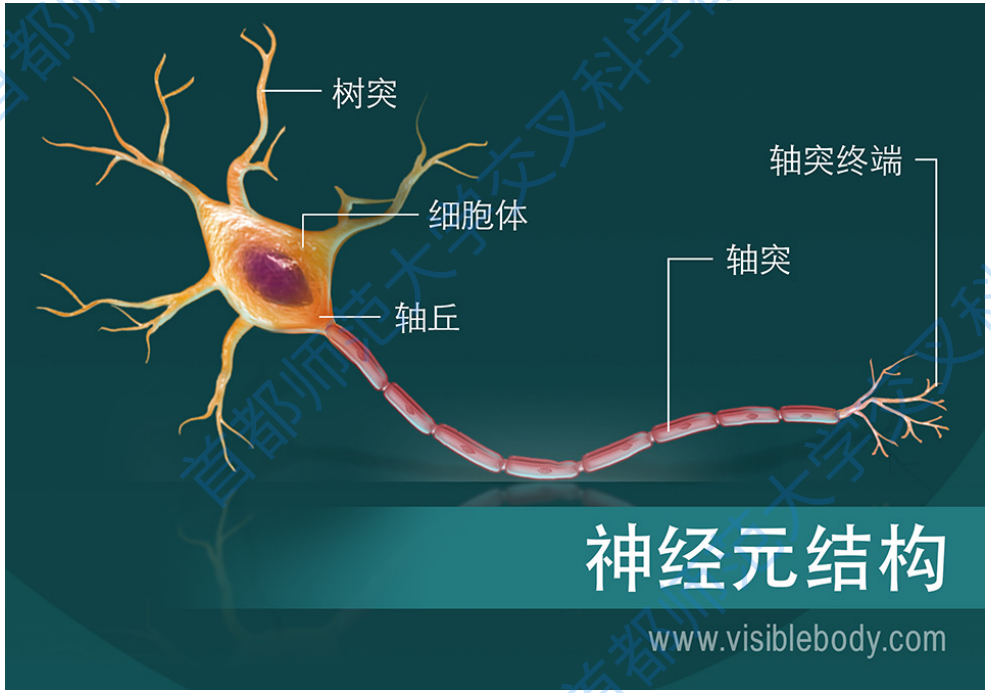
- 感知机
  - 多层感知机
- 神经网络
  - 激活函数
  - 神经网络每层的实现
  - 构建神经网络

感知机

- 感知机是由美国学者 Frank Rosenblatt 在1957 年发明的一种人工神经网络。
  - 单层的人工神经网络
- 感知机是神经网络（深度学习）的起源的算法，学习感知机的构造也是学习通向神经网络和深度学习的一种重要思想。
  - 感知机的结构是现代深度学习网络的最初形态。

感知机

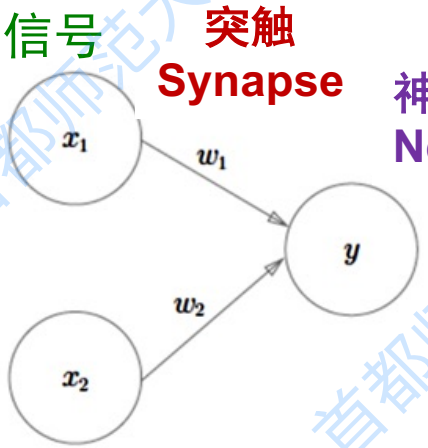
- 感知机是生物神经细胞的简单抽象。
  - 单个神经细胞视为只有两种状态的机器：激动时是“是”，未激动时“否”，状态取决于从其他神经细胞收到的输入信号量，及突触的强度。



感知机

- 感知机是生物神经细胞的简单抽象。
  - 单个神经细胞视为只有两种状态的机器：激动时是“是”，未激动时“否”，状态取决于从其他神经细胞收到的输入信号量，及突触的强度。
  - 感知机接收多个输入信号，输出一个信号。感知机的输出信号只有 **激活** 和 **抑制 (未激活)** 两种状态，使用1代表激活，0代表未激活。
  - 下图是一个接收两个输入信号的感知机的例子。
    - x1、x2 是输入信号，y 是输出信号，w1、w2 是权重，阈值θ。

从其他神经元传来的信号

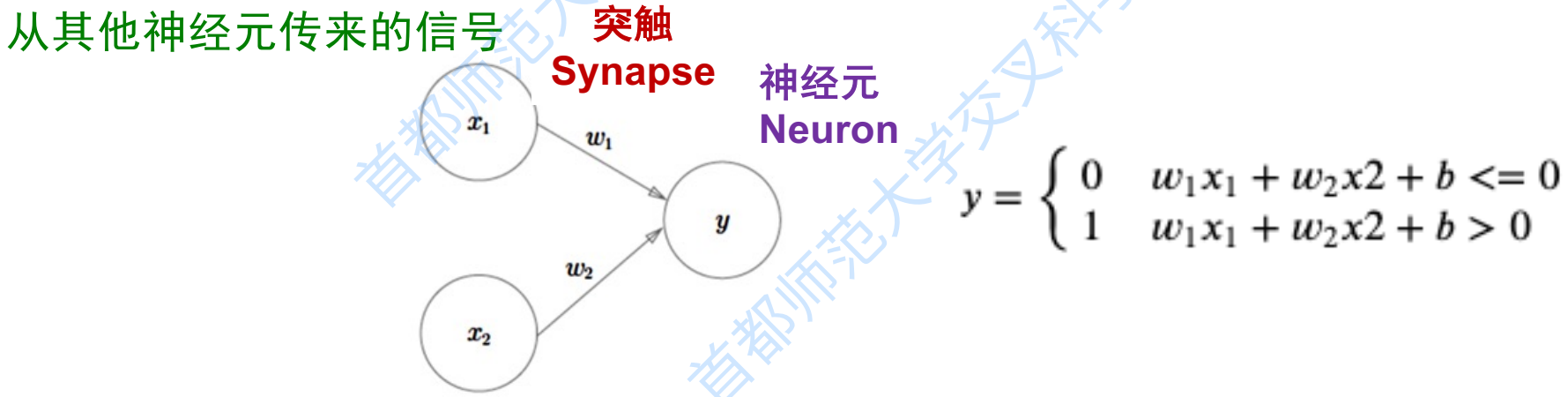


$$y = \begin{cases} 0 & w_1x_1 + w_2x2 <= \theta \\ 1 & w_1x_1 + w_2x2 > \theta \end{cases}$$

“神经元被激活”

感知机

- 感知机是生物神经细胞的简单抽象。
  - 单个神经细胞视为只有两种状态的机器：激动时是“是”，未激动时“否”，状态取决于从其他神经细胞收到的输入信号量，及突触的强度。
  - 感知机接收多个输入信号，输出一个信号。感知机的输出信号只有 **激活** 和 **抑制 (未激活)** 两种状态，使用1代表激活，0代表未激活。
  - 下图是一个接收两个输入信号的感知机的例子。
    - x1、x2 是输入信号，y 是输出信号，w1、w2 是权重，b是偏置。



实现逻辑运算

- 用感知机来解决简单的逻辑电路问题。
- 与门（AND gate）：两个输入，一个输出，只有两个输入都为1的时候输出1，其他输出0。

真值表

x1	x2	y
0	0	0
1	0	0
0	1	0
1	1	1

实现逻辑运算

- 用感知机来解决简单的逻辑电路问题。
- 与门（AND gate）：两个输入，一个输出，只有两个输入都为1的时候输出1，其他输出0。
- 模拟感知机，实现与门

```python
import numpy as np

def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1
```

```python
print(AND(0, 0)) # 输出0
print(AND(1, 0)) # 输出0
print(AND(0, 1)) # 输出0
print(AND(1, 1)) # 输出1
```

```
0
0
0
1
```

实现逻辑运算

- 用感知机来解决简单的逻辑电路问题。
- **或门**（OR gate）：两个输入，一个输出，只有两个输入都为0的时候输出0，其他输出1。
- 非门（NOT gate）：一个输入，一个输出，输入和输出始终相反

真值表

x1	x2	y
0	0	0
1	0	1
0	1	1
1	1	1

真值表

x	y
0	0
1	0

实现逻辑运算

- 用感知机来解决简单的逻辑电路问题。
- **或门**（OR gate）：两个输入，一个输出，只有两个输入都为0的时候输出0，其他输出1。
- 非门（NOT gate）：一个输入，一个输出，输入和输出始终相反

```
def OR(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.2
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1
```

```
def NOT(x):
    return 1 if x <= 0 else 0
```

感知机的局限性

- 用感知机来解决简单的逻辑电路问题。
- **异或门**（XOR gate）：两个输入，一个输出，仅当两个输入其中一方为1 时，才会输出1

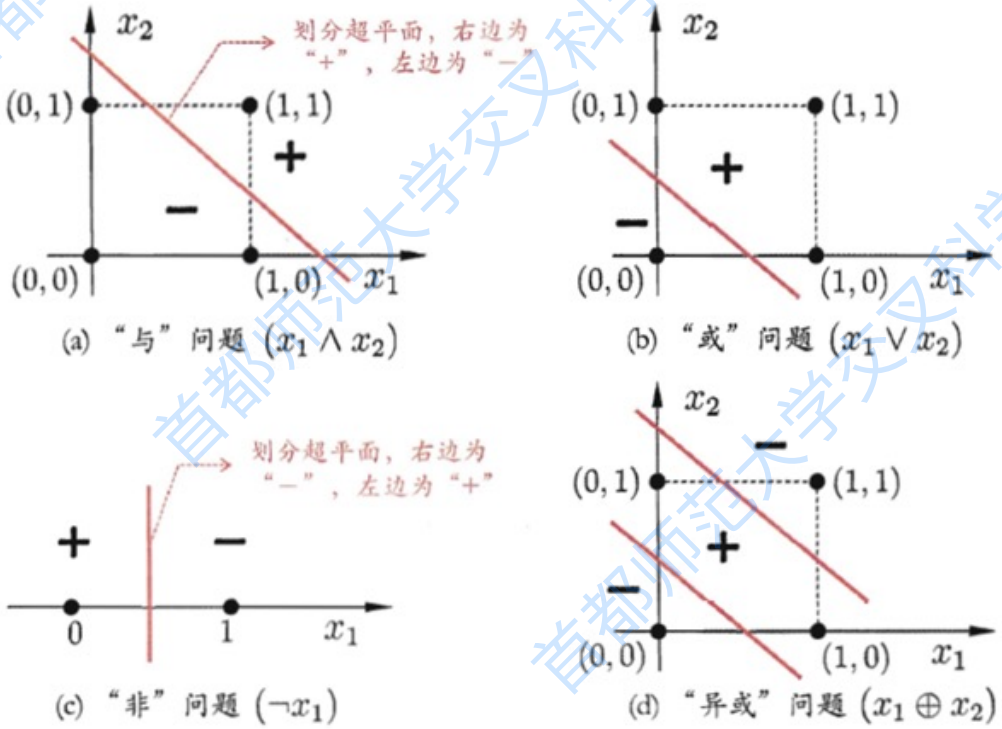
真值表

x1	x2	y
0	0	0
1	0	1
0	1	1
1	1	0

实际上，用前面介绍的感知机是无法实现这个异或门的。

感知机的局限性

- 只拥有一层功能神经元，其学习能力非常有限。
- 与、或、非问题都是线性可分(linearly separable)的问题，即存在一个线性超平面能将它们分开。
- 但，异或属于非线性可分问题。



感知机的局限性

- 只拥有一层功能神经元，其学习能力非常有限。
- 与、或、非问题都是线性可分(linearly separable)的问题，即存在一个线性超平面能将它们分开。
- 但，异或属于非线性可分问题。

1969年，马文·明斯基和西摩尔·派普特在《Perceptrons》书中，仔细分析了以感知机为代表的单层神经网络系统的功能及局限，证明感知机不能解决简单的异或（XOR）等线性不可分问题，

80年代，人们认识到多层感知机没有单层感知机固有的缺陷及并提出反向传播算法

多层感知机

- 多层感知机(Multi-layered Perception, MLP):通过叠加层来构造感知机可以解决非线性可分问题。
- 使用之前定义的AND函数、OR函数，可以轻松实现异或门。

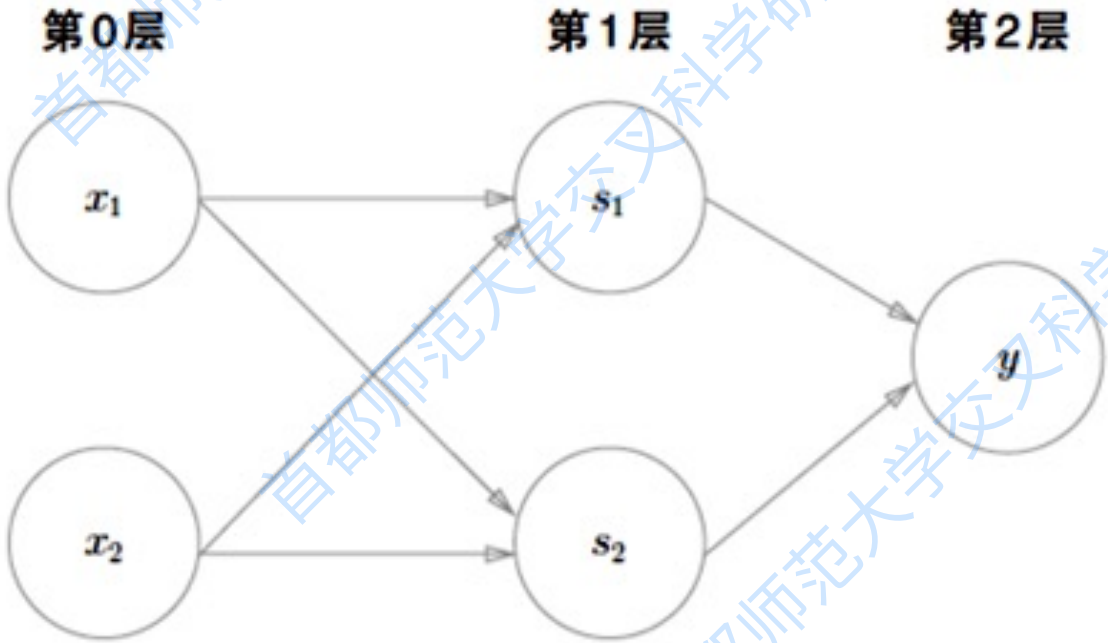
```
def XOR(x1, x2):
    s1 = not AND(x1, x2)
    s2 = OR(x1, x2)
    y = AND(s1, s2)
    return y
```

```
print(XOR(0, 0)) # 输出0
print(XOR(1, 0)) # 输出1
print(XOR(0, 1)) # 输出1
print(XOR(1, 1)) # 输出0
```

```
0
1
1
0
```

多层感知机

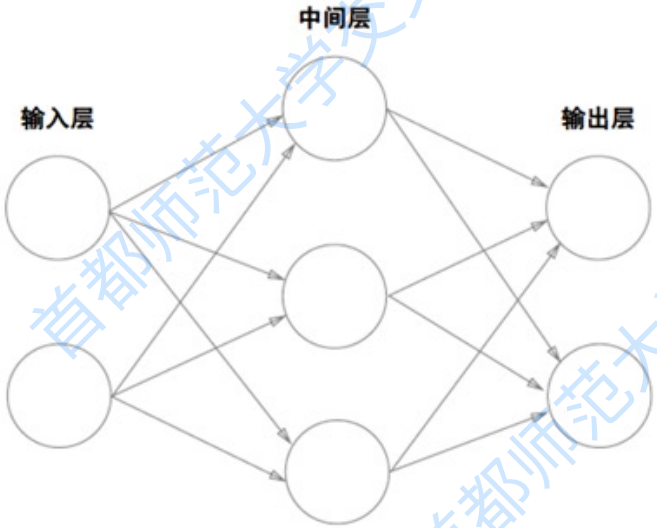
- 多层感知机(Multi-layered Perception, MLP):通过叠加层来构造感知机可以解决非线性可分问题。
- 使用之前定义的AND函数、OR函数，可以轻松实现异或门。



与门、或门是单层感知机，而异或门是一种
多层感知机（MLP, multi-layered perceptron）

神经网络

- 神经网络：层级结构。
- **多层前馈神经网络**(multi-layer feedforward neural network)：每层神经元与下一层神经元全互连，神经元之间不存在同层连接，也不存在跨层连接。
  - **输入层，输出层，中间层(**隐藏层， hidden layer)。
  - 中间层和输出层神经元都是拥有**激活函数**的功能神经元。



$$\sigma\left(\sum_i w_i x_i + b\right)$$

激活函数

- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **阶跃函数**
  - 将输入值映射为输出值 0 或者 1 ，0 对应神经元抑制，1 对应神经元激活。因此阶跃函数具有不连续、不光滑等性质。

```
import numpy as np
def step_function(x):
    if x > 0:
        return 1
    else:
        return 0
```

但是参数 x 只能接受实数，但不允许参数取 NumPy 数组

激活函数

- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **阶跃函数**
  - 将输入值映射为输出值 0 或者 1 ，0 对应神经元抑制，1 对应神经元激活。因此阶跃函数具有不连续、不光滑等性质。

```
import numpy as np
def step_function(x):
    if x > 0:
        return 1
    else:
        return 0
```

```
step_function(np.array([-1.0, 1.0, 2.0]))
```

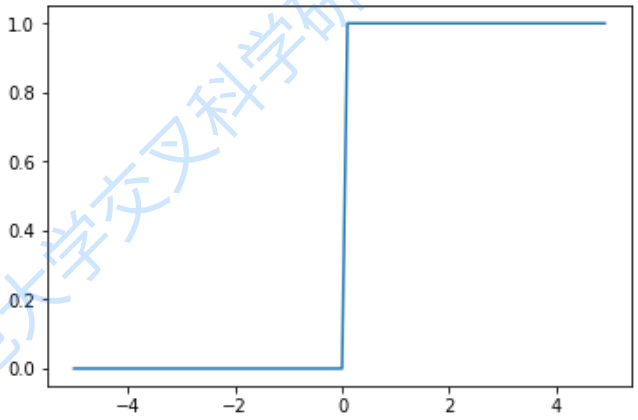
激活函数

- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **阶跃函数**
  - 将输入值映射为输出值 0 或者 1 ，0 对应神经元抑制，1 对应神经元激活。因此阶跃函数具有不连续、不光滑等性质。

```
import numpy as np
import matplotlib.pylab as plt


def step_function(x):
    return np.array(x > 0, dtype=np.int)

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.show()
```



激活函数

- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **Sigmoid 函数**
  - 由于阶跃函数的不连续性，实际中常用 sigmoid 函数作为激活函数。

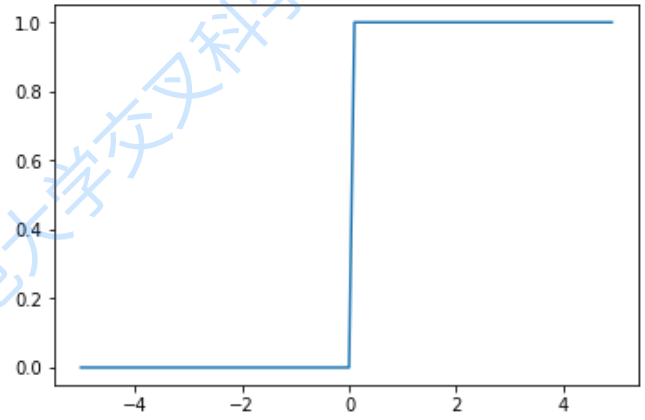
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

它的导数有性质：

$$\sigma(x)' = \sigma(x) \cdot (1 - \sigma(x))$$

```
def step_function(x):
    return np.array(x > 0, dtype=np.int)

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.show()
```



激活函数

- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **Sigmoid 函数**
  - 由于阶跃函数的不连续性，实际中常用 sigmoid 函数作为激活函数。

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

它的导数有性质：

$$\sigma(x)' = \sigma(x) \cdot (1 - \sigma(x))$$

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
x = np.array([-1.0, 1.0, 2.0])
sigmoid(x)
```

```
array([0.26894142, 0.73105858, 0.88079708])
```

激活函数

- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **Sigmoid 函数**
  - 由于阶跃函数的不连续性，实际中常用 sigmoid 函数作为激活函数。

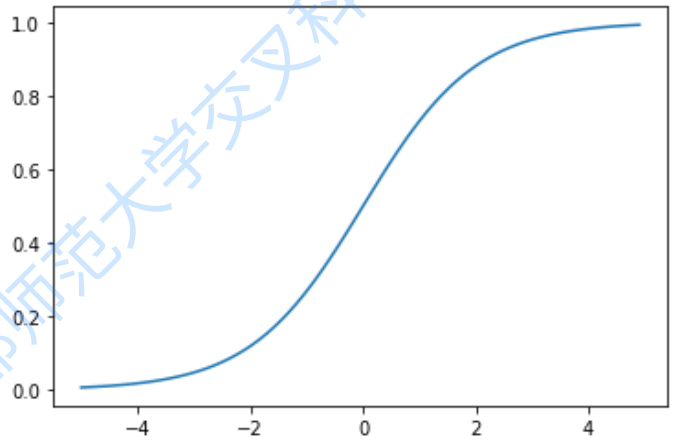
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

它的导数有性质：

$$\sigma(x)' = \sigma(x) \cdot (1 - \sigma(x))$$

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.show()
```



激活函数

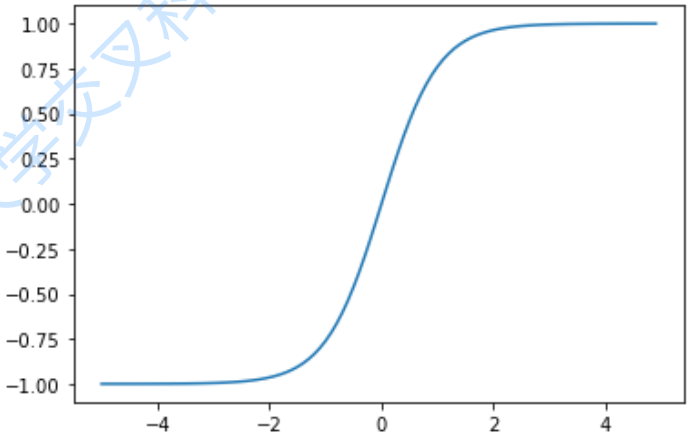
- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **Tanh 函数**
  - 双曲正切函数有时也被用做神经网络的激活函数
  - 修正了 Sigmoid 函数不关于原点对称的问题，可是它在两边还是有饱和（也就是导数趋近于0）的问题。

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

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

```
def tanh(x):
    return np.tanh(x)
```

```
x = np.arange(-5.0, 5.0, 0.1)
y = tanh(x)
plt.plot(x, y)
plt.show()
```



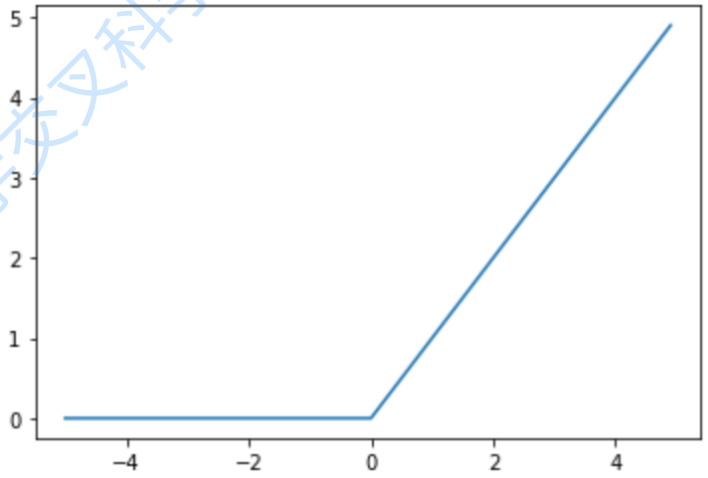
激活函数

- 神经网络的激活函数必须使用非线性函数，否则加深神经网络的层数就没有意义了。
- **ReLU函数**
  - 最近则主要使用 ReLU函数可以表示为下面的式：

$$h(x) = \begin{cases} x & x > 0 \\ 0 & x <= 0 \end{cases}$$

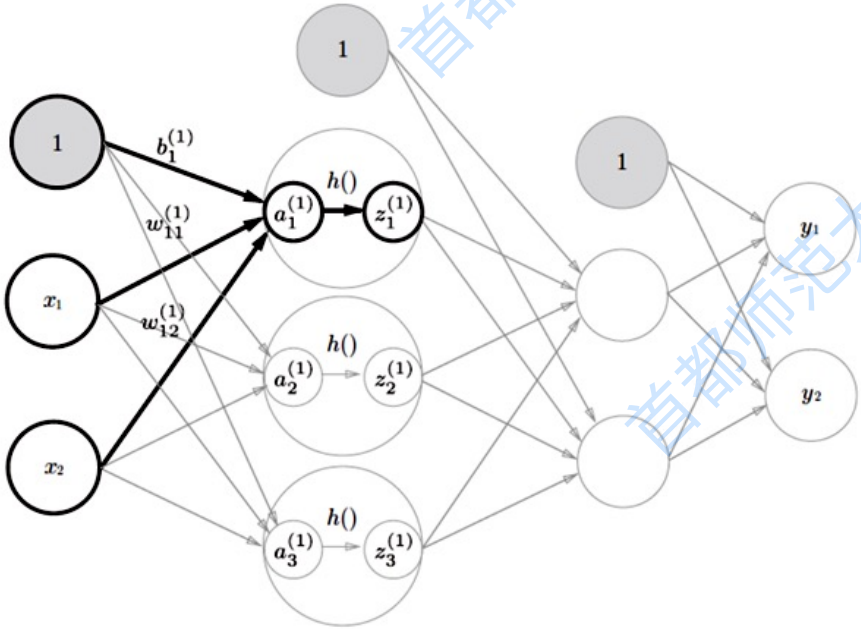
```
def relu(x):
    return np.maximum(0, x)
```

```
x = np.arange(-5.0, 5.0, 0.1)
y = relu(x)
plt.plot(x, y)
plt.show()
```



神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 先看一下从输入层到第 1 层的第 1 个神经元的信号传递过程，图中增加了表示偏置的神经元“1”。

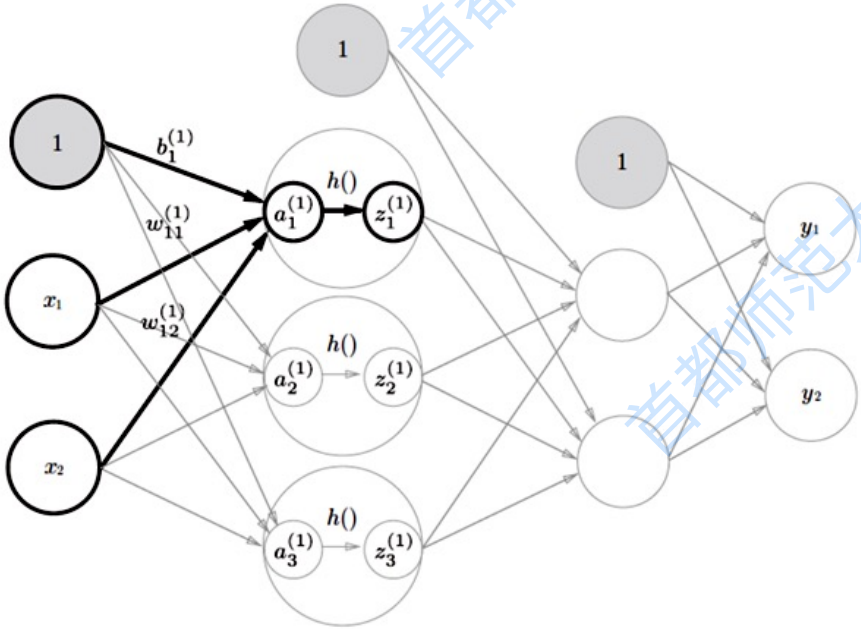


$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

其中，$w_{12}^{(1)}$表示前一层的第 2 个神经元x_2到后一层的第 1 个神经元 $a_1^{(1)}$ 的权重。

神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 先看一下从输入层到第 1 层的第 1 个神经元的信号传递过程，图中增加了表示偏置的神经元“1”。



$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

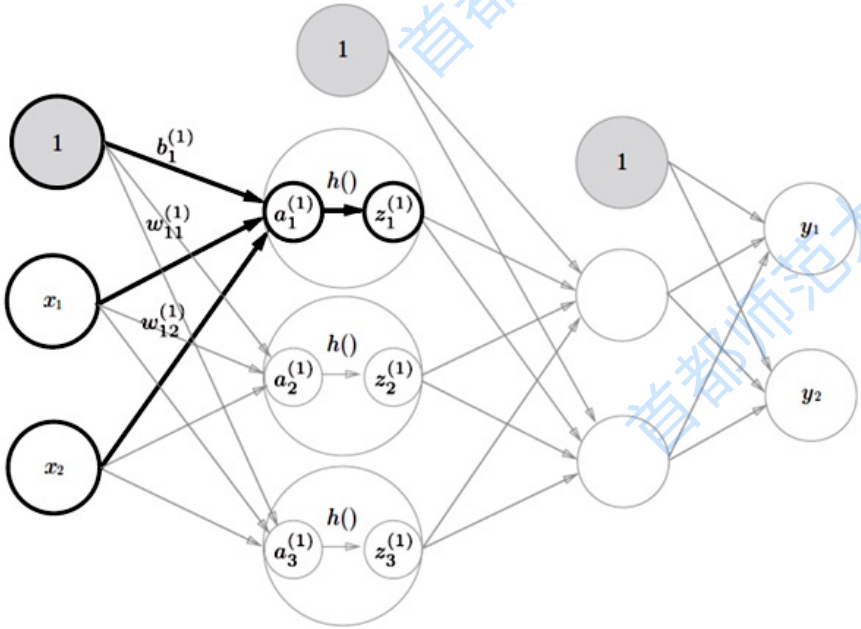
其中，$w_{12}^{(1)}$表示前一层的第 2 个神经元x_2到后一层的第 1 个神经元 $a_1^{(1)}$ 的权重。

如果使用矩阵的乘法运算，第 1 层的加权和

$$\boldsymbol{A}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{B}^{(1)}$$

神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 先看一下从输入层到第 1 层的第 1 个神经元的信号传递过程，图中增加了表示偏置的神经元“1”。



$$A^{(1)} = XW^{(1)} + B^{(1)}$$

其中

$$A^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}$$

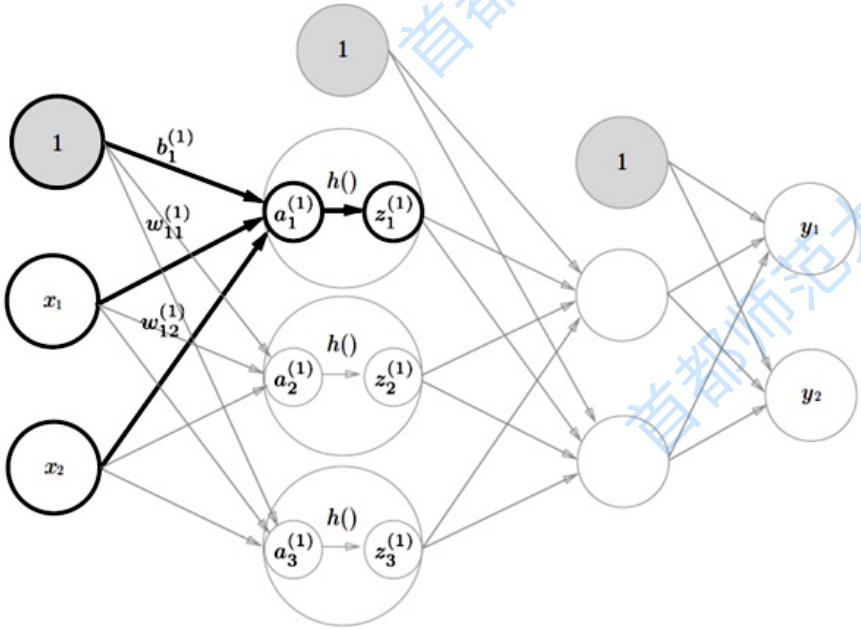
$$X = \begin{pmatrix} x_1 & x_2 \end{pmatrix}$$

$$B^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 先看一下从输入层到第 1 层的第 1 个神经元的信号传递过程，图中增加了表示偏置的神经元“1”。

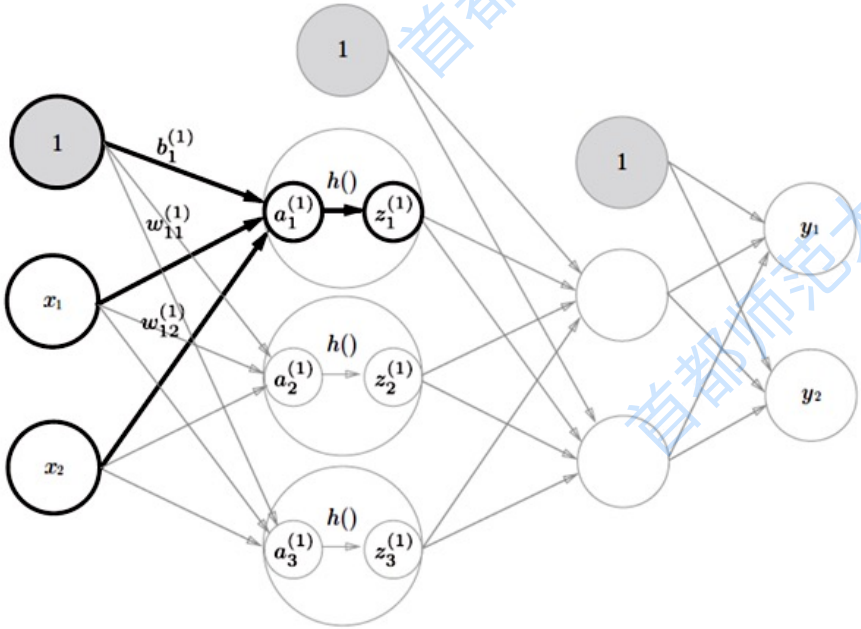


$$A^{(1)} = XW^{(1)} + B^{(1)}$$

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5],
               [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
A1 = np.dot(X, W1) + B1
print(X.shape)
print(W1.shape)
print(B1.shape)
```

神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 先看一下从输入层到第 1 层的第 1 个神经元的信号传递过程，图中增加了表示偏置的神经元“1”。



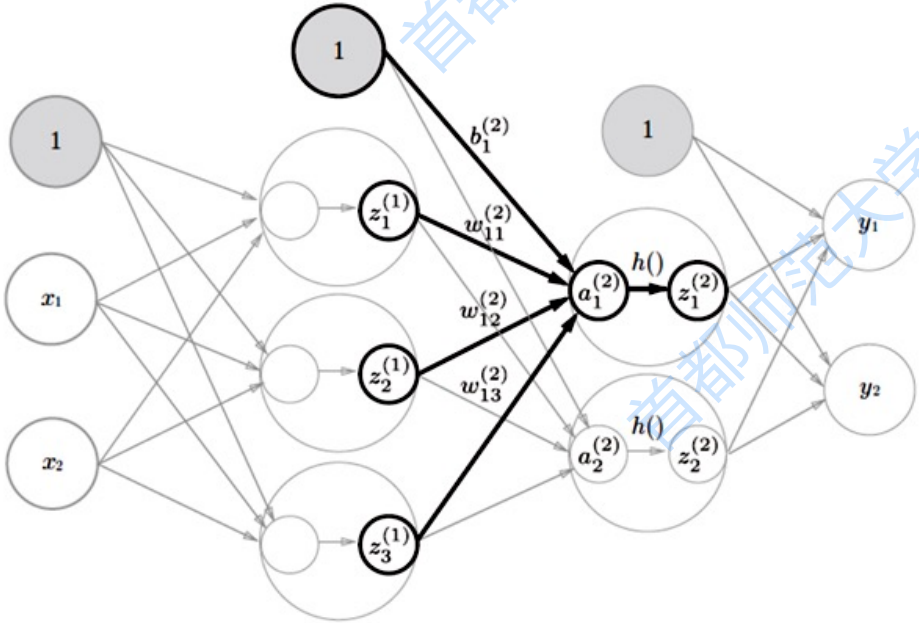
$$Z^{(1)} = \sigma(A^{(1)})$$

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5],
               [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)
print(A1)
print(Z1)
```

```
[0.3 0.7 1.1]
[0.57444252 0.66818777 0.75026011]
```

神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 实现第 1 层到第 2 层的信号传递，其中第 1 层的输出$Z^{(1)}$ 变成了第 2 层的输入。



$$a_1^{(2)} = w_{11}^{(2)} z_1^{(1)} + w_{12}^{(2)} z_2^{(1)} + w_{13}^{(2)} z_3^{(1)} + b_1^{(2)}$$

$$A^{(2)} = Z^{(1)} W^{(1)} + B^{(2)}$$

$$A^{(2)} = \begin{pmatrix} a_1^{(2)} & a_2^{(2)} \end{pmatrix}$$

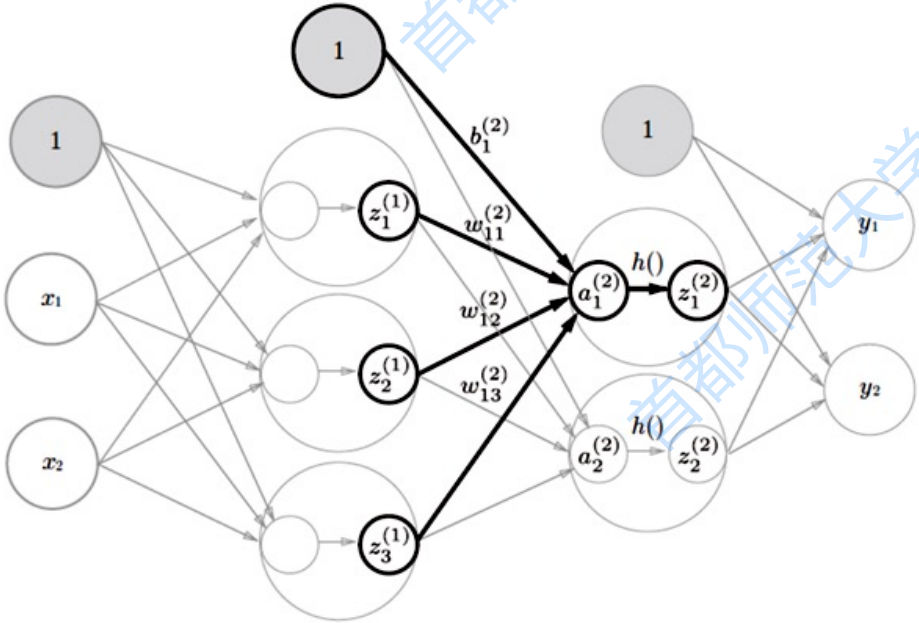
$$Z^{(1)} = \begin{pmatrix} z_1^{(2)} & z_2^{(2)} & z_3^{(2)} \end{pmatrix}$$

$$B^{(2)} = \begin{pmatrix} b_1^{(2)} & b_2^{(2)} \end{pmatrix}$$

$$W^{(2)} = \begin{pmatrix} w_{11}^{(2)} & w_{21}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} \\ w_{13}^{(2)} & w_{23}^{(2)} \end{pmatrix}$$

神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 实现第 1 层到第 2 层的信号传递，其中第 1 层的输出$Z^{(1)}$ 变成了第 2 层的输入。



$$A^{(2)} = Z^{(1)}W^{(1)} + B^{(2)}$$

$$Z^{(2)} = \sigma(A^{(2)})$$

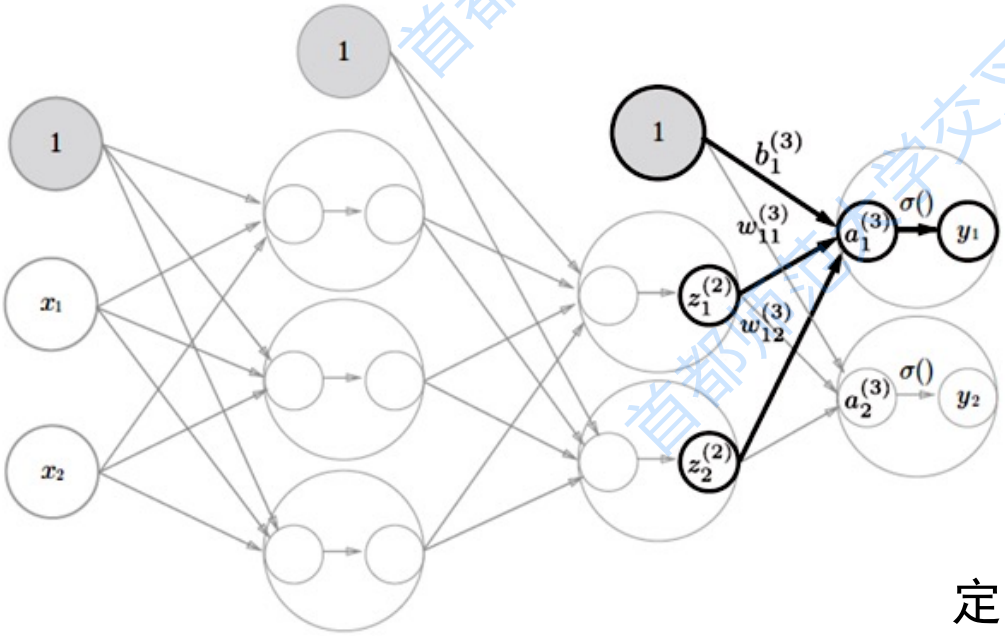
```
W2 = np.array([[0.1, 0.4],
                [0.2, 0.5],
                [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

A2 = np.dot(Z1, W2) + B2

Z2 = sigmoid(A2)
print(A1)
print(Z2)
[0.51615984 1.21402696]
[0.62624937 0.7710107 ]
```

神经网络的实现

- 现在我们来实现一个简单 3 层神经网络从输入到输出的（前向）处理。
  - 第 2 层到输出层的信号传递。输出层的实现也和之前的实现基本相同。
  - 最后的激活函数和之前的隐藏层有所不同。




```
def identity_function(x):
    return x

W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])
A3 = np.dot(Z2, W3) + B3
Y = identity_function(A3)
print(Y)
```

定义了 identity_function() 函数（也称为“恒等函数”），并将其作为输出层的激活函数。

前向传播(forward propagation)代码小结

```python
def init_network():
    network = {}
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
    network['b2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
    network['b3'] = np.array([0.1, 0.2])
    return network

def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)
    return y

network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
print(y)
```

定义了 init_network() 和 forward() 函数

输出层的设计

- 神经网络可以用在分类问题和回归问题上，需要根据具体任务改变输出层的激活函数。
  - 一般而言，回归问题用恒等函数，二分类问题使用 sigmoid 函数，多分类问题用 softmax 函数。

输出层共有 n 个神经元，计算第 k 个神经元的输出值 yk。
神经网络把输出值最大的神经元所对应的类别作为识别结果。

softmax 函数

$$y_k = \frac{exp(a_k)}{\sum_{i=1}^{n} exp(a_i)}$$

$$\sum_{i=1}^{n} y_i = 1$$

- 输出是 0.0 到 1.0 之间的实数。
- 输出值的总和为 1。
- 单调递增

softmax函数的实现

$$y_k = \frac{exp(a_k)}{\sum_{i=1}^{n} exp(a_i)}$$

$$\sum_{i=1}^{n} y_i = 1$$

```
a = np.array([0.3, 2.9, 4.0])
exp_a = np.exp(a)                    # 指数函数
```

```
print(exp_a)
```

```
[ 1.34985881 18.17414537 54.59815003]
```

```
sum_exp_a = np.sum(exp_a)            # 指数函数的和
```

```
print(sum_exp_a)
```

```
74.1221542101633
```

```
y = exp_a / sum_exp_a
```

```
print(y)
```

```
[0.01821127 0.24519181 0.73659691]
```

```
np.sum(y)
```

```
1.0
```

softmax函数的实现

$$y_k = \frac{exp(a_k)}{\sum_{i=1}^{n} exp(a_i)}$$

$$\sum_{i=1}^{n} y_i = 1$$

```
def softmax(a):
    exp_a = np.exp(a)
    return exp_a / np.sum(exp_a)
```

上面的 softmax 函数的实现虽然正确，但在运算起来会有问题

- 当a非常大的时候，指数函数运算数值溢出的问题。
- 这例如，e^{100} 等于2.69e+43，e^{1000} 的结果会返回一个表示无穷大的inf。
- 如果在这些超大值之间进行除法运算，结果会出现“不确定”的情况。

```
np.exp(1000)
```

```
inf
```

```
a = np.array([1010, 1000, 990])
softmax(a)
```

```
RuntimeWarning: invalid value encountered in true_divide
This is separate from the ipykernel package so we can
avoid doing imports until
```

```
array([nan, nan, nan])
```

softmax函数的实现

$$y_k = \frac{exp(a_k)}{\sum_{i=1}^{n} exp(a_i)}$$

$$= \frac{exp(a_k - \max(a))}{\sum_{i=1}^{n} exp(a_i - \max(a))}$$

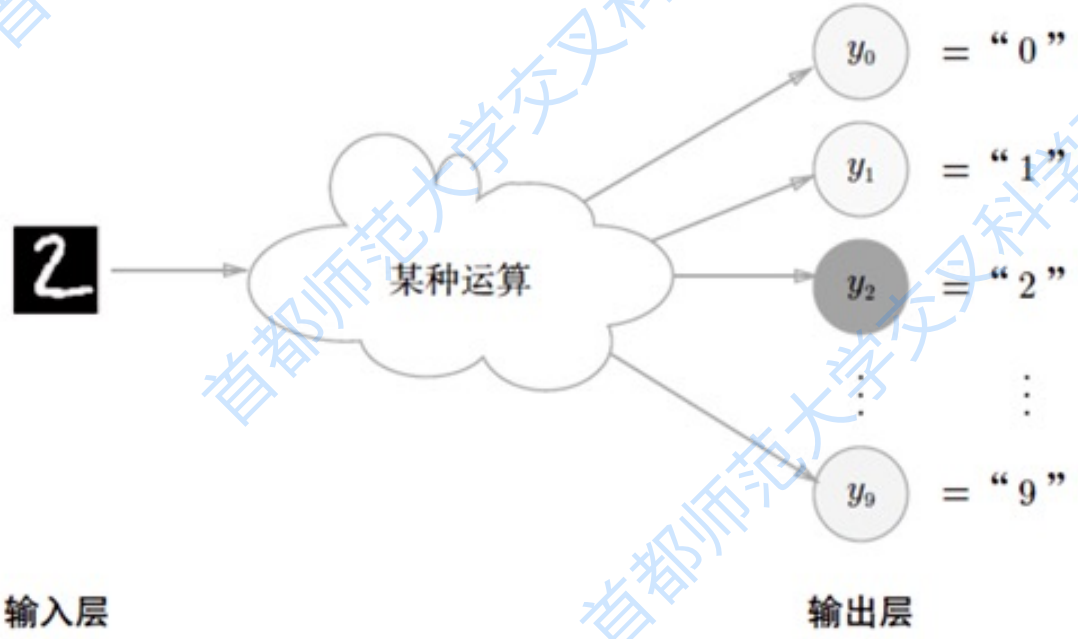
```python
def softmax(a):
    exp_a = np.exp(a - np.max(a)) # 溢出对策
    return exp_a / np.sum(exp_a)
```

```python
a = np.array([1010, 1000, 990])
softmax(a)
```

```
array([9.99954600e-01, 4.53978686e-05, 2.06106005e-09])
```

输出神经元的数量

- 输出层的神经元数量需要根据待解决的问题来决定。
  - 对于分类问题，输出层的神经元数量一般设定为类别的数量。
    - 比如，对于手写数字图像识别问题，预测是图中的数字 0 到 9 中的哪一个的问题（10 类别分类问题），可以将输出层的神经元设定为 10 个。



神经网络中的“Hello world”

- 手写数字识别
  - 我们有一批图片，是黑白数字图像，构建模型对图像中的数字正确分类

神经网络中的“Hello world”

- 手写数字识别
  - 我们有一批图片，是黑白数字图像，构建模型对图像中的数字正确分类
- MNIST数据集
  - 由 0 到 9 的黑白数字图像构成的。
  - 每张图片大小和分辨率：28 像素× 28 像素的 8bit 灰度图像（单通道），每个像素的取值在 0 到 255 之间。
    - 每个图像数据都相应地标有“7”、“2”、 “1”等标签。
  - 训练集包含 6 万张图像，测试集包含 1 万张图像
  - 机器学习领域最有名的数据集之一

获取数据集

```
from demo_code.download import load_mnist
(train_images, train_labels), (test_images, test_labels) = load_mnist()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 2s 0us/step
Dataset loading completed
```

```
def load_mnist(path='mnist.npz'):
    origin_folder = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/'
    path = get_file(path, origin=origin_folder + 'mnist.npz',
file_hash='731c5ac602752760c8e48fbffcf8c3b850d9dc2a2aedcf2cc48468fc17b673d1')
    with np.load(path, allow_pickle=True) as f:
        x_train, y_train = f['x_train'], f['y_train']
        x_test, y_test = f['x_test'], f['y_test']

    print('Dataset loading completed')

    return (x_train, y_train), (x_test, y_test)
```

获取数据集

```
from demo_code.download import load_mnist
(train_images, train_labels), (test_images, test_labels) = load_mnist()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 2s 0us/step
Dataset loading completed
```

```
print(type(train_images),train_images.shape)
```

```
<class 'numpy.ndarray'> (60000, 28, 28)
```

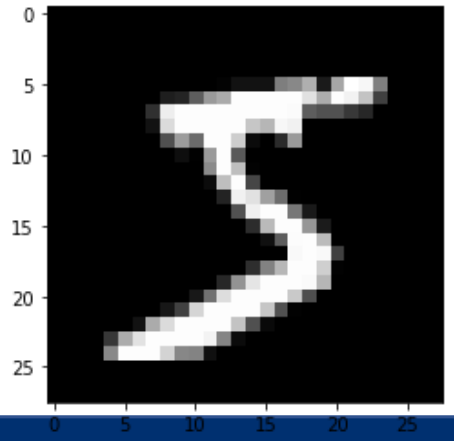
```
image = train_images[0]
label = train_labels[0]
print(label)
print(type(image))
```

```
5 <class 'numpy.ndarray'>
```

```
import numpy as np
import matplotlib.pyplot as plt

print(image.shape)
plt.imshow(image, cmap=plt.cm.gray)
plt.show()
```

```
(28, 28)
```



神经网络的推理

- 我们有一个训练好的网络，存在sample_weight.pkl中，文件以字典的形式保存了权重和偏置参数

```
def init_network():
    with open("demo_code/mnist_weight.pkl", 'rb') as f:
        network = pickle.load(f)
    return network
```

神经网络的输入层有 784 个神经元（图像分辨率 28 × 28 = 784），输出层有 10 个神经元（数字 0 到9，共 10 个类别）。此外，这个神经网络有 2 个隐藏层，第 1 个隐藏层有 50 个神经元，第 2 个隐藏层有 100 个神经元。

```
def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)
    return y
```

神经网络的推理

```
def get_data():
    (train_images, train_labels), (test_images, test_labels) = load_mnist()
    return test_images, test_labels
```

```
import pickle

network = init_network()

test_images, test_labels = get_data()
test_images = test_images.astype(np.float32) / 255.0      # Normalization

accuracy_cnt = 0

for i in range(len(test_images)):
    y = predict(network, test_images[i].flatten())
    p = np.argmax(y)                    # argmax获取数组中最大值的索引
    if p == test_labels[i]:
        accuracy_cnt += 1

print("Accuracy: " + str(float(accuracy_cnt) / len(test_images)))
```

神经网络的推理

```
def get_data():
    (train_images, train_labels), (test_images, test_labels) = load_mnist()
    return test_images, test_labels
```

```
import pickle

network = init_network()

test_images, test_labels = get_data()                          图像预处理
test_images = test_images.astype(np.float32) / 255.0        # Normalization

accuracy_cnt = 0

for i in range(len(test_images)):
    y = predict(network, test_images[i].flatten())
    p = np.argmax(y)                        # argmax获取数组中最大值的索引
    if p == test_labels[i]:
        accuracy_cnt += 1

print("Accuracy: " + str(float(accuracy_cnt) / len(test_images)))
```

神经网络的推理

```
def get_data():
    (train_images, train_labels), (test_images, test_labels) = load_mnist()
    return test_images, test_labels
```

```
import pickle

network = init_network()

test_images, test_labels = get_data()
test_images = test_images.astype(np.float32) / 255.0      # Normalization

accuracy_cnt = 0

for i in range(len(test_images)):
    y = predict(network, test_images[i].flatten())
    p = np.argmax(y)                    # argmax获取数组中最大值的索引
    if p == test_labels[i]:
        accuracy_cnt += 1

print("Accuracy: " + str(float(accuracy_cnt) / len(test_images)))
```

```
Accuracy: 0.9352
```

批处理(Batch)

- 打包输入多张图像的情形。
  - 例如，我们想用 predict() 函数一次性打包处理 100 张图像，将 x 的形状改为 100 × 784，

```
network = init_network()

test_images, test_labels = get_data()
test_images = test_images.astype(np.float32) / 255.0      # Normalization

batch_size = 100
accuracy_cnt = 0

for i in range(0, len(test_images), batch_size):
    test_images_batch = test_images[i:i+batch_size].reshape(batch_size, 784)
    y_batch = predict(network, test_images_batch)
    p = np.argmax(y_batch, axis=1)          # 沿指定轴找到最大值元素的索引
    accuracy_cnt += np.sum(p == test_labels[i:i+batch_size])

print("Accuracy: " + str(float(accuracy_cnt) / len(test_images)))
```

小结

- 感知机
  - 多层感知机
- 神经网络
  - 激活函数
    - Sigmoid
    - Tanh
    - Relu
  - 神经网络每层的实现
  - 神经网络中的hellowor ld例子
    - 数据集
    - 推理