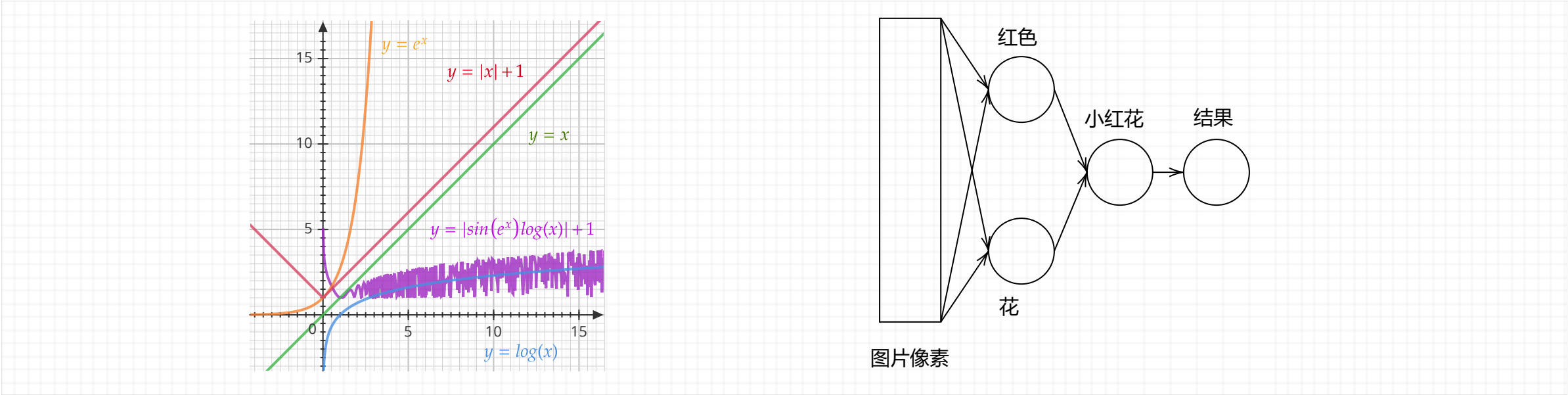


多层神经网络

上面说我们可以使用一个非线性的函数来将线性回归转化成符合实际的非线性回归，那么如果现实情况非常复杂，那我们可能需要将两个或多个非线性的输出组合成一个单元去拟合一个非线性的结果。如果情况更复杂，那么我们还可以把很多种这样的计算单元在组合成新的计算单元去拟合。如下图的紫色函数图像就是由一堆其他函数组合出来的，这里如果类比于控制系统中的传递函数、或者是叠加原理也很好理解。例如我们要识别图像中有一朵小红花。那我们就可以采用一组计算单元去识别红色，另一组单元去识别花，他们两个的计算结果组合在一起就能得到图片中有没有一朵小红花，当然这是理想的情况。

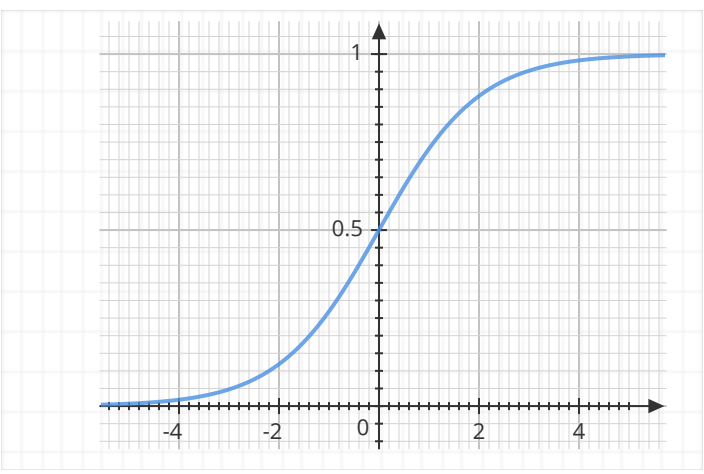
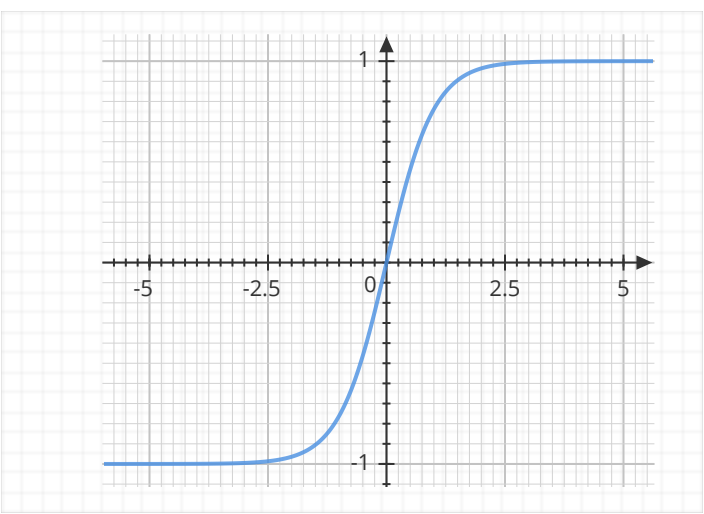


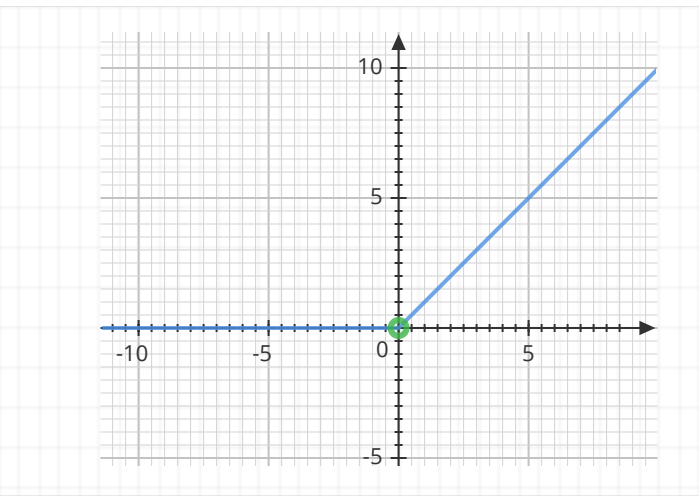
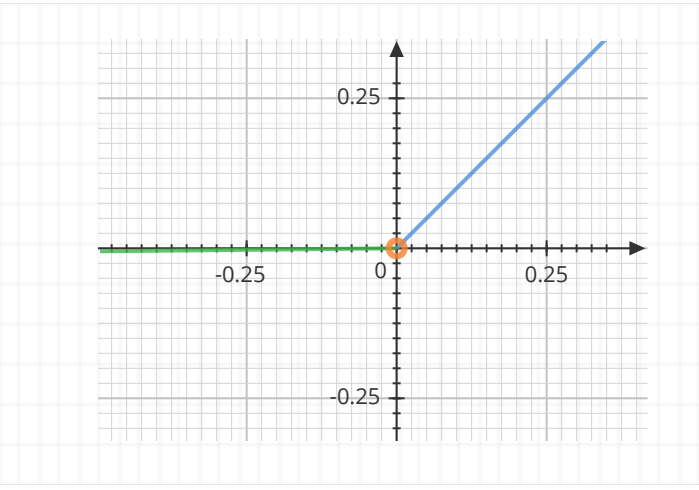
现实中当然还有更加复杂的例子，比如我要识别图片中有一只狗。那就要提取很多狗的特征：头、身体、四肢、尾巴；而头又可以分为：眼睛、鼻子、耳朵等；耳朵又有形状、毛色等特征。于是我们就需要一个很庞大的组合来拟合最终的结果。看起来就像一张网，大概这就是它为什么被称为神经网络吧，但实际上对于神经元节点来说却更像是一棵树。

激活函数

有一种解释说，真正的神经元细胞并不是接收到外界刺激就产生反应的。而是当外界刺激到达一定阈值之后才产生电信号。每个神经元都有激活函数哦！

我们一般采用的激活函数主要有以下四种类型：

函数名	表达式	导数	图像
<i>Sigmoid</i>	$y = \frac{1}{1 + e^{-z}}$	$y' = y(1 - y)$	
<i>tanh</i>	$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$y' = 1 - y^2$	

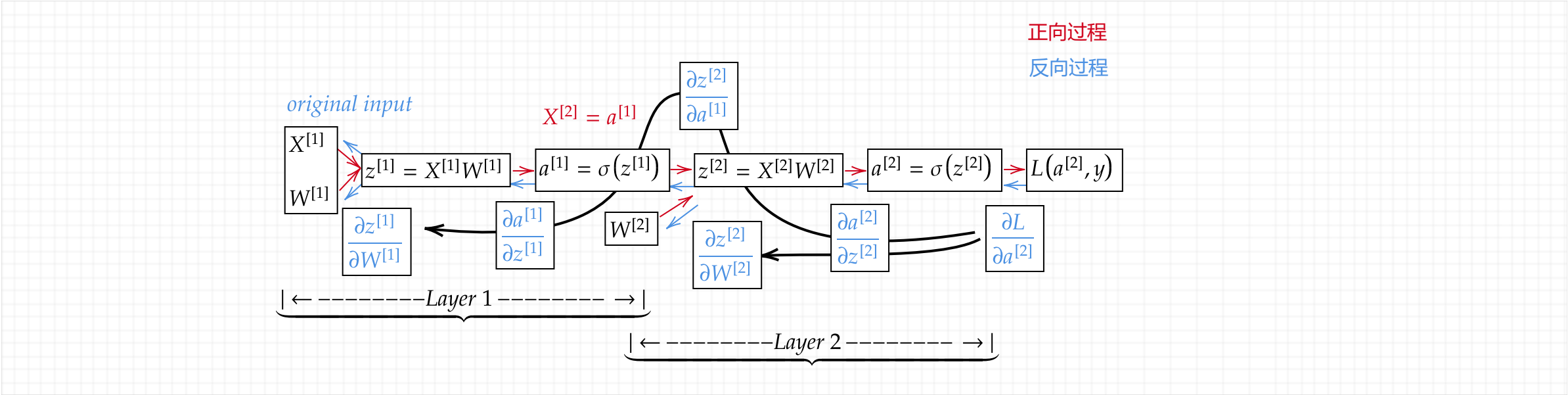
<i>ReLU</i>	$y = \max(0, z)$	$y' = \begin{cases} 0 & z < 0 \\ 1 & z > 0 \\ ? & z = 0 \end{cases}$	
<i>Leaky ReLU</i>	$y = \begin{cases} z & z > 0 \\ 0.01z & z < 0 \\ 0 & z = 0 \end{cases}$	$y' = \begin{cases} 0.01 & z < 0 \\ 1 & z > 0 \\ ? & z = 0 \end{cases}$	

工程上，除了二元分类之外，不推荐用Sigmoid，因为tanh的表现几乎总是更优秀。而对于隐藏层，更多的是选用ReLU, Leaky - ReLU。

```
1 # https://www.cnblogs.com/xiximayou/p/12713081.html
2 # 比较好的实现方式, 值得学习
3 class LeakyReLU():
4     def __init__(self, alpha=0.2):
5         self.alpha = alpha
6
7     def __call__(self, x):
8         return np.where(x >= 0, x, self.alpha * x)
9
10    def gradient(self, x):
11        return np.where(x >= 0, 1, self.alpha)
```

计算过程

以只有一个隐藏层的神经元为例：



而我们要做的就是：

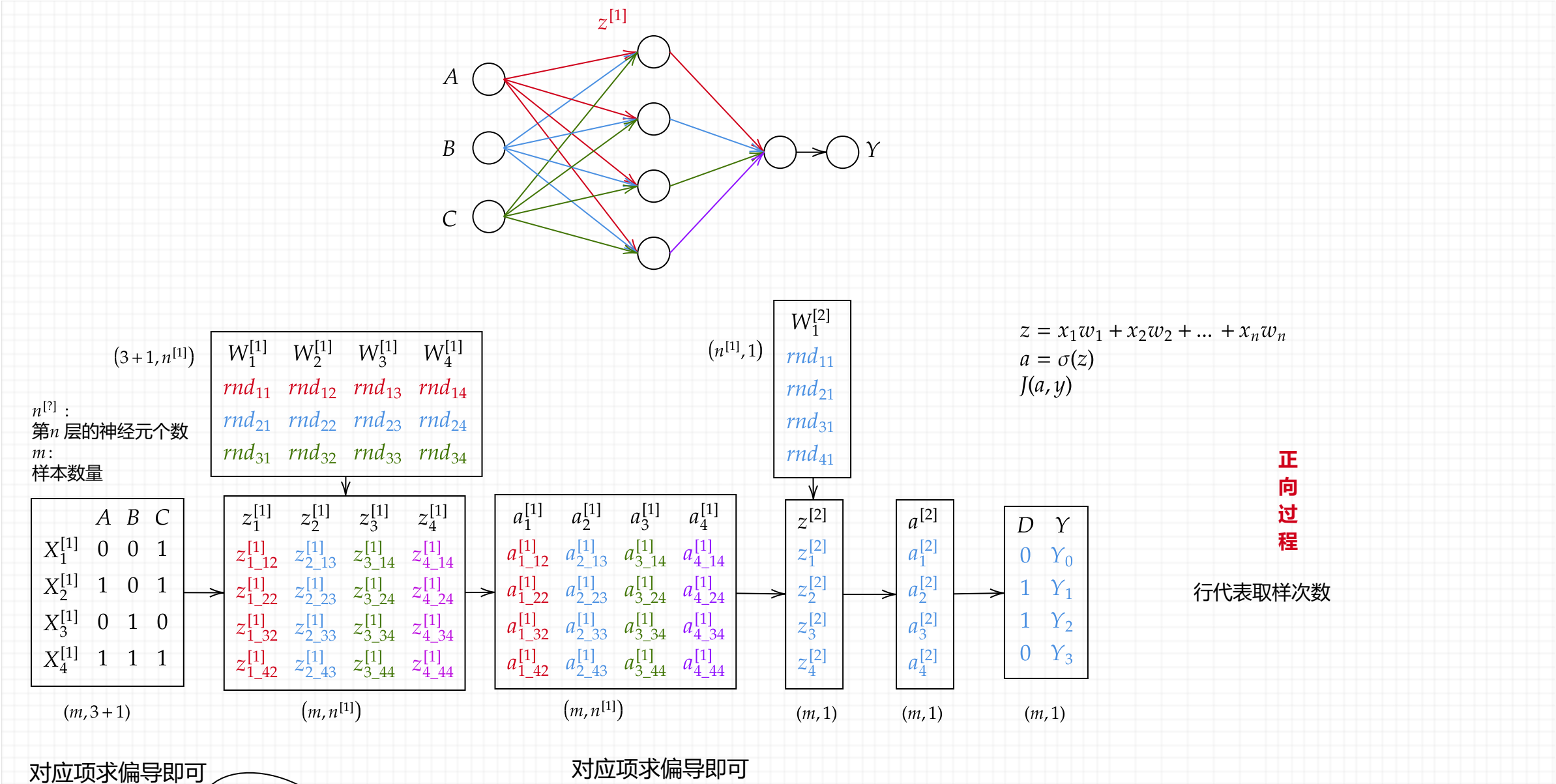
1. 假设两组权重： $W^{[1]}, W^{[2]}$
2. 计算四组隐藏层数据： $z^{[1]}, z^{[2]}, a^{[1]}, a^{[2]}$
3. 推导，并化简反向过程中的偏微分方程： $\partial W^{[1]}, \partial W^{[2]}, \frac{\partial z^{[2]}}{\partial a^{[1]}}$ ，蓝色的别忘掉
4. $a^{[1]}$ 会随每次计算更新
5. 然后就迭代修正误差，好像就是这么简单。

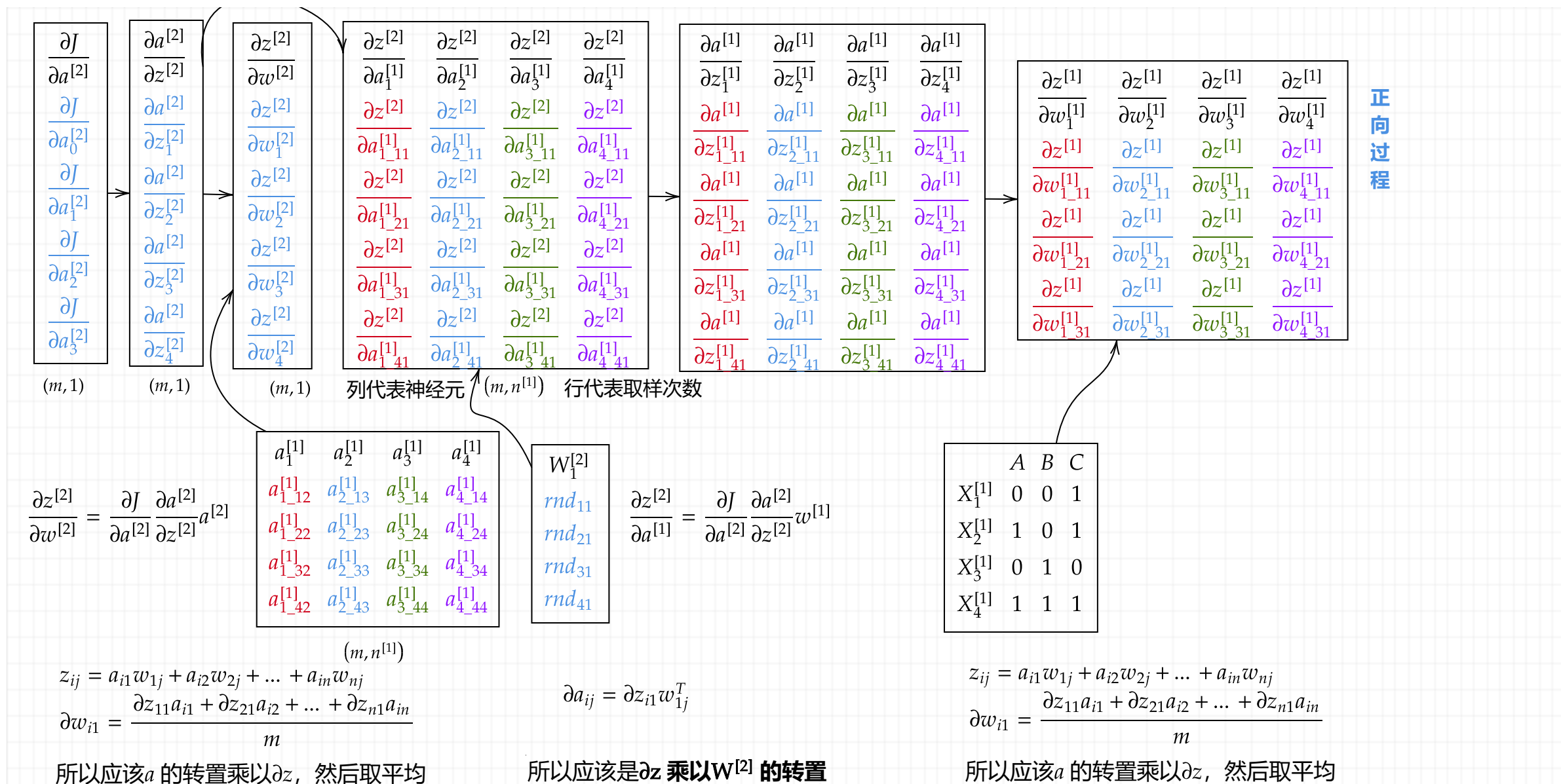
依然是看电影的例子：

A	B	C	D
0	0	1	0

1	0	1	1
0	1	0	1
1	1	1	0
1	0	0	?

从上表可以大致看出，只有当 $A \text{ xor } B = 1$ 时， D 才会参加。由单个神经元去计算的话，我们会发现即使是样本数据，单个神经元也会给出非常离谱的结果0.5。所以我们采用下面的模型（方便起见，我们计算时不再引入常数项）：





对理解 ∂a_1 时, 我们可以想:

对于第一次采样来说: $\partial a_1 = \partial z_1 w_1, \partial a_2 = \partial z_1 w_2, ..., \partial a_n = \partial z_1 w_n$

对于第二次采样来说: $\partial a_1 = \partial z_2 w_1, \partial a_2 = \partial z_2 w_2, \dots, \partial a_n = \partial z_2 w_n$
就很容易理解 $\partial a_{ij} = \partial z_{i1} w_{1j}^T$ 了

代码及解释

摘自【[博客园-西西嘛呦](#)】，数据较上文有变化，但是计算思路是一样的

```
1  # activation.py
2
3  '''
4  梯度在函数调用时就能计算出来
5  这样做的好处可以避免遗漏
6  '''
7  import numpy as np
8
9  class Sigmoid():
10     def __init__(self):
11         pass
12
13     def __call__(self, z):
14         return 1/(1+np.exp(-z))
15
16     def gradient(self, z):
17         a = self.__call__(z)
18         return a*(1-a)
19
20 class Tanh():
21     def __init__(self):
22         pass
23
24     def __call__(self, z):
25         ez = np.exp(z)
26         e_z = np.exp(-z)
27         return (ez-e_z)/(ez+e_z)
```

```

28
29 def gradient(self,z):
30     a = self.__call__(z)
31     return 1- np.power(a,2)
32
33 class ReLU():
34     def __init__(self):
35         pass
36
37     def __call__(self, z):
38         return np.maximum(z,0)
39
40     def gradient(self, z):
41         return np.where(z>0, 1, 0)
42
43 class LeakyReLU(): # leakrelu = LeakyReLU(alpha=0.1)  ## 调用
44     def __init__(self,alpha=0.01):
45         self.alpha = alpha
46
47     def __call__(self, z):
48         return np.where(z>0, 1, self.alpha*z)
49
50     def gradient(self, z):
51         return np.where(z>0, 1, self.alpha)

```

为了简化计算，我们将抛弃一般线性回归中的常数项，重点关注多层神经网络的正向过程和反向过程

```

1 import numpy as np
2 import activation

```

```
3
4
5 def main():
6     np.random.seed(1)
7
8     alpha = 0.03 # 学习率 learning rate
9     sigmoid = activation.Sigmoid()
10
11     a0,y = loadData()          # A,B,C,D
12                                # 1,1,1,0
13                                # 0,1,1,1
14                                # 0,0,1,0
15                                # 1,0,1,1
16                                # 0,0,0,0
17                                # 0,1,0,1
18                                # 1,1,0,0
19                                # 1,0,0,1
20     # 每层节点个数: 10-3, 11-4, 12-1, 可以确定权重矩阵的维度
21     w1 = generateWeights((3, 4))
22     w2 = generateWeights((4, 1))
23
24     for i in range(100000):
25         z1 = np.dot(a0, w1) # 正向过程: 每一级的输出都是下一级的输入
26         a1 = sigmoid(z1) #
27
28         z2 = np.dot(a1, w2)
29         a2 = sigmoid(z2)
30
31     loss = Loss(a2, y) # 反向过程, 由后至前反推一遍
```

```

32     grad_a2 = loss.gradient()
33     grad_z2 = grad_a2 * sigmoid.gradient(z2)
34     grad_w2 = np.dot(a1.T, grad_z2)/len(a1) # 对于权重来说，因为样本数量很多，成本函数对应的是平均值
35
36     grad_a1 = np.dot(grad_z2, w2.T) # 对于a 来说，没有求平均值的说法，因为它本身就包含着采样次数的信息
37     # 这一步非常重要，而且不太好理解，最好能多通过流程图在稿纸上多演算几次运算过程和矩阵乘法
38     grad_z1 = grad_a1 * sigmoid.gradient(z1)
39     grad_w1 = np.dot(a0.T, grad_z1)/len(a0) # 成本函数对应的是平均值
40     # 看反向过程的每一层都有求da, dz, dw。而且基本都是死公式
41
42     w1 -= alpha * grad_w1
43     w2 -= alpha * grad_w2
44
45 # 验证
46 s0 = [
47     [0, 0, 0],
48     [0, 0, 1],
49     [0, 1, 0],
50     [0, 1, 1],
51     [1, 0, 0],
52     [1, 0, 1],
53     [1, 1, 0],
54     [1, 1, 1]]
55 s1 = np.dot(s0, w1)
56 s2 = sigmoid(s1)
57 s3 = np.dot(s2, w2)
58 s4 = sigmoid(s3)
59 print(s4)
60

```

```
61
62
63 def loadData():
64     org_data = np.loadtxt(
65         './data.csv',
66         delimiter=',',
67         usecols=(0, 1, 2, 3),
68         skiprows=1
69     )
70     exp_data = org_data[:, 0:3]
71     res_data = org_data[:, 3:4]
72     return exp_data, res_data
73
74 def generateWeights(shape): # 生成权重矩阵
75     weight = 2 * np.random.random(shape)-1
76     return weight
77
78 class Loss(): # 损失函数 (对于单个样本来说
79     def __init__(self, a: np.array, y: np.array):
80         self.a = a
81         self.y = y
82
83     def __call__(self):
84         a = self.a
85         y = self.y
86         return -(y*np.log(a))-(1-y)*np.log(1-a)
87
88     def gradient(self):
89         a = self.a
```

```
90         y = self.y
91         return -y/a + (1-y)/(1-a)
92
93 if __name__ == "__main__":
94     main()
```