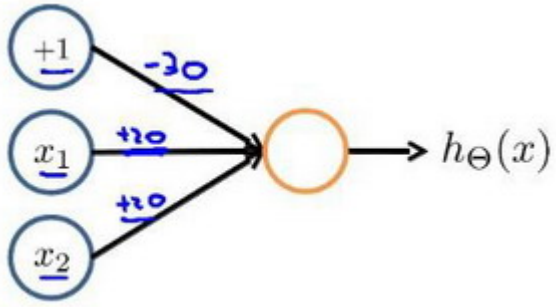
神经网络的学习

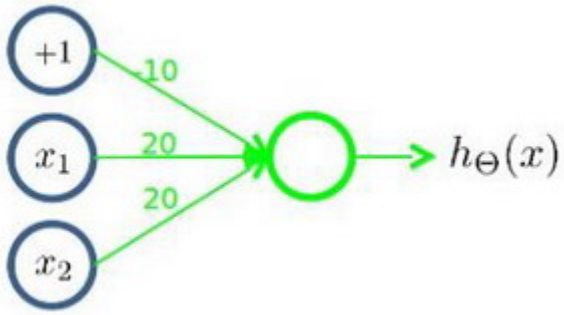
1.神经网络中的二元逻辑计算符

> 就是说这个输入为 0/1 输出为结果 0/1(当然应该是有激活函数，这里是sigmoid函数)

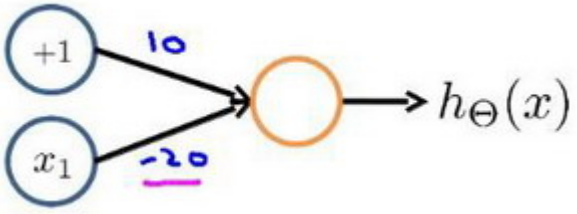
下图的神经元（三个权重分别为-30，20，20）可以被视为作用同于逻辑与（**AND**）：



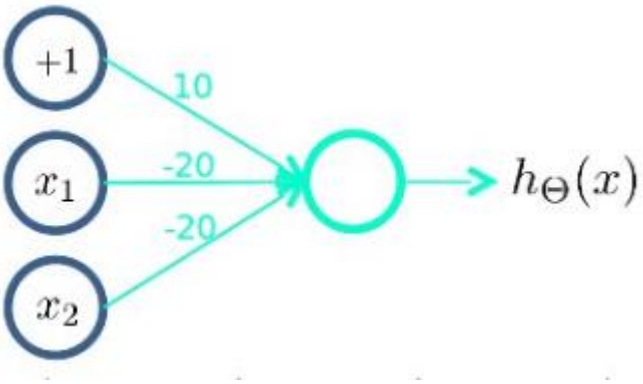
下图的神经元（三个权重分别为-10，20，20）可以被视为作用等同于逻辑或（**OR**）：



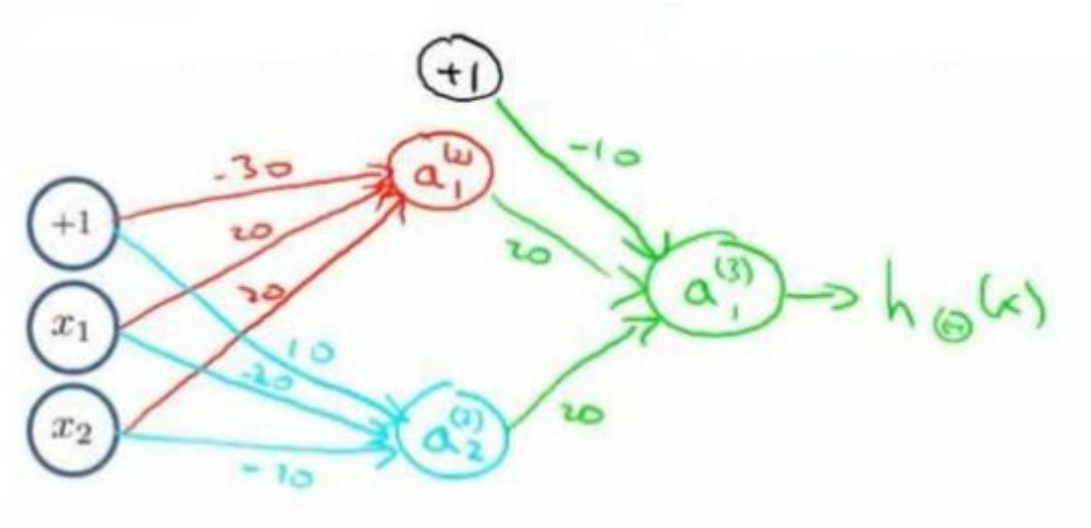
下图的神经元（两个权重分别为 10，-20）可以被视为作用等同于逻辑非（**NOT**）：



我们可以利用神经元来组合成更为复杂的神经网络以实现更复杂的运算。例如我们要实现 **XNOR** 功能（输入的两个值必须一样，均为1或均为0），即 首先构造一个能表达部分的神经元：

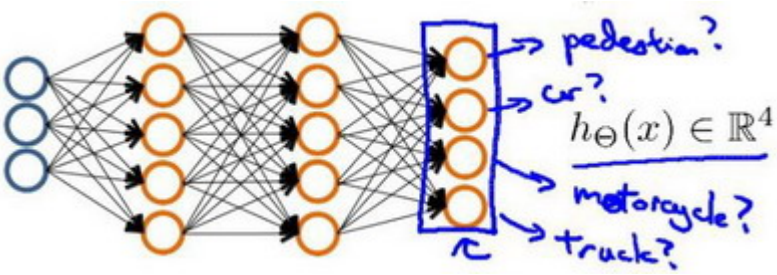


然后将表示 **AND** 的神经元和表示的神经元以及表示 OR 的神经元进行组合：



我们就得到了一个能实现 运算符功能的神经网络。

2.多分类网络



2.1 代价函数

> 和逻辑回归相似，但要复杂点

在逻辑回归中，我们只有一个输出变量，又称标量（**scalar**），也只有一个因变量y，但是在神经网络中，我们可以有很多输出变量，我们的$h_\theta(x)$是一个维度为K的向量，并且我们训练集中的因变量也是同样维度的一个向量，因此我们的代价函数会比逻辑回归更加复杂一些，为：$h_\theta\left(x\right) \in \mathbb{R}^K \left(h_\theta\left(x\right)\right)_i = i^{th}\text{output}$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{k} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + \left(1 - y_k^{(i)}\right) \log\left(1 - \left(h_\Theta\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2$$

这个看起来复杂很多的代价函数背后的思想还是一样的，我们希望通过代价函数来观察算法预测的结果与真实情况的误差有多大，唯一不同的是，对于每一行特征，我们都会给出K个预测，基本上我们可以利用循环，对每一行特征都预测K个不同结果，然后在利用循环在K个预测中选择可能性最高的一个，将其与y中的实际数据进行比较。

正则化的那一项只是排除了每一层θ_0后，每一层的θ 矩阵的和。最里层的循环j循环所有的行（由s_l +1 层的激活单元数决定），循环i则循环所有的列，由该层（s_l层）的激活单元数所决定。即：$h_\theta(x)$与真实值之间的距离为每个样本-每个类输出的加和，对参数进行**regularization**的**bias**项处理所有参数的平方和。

2.2 反向传播算法

2.3 梯度检验

> 应该是用来验证求的梯度对不对（好像使用pytorch的就不需要验证了？）

[# gradient checking（梯度检验）]

2.4 随机初始化

> pytorch如何参数初始化

PYTHON

```
from torch.nn import init
#define the initial function to init the layer's parameters for the ne
def weigth_init(m):
    if isinstance(m, nn.Conv2d):
        init.xavier_uniform_(m.weight.data)
        init.constant_(m.bias.data,0.1)
    elif isinstance(m, nn.BatchNorm2d):
        m.weight.data.fill_(1)
        m.bias.data.zero_()
    elif isinstance(m, nn.Linear):
        m.weight.data.normal_(0,0.01)
        m.bias.data.zero_()
```

```
#Define Network
model = Net(args.input_channel,args.output_channel)
model.apply(weigth_init)
```

> 或者使用另一种方式进行参数初始化

PYTHON

```python
def initNetParams(net):
    '''Init net parameters.'''
    for m in net.modules():
        if isinstance(m, nn.Conv2d):
            init.xavier_uniform(m.weight)
            if m.bias:
                init.constant(m.bias, 0)
        elif isinstance(m, nn.BatchNorm2d):
            init.constant(m.weight, 1)
            init.constant(m.bias, 0)
        elif isinstance(m, nn.Linear):
            init.normal(m.weight, std=1e-3)
            if m.bias:
                init.constant(m.bias, 0)

initNetParams(net)
```