

第四周学习笔记

2022-05-05

第一章 神经网络入门

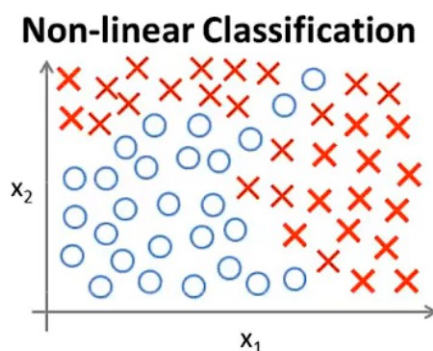
1.1 非线性模型

神经网络实际上是一个相对古老的算法，但后面沉寂了一段时间，但是现在又慢慢成为机器学习问题的首选技术。

既然我们已经有了线性回归和 *logistic* 回归了，为什么还需要研究神经网络这个算法？

为了阐述研究神经网络算法的目的，我们首先来看几个例子。这几个例子都需要学习复杂的非线性假设。

假设有一个监督学习的分类问题，训练集如下图所示：



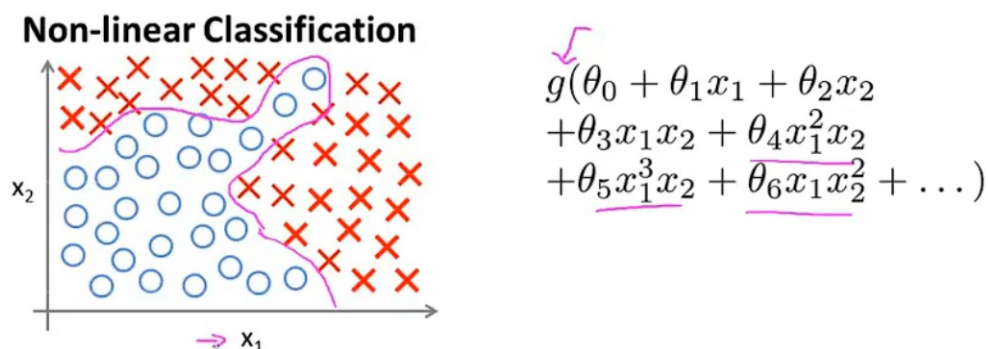
当使用 *logistic* 回归时，我们可以构造回归函数为：

$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \dots)$$

其中，这里的 g 仍然是 *sigmoid* 函数，其中可以包含很多像上面的多项式。

分类问题依旧可以解决。但是有一个很明显的弊端，那就是当数据的特征太多时，计算的负荷会非常大。

比如下图所示的决策边界：



当只有两个特征，比如 x_1 和 x_2 时，我们可以应用的很好，计算也比较简单。之前的练习题我们也已经看到过，使用非线性的多项式项，能够帮助我们建立更好的分类模型。

假设我们有非常多的特征，比如之前做的预测房价的问题，有大于 100 个变量，我们希望用这 100 个特征来构建一个非线性的多项式模型，结果将是数量非常惊人的特征组合，即便我们只采用两两特征的组合， $x_1 x_2 + x_1 x_3 + x_1 x_4 + \dots + x_2 x_3 + x_2 x_4 + \dots + x_{99} x_{100}$ ，复杂度为 $o(n^2)$ ，换句话说，我们也会有接近 5000 个组合而成的特征。这对于一般的逻辑回归来说需要计算的特征太多了。最后产生的问题也就很明确，那就是计算的负荷很大。

当然，我们也有对应的解决办法。例如我们可以减少数据的特征，比如只拟合二次方， $x_1^2 + x_2^2 + \dots + x_{100}^2$ ，这样我们就能把特征的数量减少到 100 个，但是拟合效果肯定不理想，因为漏掉了一些相关项的特征，最后可能导致有些特殊情况无法实现。

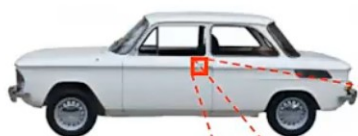
再引入一个例子，假设我们希望训练出一个模型来识别视觉对象（例如识别一张图片上是否是一辆汽车），我们该怎样实现这个问题？

首先需要明白的是，当我们人眼看到一个事物时，例如车的门把手，对应的计算机看到的却是一个数据矩阵，或者表示像素强度值的网格。

如下图所示：

What is this?

You see this:

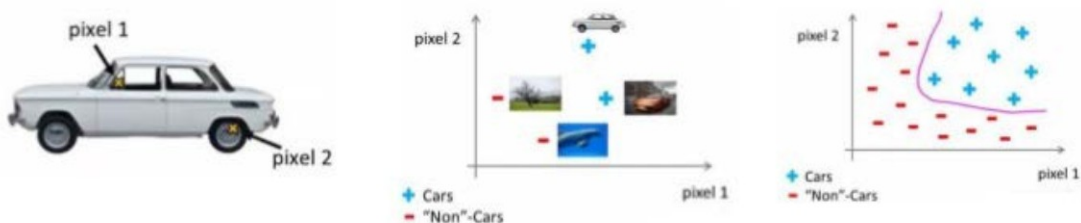


But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	75	65
20	43	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

我们拿到了很多汽车的图片和很多非汽车的图片，然后利用这些图片上一个个像素的值（饱和度或亮度）来作为特征。

假如我们只选用灰度图片，每个像素则只有一个值（而非 *RGB* 值），我们可以选取图片上的两个不同位置上的两个像素，训练出一个逻辑回归算法，然后利用选取的两个像素的值来判断图片上是否是汽车，具体步骤如下图所示：



假设我们采用的都是 50×50 像素的小图片，并且我们将所有的像素视为特征，则会有 2500 个特征，如果我们要进一步将所有特征两两组合构成一个多项式模型，则会有约接近三百万个特征。根据普通的逻辑回归模型，并不能有效地处理这么多的特征，这时候我们就需要神经网络来解决。

1.2 神经元和大脑

通过上面的学习我们已经知道，神经网络是计算量有些偏大的算法。然而大概由于近些年计算机的运行速度变快，才足以真正运行起大规模的神经网络。

当研究人员想模拟大脑时，是指想制造出与人类大脑作用效果相同的机器模型。大脑可以学会去以看而不是听的方式处理图像，学会处理我们的触觉。

这种情况下与人体进行类比可能会方便理解。人体的脑组织可以处理光、声或触觉信号，那么也许存在一种学习算法，可以同时处理视觉、听觉和触觉，而不是需要运行上千个不同的程序，或者上千个不同的算法来做这些大脑所完成的成千上万的美好事情。也许我们需要做的就是找出一些近似的或实际的大脑学习算法，然后像人类大脑一样通过自学掌握如何处理这些不同类型的数据。在很大的程度上，可以猜想，如果我们把任何一种传感器接入到大脑的几乎任何一个部位的话，大脑就会学会处理它。

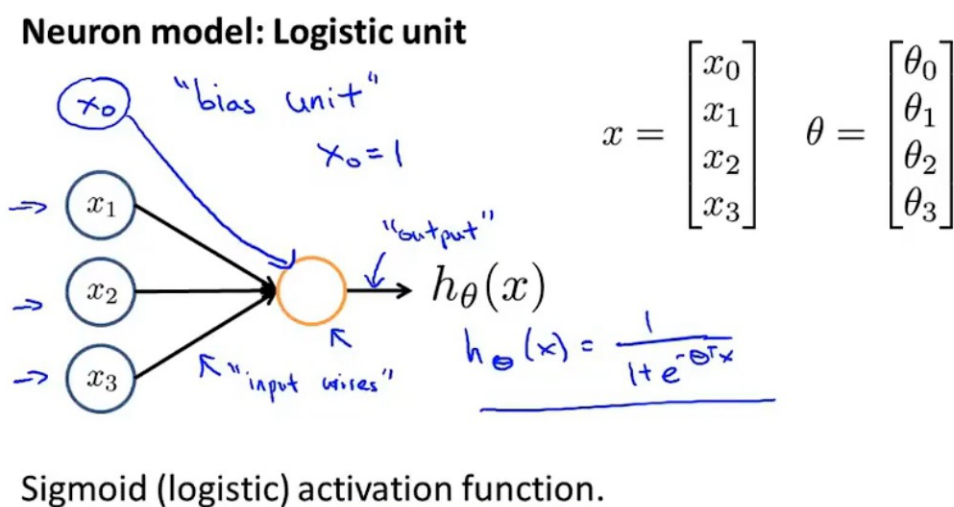
1.3 模型表示

1.3.1 模型的理解一

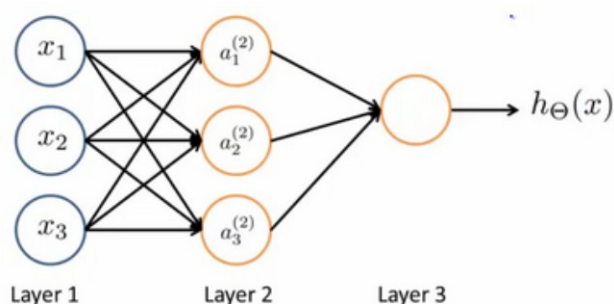
为了构建神经网络模型，我们需要首先思考大脑中的神经网络是怎样的？

大脑中的每一个神经元都可以被认为是一个处理单元（神经核），它含有许多输入（树突），并且有一个输出（轴突）。神经网络是大量神经元相互链接并通过电脉冲来交流的一个网络。

神经网络模型建立在很多神经元之上，每一个神经元又是一个个学习模型。这些神经元（也叫激活单元，*activation unit*）采纳一些特征作为输入，并且根据本身的模型提供一个输出。下图是一个以逻辑回归模型作为自身学习模型的神经元示例，在神经网络中，参数又可被称为权重（*weight*）。



我们设计出了类似于神经元的神经网络，效果如下：



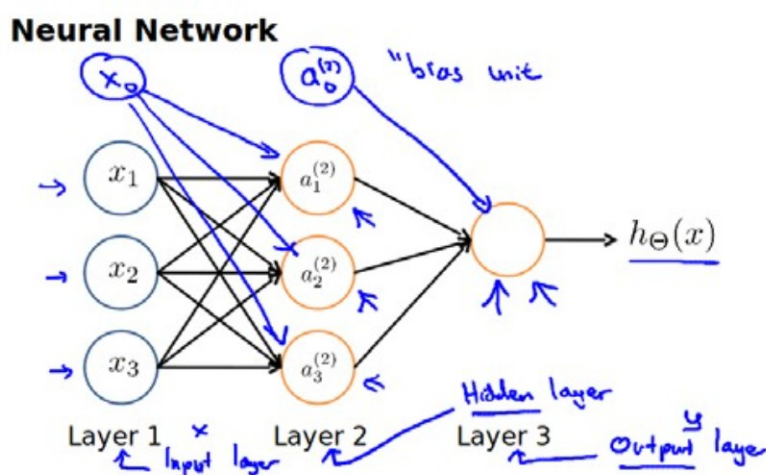
其中 x_1, x_2, x_3 是输入单元 (*input units*), 有必要的时候, 我们会增加一个额外的节点 x_0 , 这个额外的节点我们称为偏置单元或者偏置神经元, 我们将原始数据输入给它们 (x_0 总是等于 1)。 a_1, a_2, a_3 是中间单元, 它们负责将数据进行处理, 然后呈递到下一层。最后是输出单元, 它负责计算 $h_{\Theta}(x)$ 。

结合之前学习的逻辑回归, 我们知道 $h_{\theta}(x) = \frac{1}{1+e^{-\theta^T x}}$ 。

这是一个带有 *sigmoid* 或者 *logistic* 激活函数的人工神经元。

神经网络模型是许多逻辑单元按照不同层级组织起来的网络, 每一层的输出变量都是下一层的输入变量。

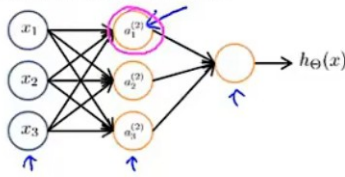
下图为一个 3 层的神经网络, 第一层成为输入层 (*Input Layer*), 最后一层称为输出层 (*Output Layer*), 中间一层成为隐藏层 (*Hidden Layers*)。但实际上任何非输入层和非输出层的层就被称为隐藏层。我们为每一层都增加一个偏置单元 (*bias unit*)。



为了解释这个神经网络具体的计算步骤, 里面还有一些记号需要解释。

$a_i^{(j)}$ 代表第 j 层的第 i 个激活单元; $\theta^{(j)}$ 代表从第 j 层映射到第 $j+1$ 层时的权重的矩阵, 例如 $\theta^{(1)}$ 代表从第一层映射到第二层的权重的矩阵。其尺寸为: 以第 $j+1$ 层的激活单元数量为行数, 以第 j 层的激活单元数加一为列数的矩阵。

Neural Network



$\rightarrow a_i^{(j)}$ = “activation” of unit i in layer j
 $\rightarrow \Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j + 1$

例如：上图所示的神经网络中 $\theta^{(1)}$ 的尺寸为 3×4 。

对于上图所示的模型，激活单元和输出分别表达为：

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$

$$h_\theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

其中， g 称为 *sigmoid* 函数或者 *sigmoid* 激活函数，也叫做 *logistic* 激活函数。

上面进行的讨论中只是将特征矩阵中的一行（一个训练实例）喂给了神经网络，我们需要将整个训练集都喂给我们的神经网络算法来学习模型。

我们可以知道：每一个 a 都是由上一层所有的 x 和对应的参数 θ 所决定的。

总结一下，其中的神经网络定义了函数 $h_\theta(x)$ ，从输入 x 到输出 y 的映射，这些假设被参数化，我们可以将参数记为 θ ，这样一来改变 θ 就能得到不同的假设，就能得到不同的函数，比如从 x 到 y 的映射。

我们把这样从左到右的算法称为前向传播算法 (*FORWARD PROPAGATION*)。

其中， x ， θ ， a 的矩阵分别表示为：

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \theta = \begin{bmatrix} \theta_{10}, \dots, \dots, \dots \\ \dots, \dots, \dots, \dots \\ \dots, \dots, \dots, \theta_{33} \end{bmatrix}, a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (1.1)$$

根据上面的神经网络，我们不难看出等式关系： $\theta \times X = a$ 。

1.3.2 模型的理解二

在前面的笔记中讲解了怎样用数学来定义或者计算神经网络的假设函数，在这一节中，我们将会尝试学习如何高效进行计算，并展示了一个向量化的实现方法。

以上面的神经网络为例，我们计算假设的步骤如下：

$$\begin{aligned}
 a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\
 a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\
 a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\
 h_\theta(x) &= a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})
 \end{aligned}$$

通过这些方程，我们计算出三个隐藏单元的激活值，然后利用这些值计算出最终输出假设函数 $h_\theta(x)$ 的值接下来，我们定义一些额外的项。

$$\begin{aligned}
 z_1^{(2)} &= \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \\
 z_2^{(2)} &= \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \\
 z_3^{(2)} &= \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3
 \end{aligned}$$

这些 z 值都是线性组合，将某个特定的神经元的输入值 x_0, x_1, x_2, x_3 的加权线性组合而成。

利用向量化的方法会使得计算更为简便。以上面的神经网络为例，试着计算第二层的值。

$$\begin{aligned}
 X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, z^{(2)} &= \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} \\
 z^{(2)} &= \Theta^{(1)} x \\
 a^{(2)} &= g(z^{(2)})
 \end{aligned} \tag{1.2}$$

$$\Theta = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix} \tag{1.3}$$

所以我们接下来得到：

$$g\left(\begin{bmatrix} \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \\ \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \\ \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \end{bmatrix}\right) = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \tag{1.4}$$

我们令 $z^{(2)} = \Theta^{(1)}x$, 则 $a^{(2)} = g(z^{(2)})$, 计算后添加 $a_0^{(2)} = 1$ 。计算输出的值为:

$$g\left(\begin{bmatrix} \Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} & \Theta_{13}^{(2)} \end{bmatrix} \times \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix}\right) \quad (1.5)$$

$$= g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right) = h_{\theta}(x) \quad (1.6)$$

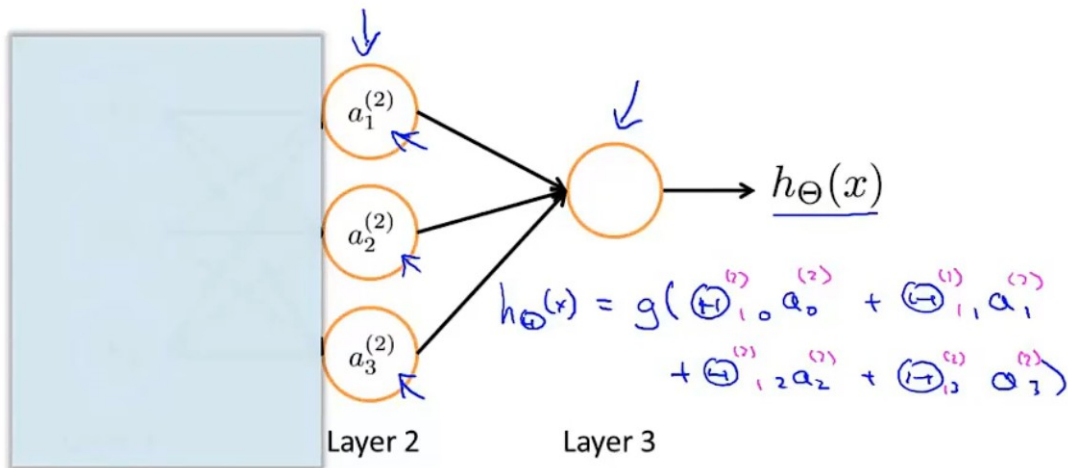
我们令 $z^{(3)} = \Theta^{(2)}a^{(2)}$, 则 $h_{\theta}(x) = a^{(3)} = g(z^{(3)})$ 。这只是针对训练集中一个训练实例所进行的计算。如果我们要对整个训练集进行计算, 我们需要将训练集特征矩阵进行转置, 使得同一个实例的特征都在同一列里。即:

$$z^{(2)} = \Theta^{(1)} \times X^T$$

$$a^{(2)} = g(z^{(2)})$$

为了更好的了解神经网络的工作原理, 我们先把左半部分遮住:

Neural Network learning its own features



右半部分其实就是以 $a_0 a_1 a_2 a_3$, 按照 *Logistic Regression* 的方式输出的 $h_{\theta}(x)$ 。

其实神经网络就像是 *logistic regression*, 只不过我们把 *logistic regression* 中的输入向量 (x_1, x_2, x_3) 变成了中间层的 $a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$, 即:

$$h_{\theta}(x) = g(\Theta_0^{(2)}a_0^{(2)} + \Theta_1^{(2)}a_1^{(2)} + \Theta_2^{(2)}a_2^{(2)} + \Theta_3^{(2)}a_3^{(2)})$$

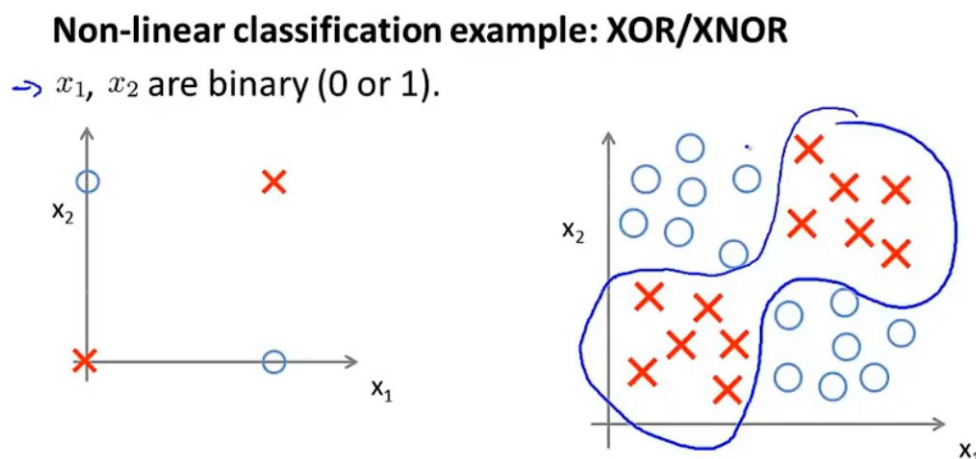
我们可以把 $a_0 a_1 a_2 a_3$ 看成更为高级的特征值, 也就是 $x_0 x_1 x_2 x_3$ 的进化体, 并且它们是由 x 和 θ 决定的, 因为是梯度下降的, 所以 a 是变化的, 并且变得越来越厉害, 所以这些更高级的特征值远比仅仅将 x 次方厉害, 也能更好的预测新数据。这就是神经网络相比于逻辑回归和线性回归的优势。

1.4 特征与直觉理解

1.4.1 第一节

本节将通过一个例子来说明神经网络是怎样计算复杂非线性函数的输入，通过学习希望可以搞明白为什么神经网络可以用来学习复杂的非线性非线性假设模型。

考虑的下面的问题:



我们有 x_1 和 x_2 两个输入特征，它们都是二进制的，二进制只能取 0 或者 1，所以 x_1 和 x_2 只可能是 0 或者 1。在这个例子中，只画了两个正样本和两个负样本，但可以把它看作是复杂的机器学习问题的一个简化版本。我们想做的是学习一个非线性的判断边界来区分这些正样本和负样本。

从本质上讲，神经网络能够通过学习得出其自身的一系列特征。在普通的逻辑回归中，我们被限制为使用数据中的原始特征 $x_1, x_2, x_3, \dots, x_n$ ，我们虽然可以使用一些二项式项来组合这些特征，但是我们仍然受到这些原始特征的限制。在神经网络中，原始特征只是输入层，在我们上面三层的神经网络例子中，第三层也就是输出层做出的预测利用的是第二层的特征，而非输入层中的原始特征，我们可以认为第二层中的特征是神经网络通过学习后自己得出的一系列用于预测输出变量的新特征。

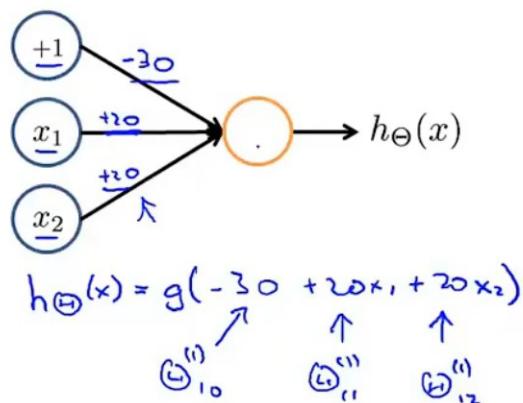
神经网络中，单层神经元（无中间层）的计算可用来表示逻辑运算，比如逻辑与 (AND)、逻辑或 (OR)。

举例说明：逻辑与 (AND)。下图中是神经网络的设计与 *output* 层表达式，右边上部分是 *sigmoid* 函数，下半部分是真值表。

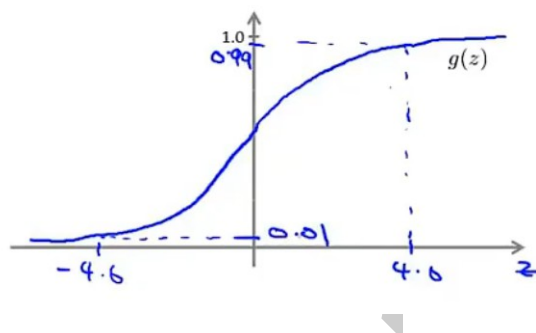
Simple example: AND

$$\rightarrow x_1, x_2 \in \{0, 1\}$$

$$\rightarrow y = x_1 \text{ AND } x_2$$



其中 $\theta_0 = -30, \theta_1 = 20, \theta_2 = 20$ ，我们的输出函数 $h_{\Theta}(x) = g(-30 + 20x_1 + 20x_2)$ 。
我们知道 $g(x)$ 的图像是：

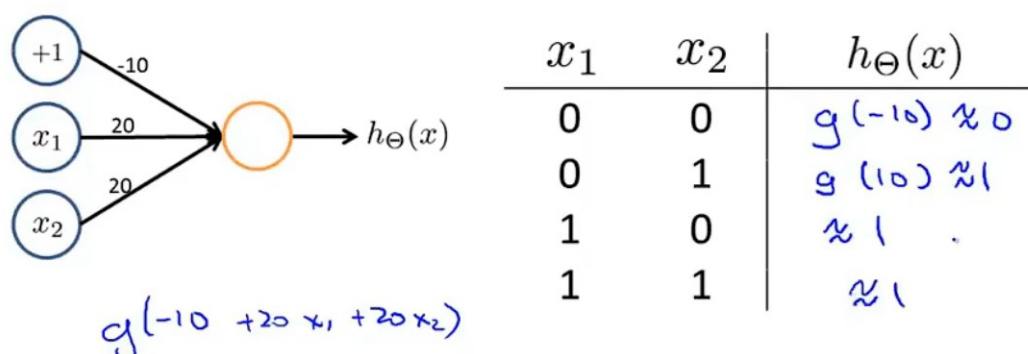


试试 x_1 和 x_2 对应的四种输入组合，看看模型的结果分别有什么：

x_1	x_2	$h_{\Theta}(x)$
0	0	$g(-30) \approx 0$
\rightarrow 0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	

所以我们有： $h_{\Theta}(x) \approx x_1 \text{ AND } x_2$ 。

接下来再介绍一下 OR 函数：

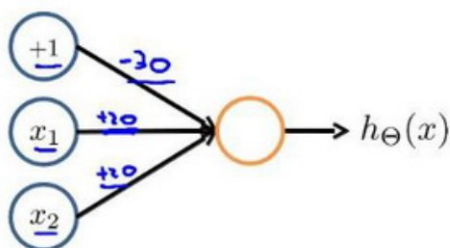
Example: OR function

OR 与 AND 整体一样，区别只在于的取值不同。

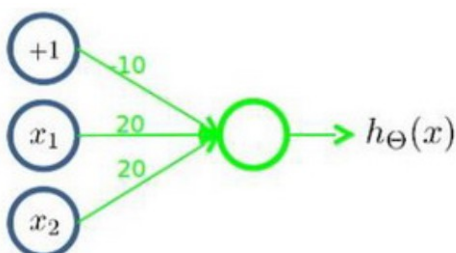
1.4.2 第二节

在上节的学习，介绍了神经网络如何被用来计算函数 AND 和函数 OR ，其中涉及到的 x_1 和 x_2 都是二进制数。

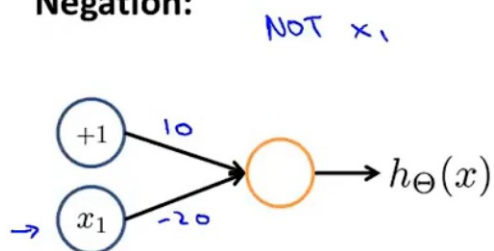
下图的神经元（三个权重分别为-30，20，20）可以被视为作用同于逻辑与（ AND ）：



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 ）：

Negation:

x_1	$h_{\Theta}(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

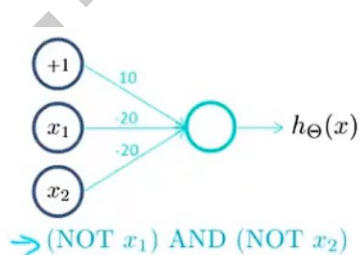
$$h_{\Theta}(x) = g(10 - 20x_1)$$

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。

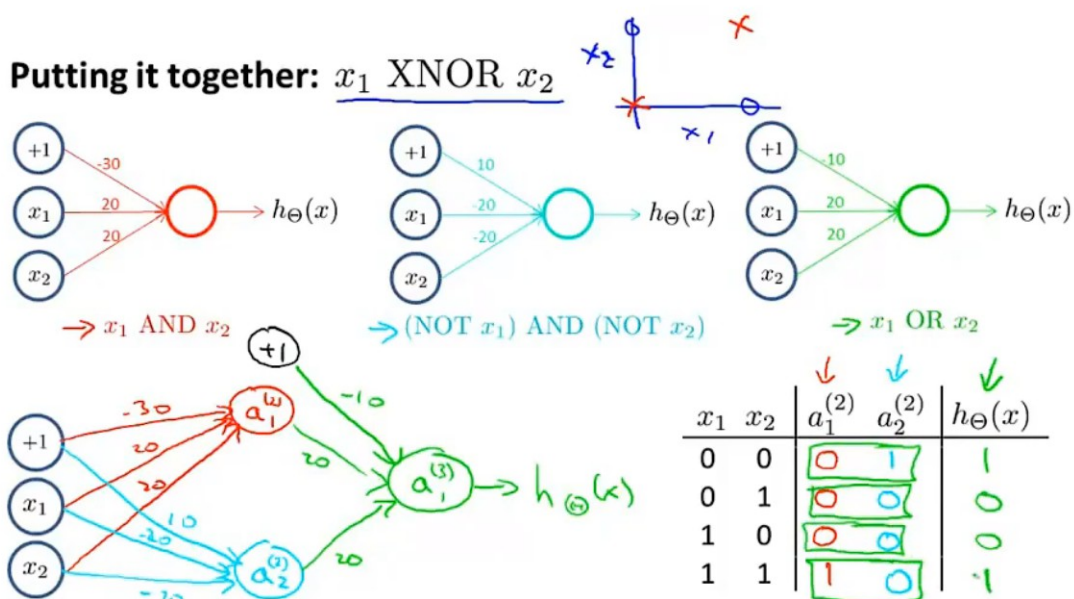
例如我们要实现 $XNOR$ 功能（输入的两个值必须一样，均为 1 或均为 0），即：

$$XNOR = (x_1 \text{ AND } x_2) \text{ OR } ((NOT \ x_1) \text{ AND } (NOT \ x_2))$$

首先构造一个能表达 $(NOT \ x_1) \text{ AND } (NOT \ x_2)$ 部分的神经元：



然后将表示 AND 的神经元和表示 $(NOT \ x_1) \text{ AND } (NOT \ x_2)$ 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 $XNOR$ 运算符功能的神经网络。按这种方法我们可以逐渐构造出越来越复杂的函数，也能得到更加厉害的特征值。这就是神经网络的厉害之处。

1.5 多元分类

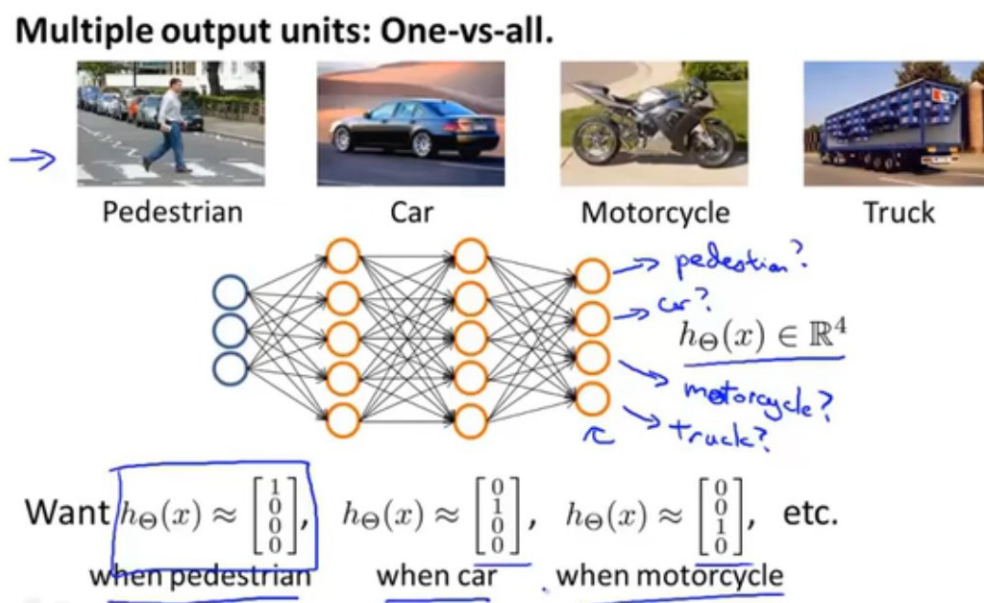
当我们有不止两种分类时，也就是 $y = 1, 2, 3, \dots$ ，当我们遇到这种情况的时候，该怎么办？

我们继续拿计算机视觉来举例子，识别路人、汽车、摩托车和卡车的图像，在输出层我们应该有 4 个值，那么我们该如何训练神经网络模型？

例如，第一个值为 1 或 0 用于预测是否是行人，第二个值用于判断是否为汽车。

输入向量 x 有三个维度，两个中间层，输出层 4 个神经元分别用来表示 4 类，也就是每一个数据在输出层都会出现 $[a \ b \ c \ d]^T$ ，且 a, b, c, d 中仅有一个为 1，表示当前类。

下面是该神经网络的可能结构示例：



现在我们有四个逻辑回归分类器，它们每一个都将识别图片中的物体是否是四种类型中的一种， $a = 1$ 则输出判断是路人， $b = 1$ 则输出判断是汽车， $c = 1$ 则输出判断是摩托车， $d = 1$ 则输出判断是卡车。

重新排版一下，下图就是四输出单元的神经网络：

Multiple output units: One-vs-all.



Want $h_{\Theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $h_{\Theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, etc.
 when pedestrian when car when motorcycle

Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$

$y^{(i)}$ one of $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$
 pedestrian car motorcycle truck

在之前的学习中我们曾经用一个整数 y 作为输出的分类标签，其中 y 可以取 1, 2, 3, 4。现在我们不用 y 表示，我们用 $y^{(i)}$ 表示：

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (1.7)$$

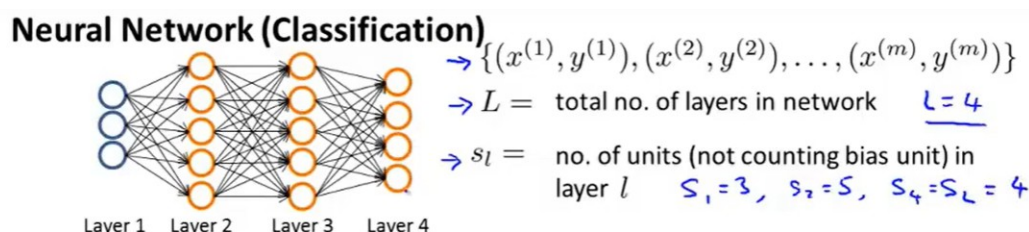
$y^{(i)}$ 的值取决于对应的图像 $x^{(i)}$ 。那么一个训练样本将由一组 $(x^{(i)}, y^{(i)})$ 组成，其中 $x^{(i)}$ 就是四种物体其中一种的图像，而 $y^{(i)}$ 就是这些向量中的一个。

我们希望找到一个方法，让神经网络输出一些数值，输出值 $h_{\theta}(x^{(i)})$ 约等于 $y^{(i)}$ ，并且 $h_{\theta}(x^{(i)})$ 和 $y^{(i)}$ 在该例子中，都是四维向量，分别代表不同的类别。

第二章 神经网络的学习

2.1 代价函数

假设我们有一个与下图类似的神经网络结构，再假设我们有一个这样的数据集，神经网络结构如下所示：



在图中， s_l 表示第 l 层的单元数，也就是神经元的个数，这其中不包括第 l 层的偏置单元。首先引入一些便于稍后讨论的新标记方法。

假设神经网络的训练样本有 m 个，每个训练样本包含一组输入 x 和一组输出 y ， L 表示神经网络层数， S_l 表示每层的神经元个数， S_l 表示输出层神经元个数， S_L 代表最后一层中处理单元的个数。

将神经网络的分类定义为两种情况：二类分类和多类分类。

二类分类： $S_L=0$ ， $y = 0$ or $y = 1$ 分别对应其中一类；

K 类分类： $S_L = k$ ， $y_i = 1$ 表示分到第 i 类 ($k > 2$)

Binary classification

$y = 0$ or $1 \leftarrow$

1 output unit \leftarrow

$$h_{\Theta}(x) \in \mathbb{R}$$

$$S_L = 1, \quad \underline{K=1}$$

Multi-class classification (K classes)

$$y \in \mathbb{R}^K \quad \text{E.g.} \quad \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \leftarrow$$

pedestrian car motorcycle truck

K output units

$$h_{\Theta}(x) \in \mathbb{R}^k$$

$$S_L = K \quad (k \geq 3)$$

我们回顾逻辑回归问题中我们的代价函数为：

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

在逻辑回归中，我们只有一个输出变量，又称标量，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：

$$h_{\theta}(x) \in \mathbb{R}^K$$

$$(h_{\theta}(x))_i = i^{th} \text{ output}$$

$(h_{\theta}(x))_i$ 表示的意思为第 i 个输出，为 K 维向量。

因此我们定义神经网络的代价函数为：

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

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减去每个类输出的加和，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

之前我们在计算神经网络预测结果的时候我们采用了一种正向传播方法，我们从第一层开始正向一层一层进行计算，直到最后一层的 $h_{\theta}(x)$ 。

现在，为了计算代价函数的偏导数 $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ 以及代价函数的值。

如何计算这些偏导项？我们从只有一个训练样本的情况开始说起。假设我们的整个训练集只包含一个训练样本，把这样一个训练样本记为 (x, y) ，我们先粗略看一下使用这个训练样本来计算的顺序。首先我们应用前向传播方法来计算一下在给定输出的时候，

假设函数是否会真的输出结果。我们的神经网络是一个四层的神经网络，其中 $K = 4$ (需要预测四个不同的结果)， $S_L = 4$ (最后一层中处理单元的个数)， $L = 4$ (神经网络层数为四)，我们的计算步骤如下：

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)}) \quad \text{add}(a_0^{(2)})$$

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$a^{(3)} = g(z^{(3)}) \quad \text{add}(a_0^{(3)})$$

$$z^{(4)} = \Theta^{(3)} a^{(3)}$$

$$a^{(4)} = h_{\Theta}(x) = g(z^{(4)})$$

$a^{(1)}$ 就是第一层的激活值，也就是输入层在的地方，所以假设它为 x 。 $\text{add}(a_0^{(2)})$ 和 $\text{add}(a_0^{(3)})$ 的意思是分别在第二层和第三层添加一个偏置单元。

所以这样我们就实现了把前向传播向量化，这使得我们可以计算神经网络里的每一个神经元的激活值。

Gradient computation

Given one training example (x, y) :

Forward propagation:

$$\underline{a^{(1)}} = \underline{x}$$

$$\rightarrow z^{(2)} = \Theta^{(1)} a^{(1)}$$

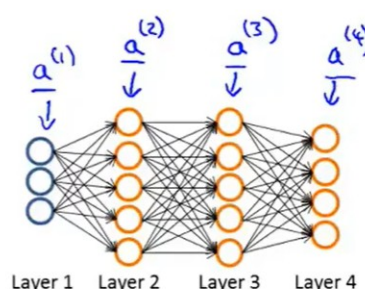
$$\rightarrow a^{(2)} = g(z^{(2)}) \quad (\text{add } \underline{a_0^{(2)}})$$

$$\rightarrow z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$\rightarrow a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$

$$\rightarrow z^{(4)} = \Theta^{(3)} a^{(3)}$$

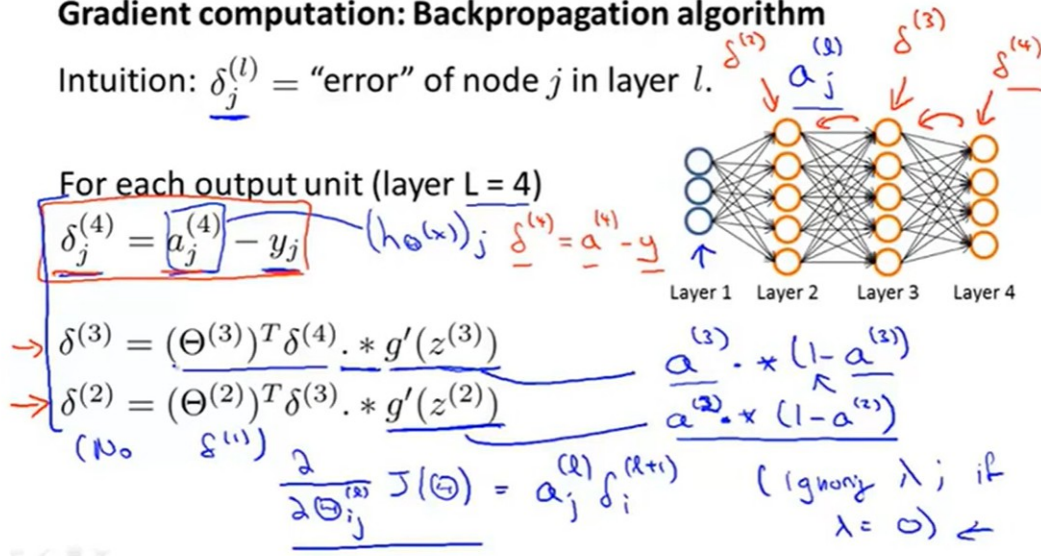
$$\rightarrow \underline{a^{(4)}} = \underline{h_{\Theta}(x)} = g(z^{(4)})$$



接下来，为了计算倒数项，我们将采用一种反向传播算法，也就是首先计算最后一层的误差，然后再一层一层反向求出各层的误差，直到倒数第二层，我们的实现步骤如下图所示：

Gradient computation: Backpropagation algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l .



我们从最后一层的误差开始计算，误差是激活单元的预测值 ($a^{(4)}$) 与实际值 (y^k) 之间的误差， $k = 1, 2, \dots, k$ 。我们用 δ 来表示误差，则：

$$\delta^{(4)} = a^{(4)} - y$$

我们利用这个误差值来计算前一层的误差，有：

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} * g'(z^{(3)})$$

其中 $g'(z^{(3)})$ 是 S 形函数的导数， $g'(z^{(3)}) = a^{(3)} * (1 - a^{(3)})$ 。而 $(\Theta^{(3)})^T \delta^{(4)}$ 则是权重导致的误差的和。

下一步是继续计算第二层的误差：

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} * g'(z^{(2)})$$

因为第一层是输入变量，不存在误差。

我们有了所有的误差的表达式后，便可以计算代价函数的偏导数了，假设 $\lambda = 0$ ，即我们不做任何正则化处理时有：

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

重要的是清楚地知道上面式子中上下标的含义：

l 代表目前所计算的是第几层； j 代表目前计算层中的激活单元的下标，也将是下一层的第 j 个输入变量的下标； i 代表下一层中误差单元的下标，是受到权重矩阵中第 i 行影响的下一层中的误差单元的下标。

如果我们考虑正则化处理，并且我们的训练集是一个特征矩阵而非向量。在上面的特殊情况中，我们需要计算每一层的误差单元来计算代价函数的偏导数。在更为一般的情况下，我们同样需要计算每一层的误差单元，但是我们需要为整个训练集计算误差单元，此时的误差单元也是一个矩阵，我们用 $\Delta_{ij}^{(l)}$ 来表示这个误差矩阵。第 l 层的第 i 个激活单元受到第 j 个参数影响而导致的误差。

我们的算法表示为：

```
for i = 1 : m{
    set a(i) = x(i)
    perform forward propagation to complete a(i) l = 1, 2, 3, ..., L
    using  $\delta^{(L)} = a^{(L)} - y^i$ 
    perform back propagation to compute all-previous layer error vector
     $\Delta_{ij}^{(l)} =: \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{l+1}$ 
}
```

即首先用正向传播方法计算出每一层的激活单元，利用训练集的结果与神经网络预测的结果求出最后一层的误差，然后利用该误差运用反向传播法计算出直至第二层的所有误差。

在求出了 $\Delta_{ij}^{(l)}$ 之后，我们便可以计算代价函数的偏导数了，计算方法如下：

$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \quad \text{if } j \neq 0$$

$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} \quad \text{if } j = 0$$

最后我们得出， $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$ 。

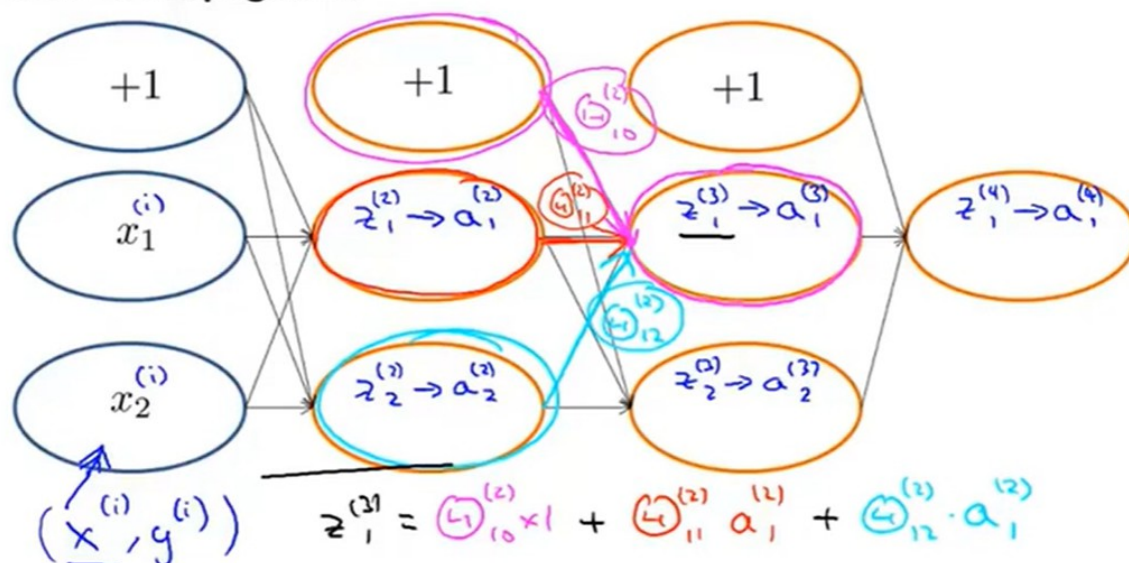
2.3 反向传播算法的直观理解

第一印象通常是，这个算法需要那么多繁杂的步骤，简直是太复杂了，实在不知道这些步骤，到底应该如何合在一起使用。

相比于线性回归算法和逻辑回归算法而言，从数学的角度上讲，反向传播算法似乎并不简洁。

前向传播算法的示意图如下所示：

Forward Propagation



反向传播算法做的是：

What is backpropagation doing?

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

(x⁽ⁱ⁾, y⁽ⁱ⁾)

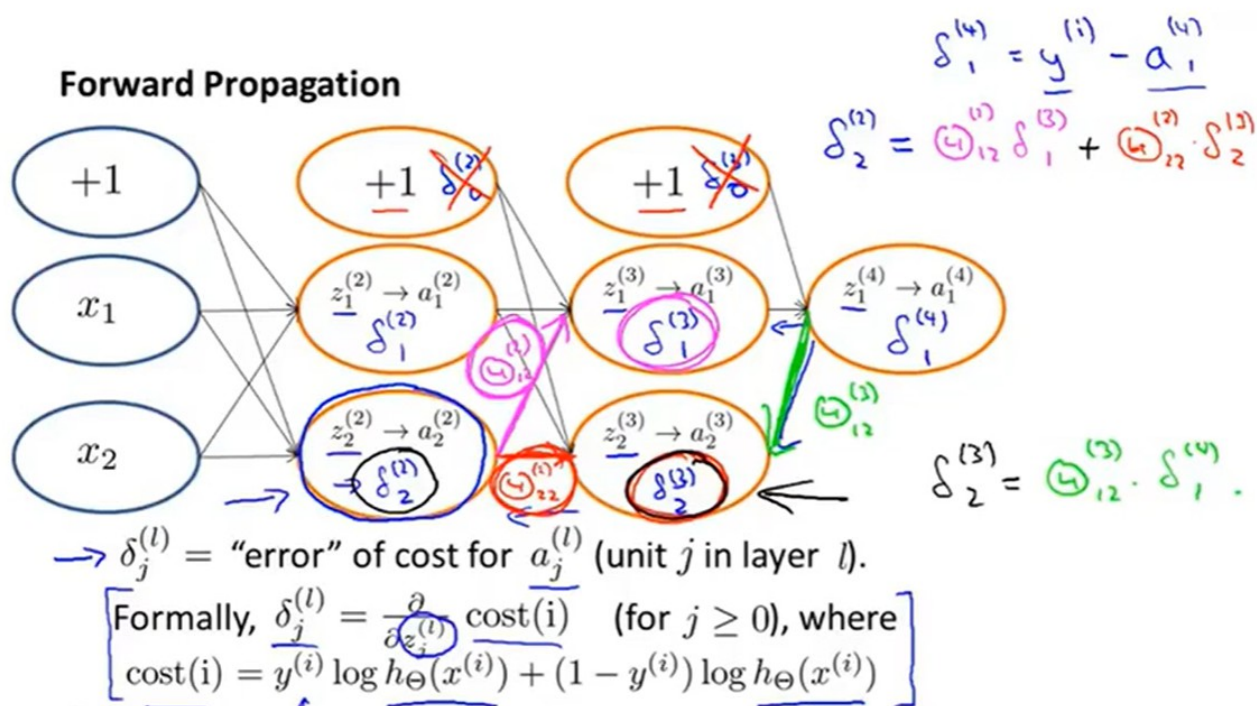
Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

$$\text{cost}(i) = y^{(i)} \log h_{\Theta}(x^{(i)}) + (1 - y^{(i)}) \log h_{\Theta}(x^{(i)})$$

(Think of $\text{cost}(i) \approx (h_{\Theta}(x^{(i)}) - y^{(i)})^2$)

I.e. how well is the network doing on example i ?

再看反向传播算法的实现过程：



2.4 实现注意：展开参数

怎样把我们的参数从矩阵展开成向量，以便我们在高级最优化步骤中的使用需要。

Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
...
optTheta = fminunc(@costFunction, initialTheta, options)
```

Neural Network (L=4):

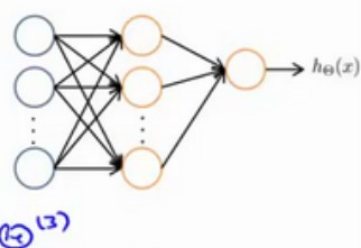
$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (Theta1, Theta2, Theta3)

$D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (D1, D2, D3)

"Unroll" into vectors

Example

$s_1 = 10, s_2 = 10, s_3 = 1$
 $\rightarrow \Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11}$
 $\rightarrow D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}$
 $\rightarrow \text{thetaVec} = [\text{Theta1}(:); \text{Theta2}(:); \text{Theta3}(:)];$
 $\rightarrow \text{DVec} = [\text{D1}(:); \text{D2}(:); \text{D3}(:)];$
 $\text{Theta1} = \text{reshape}(\text{thetaVec}(1:110), 10, 11);$
 $\text{Theta2} = \text{reshape}(\text{thetaVec}(111:220), 10, 11);$
 $\text{Theta3} = \text{reshape}(\text{thetaVec}(221:231), 1, 11);$


Learning Algorithm

- \rightarrow Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
- \rightarrow Unroll to get `initialTheta` to pass to
- $\rightarrow \text{fminunc}(@\text{costFunction}, \text{initialTheta}, \text{options})$

`function [jval, gradientVec] = costFunction(thetaVec)`
 \rightarrow From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ *reshape*
 \rightarrow Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.
 Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

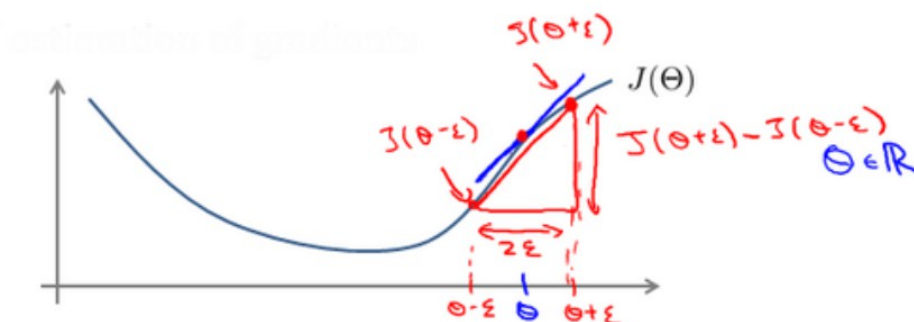
2.5 梯度检验

当我们对一个较为复杂的模型（例如神经网络）使用梯度下降算法时，可能会存在一些不容易察觉的错误，意味着，虽然代价看上去在不断减小，但最终的结果可能并不是最优解。

为了避免这样的问题，我们采取一种叫做梯度的数值检验方法。

这种方法的思想是通过估计梯度值来检验我们计算的导数值是否真的是我们要求的。

对梯度的估计采用的方法是在代价函数上沿着切线的方向选择离两个非常近的点然后计算两个点的平均值用以估计梯度。即对于某个特定的 θ ，我们计算出在 $\theta - \epsilon$ 处和 $\theta + \epsilon$ 的代价值（ ϵ 是一个非常小的值，通常选取 0.001），然后求两个代价的平均，用以估计在 θ 处的代价值。



当 θ 是一个向量时，我们则需要对偏导数进行检验。因为代价函数的偏导数检验只针对一个参数的改变进行检验，下面是一个只针对 θ_1 进行检验的示例：

$$\frac{\partial}{\partial \theta_1} = \frac{J(\theta_1 + \epsilon_1, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon_1, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon}$$

最后我们还需要对通过反向传播方法计算出的偏导数进行检验。

根据上面的算法，计算出的偏导数存储在矩阵 $\Delta_{ij}^{(l)}$ 中。检验时，我们要将该矩阵展开成为向量，同时我们也将 θ 矩阵展开为向量，我们针对每一个 θ 都计算一个近似的梯度值，将这些值存储于一个近似梯度矩阵中，最终将得出的这个矩阵同 $\Delta_{ij}^{(l)}$ 进行比较。

2.6 随机初始化

任何优化算法都需要一些初始的参数。

到目前为止我们都是初始所有参数为 0，这样的初始方法对于逻辑回归来说是可行的，但是对于神经网络来说是不可行的。

如果我们令所有的初始参数都为 0，这将意味着我们第二层的所有激活单元都会有相同的值。同理，如果我们初始所有的参数都为 0 的数，结果也是一样的。

我们通常初始参数为正负之间的随机值，假设我们要随机初始一个尺寸为 10×11 的参数矩阵，代码如下：

```
Theta1 = rand(10, 11) * (2*eps) - eps
```

2.7 神经网络的步骤

网络结构：第一件要做的事是选择网络结构，即决定选择多少层以及决定每层分别有多少个单元。

第一层的单元数即我们训练集的特征数量。最后一层的单元数是我们训练集的结果的类的数量。

如果隐藏层数大于 1，确保每个隐藏层的单元个数相同，通常情况下隐藏层单元的个数越多越好。

我们真正要决定的是隐藏层的层数和每个中间层的单元数。

训练神经网络模型的步骤如下：

1. 参数的随机初始化
2. 利用正向传播方法计算所有的 $h_{\theta}(x)$
3. 编写计算代价函数 J 的代码
4. 利用反向传播方法计算所有偏导数
5. 利用数值检验方法检验这些偏导数
6. 使用优化算法来最小化代价函数