目录

# 第一周：深度学习引言

## 1.1 欢迎

深度学习做的非常好的一个方面就是读取 X 光图像，到生活中的个性化教育，到精准化农业，甚至到驾驶汽车以及其它一些方面。下面是你将学习到的内容：

在 **cousera** 的这一系列也叫做专项课程中，在第一门课中（神经网络和深度学习），你将学习神经网络的基础，你将学习神经网络和深度学习，这门课将持续四周.

在第一门课程中，你将学习如何建立神经网络（包含一个深度神经网络），以及如何在数据上面训练他们。在这门课程的结尾，你将用一个深度神经网络进行辨认猫。

在第二门课中，我们将使用三周时间。你将进行深度学习方面的实践，学习严密地构建神经网络，如何真正让它表现良好，因此你将要学习超参数调整、正则化、诊断偏差和方差以及一些高级优化算法，比如 **Momentum** 和 **Adam** 算法，犹如黑魔法一样根据你建立网络的方式。第二门课只有三周学习时间。

在第三门课中，我们将使用两周时间来学习如何结构化你的机器学习工程。事实证明，构建机器学习系统的策略改变了深度学习的错误。举个例子：你分割数据的方式，分割成训练集、比较集或改变的验证集，以及测试集合，改变了深度学习的错误。

在第四门课程中，我们将会提到卷积神经网络(**CNN(s)**)，它经常被用于图像领域，你将会在第四门课程中学到如何搭建这样的模型。

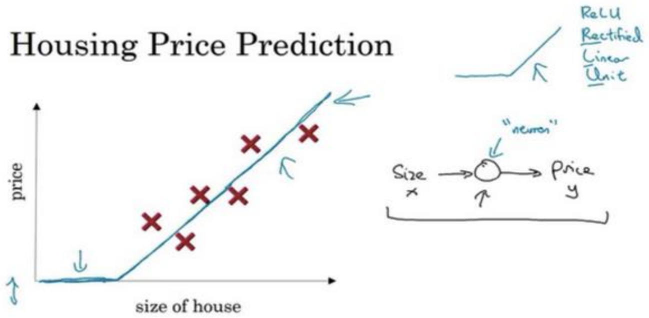
在第五门课中，你将会学习到序列模型，以及如何将它们应用于自然语言处理，以及其它问题。序列模型包括的模型有循环神经网络（**RNN**）、全称是长短期记忆网络（**LSTM**）。你将在课程五中了解其中的时期是什么含义，并且有能力应用到自然语言处理（**NLP**）问题。

吴恩达

# 1.2 什么是神经网络？

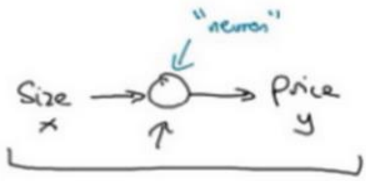
假设你有一个数据集，它包含了六栋房子的信息。你知道房屋的面积是多少平方英尺或者平方米，并且知道房屋价格。这时，你想要拟合一个根据房屋面积预测房价的函数。

如果你对线性回归很熟悉，你可能会说："好吧，让我们用这些数据拟合一条直线。"于是你可能会得到这样一条直线。




作为一个神经网络，这几乎可能是最简单的神经网络。我们把房屋的面积作为神经网络的输入（我们称之为$x$），通过一个节点（一个小圆圈），最终输出了价格（我们用$y$表示）。接着你的网络实现了左边这个函数的功能。

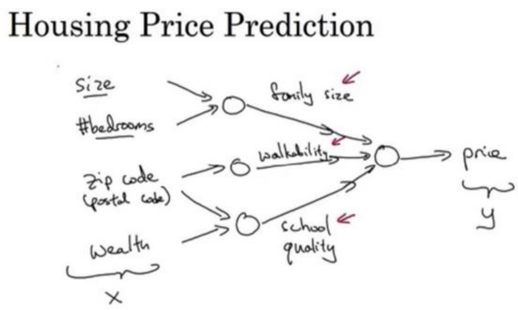
从趋近于零开始，然后变成一条直线。这个函数被称作 **ReLU** 激活函数，它的全称是 **Rectified Linear Unit**。rectify（修正）可以理解成$max(0,x)$，这也是你得到一个这种形状的函数的原因。




如果这是一个单神经元网络，不管规模大小，它正是通过把这些单个神经元叠加在一起来形成。如果你把这些神经元想象成单独的乐高积木，你就通过搭积木来完成一个更大的神经网络。

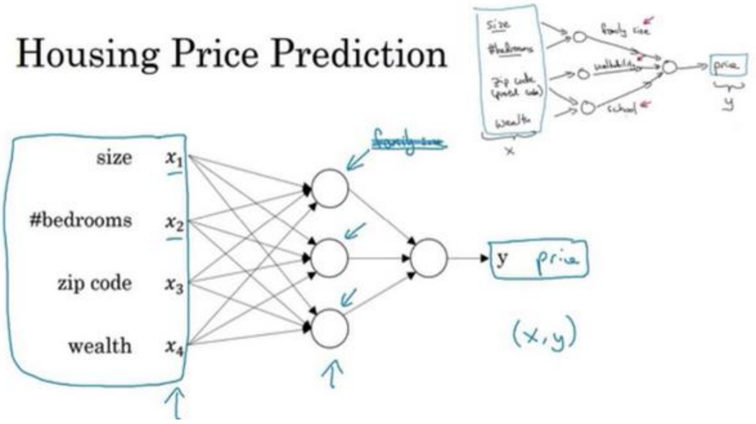
让我们来看一个例子，我们不仅仅用房屋的面积来预测它的价格，现在你有了一些有关房屋的其它特征，比如卧室的数量，或许有一个很重要的因素，一家人的数量也会影响房屋价格，这个房屋能住下一家人或者是四五个人的家庭吗？而这确实是基于房屋大小，以及真正决定一栋房子是否能适合你们家庭人数的卧室数。

换个话题，你可能知道邮政编码或许能作为一个特征，告诉你步行化程度。比如这附近是不是高度步行化，你是否能步行去杂货店或者是学校，以及你是否需要驾驶汽车。




在图上每一个画的小圆圈都可以是 **ReLU** 的一部分，也就是指修正线性单元，或者其它稍微非线性的函数。基于房屋面积和卧室数量，可以估算家庭人口，基于邮编，可以估测步行化程度或者学校的质量。最后你可能会这样想，这些决定人们乐意花费多少钱。

对于一个房子来说，这些都是与它息息相关的事情。在这个情景里，家庭人口、步行化程度以及学校的质量都能帮助你预测房屋的价格。以此为例，$x$ 是所有的这四个输入，$y$ 是你尝试预测的价格，把这些单个的神经元叠加在一起，我们就有了一个稍微大一点的神经网络。这显示了神经网络的神奇之处，虽然我已经描述了一个神经网络，它可以需要你得到房屋面积、步行化程度和学校的质量，或者其它影响价格的因素。



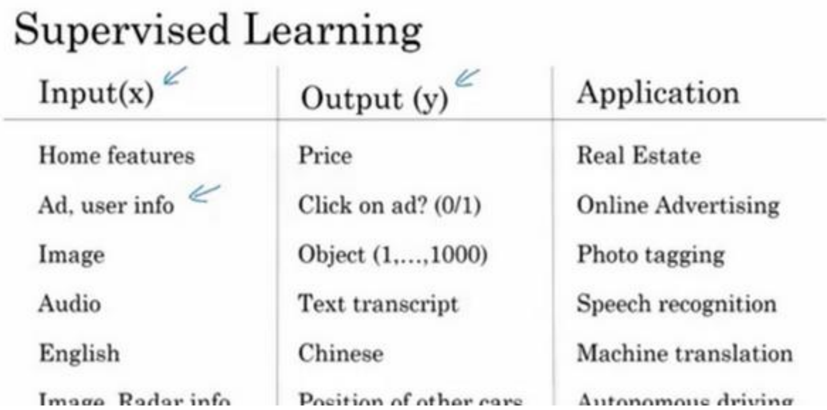

神经网络的一部分神奇之处在于，当你实现它之后，你要做的只是输入$x$，就能得到输出$y$。因为它可以自己计算你训练集中样本的数目以及所有的中间过程。所以，你实际上要做的就是：这里有四个输入的神经网络，这输入的特征可能是房屋的大小、卧室的数量、邮政编码和区域的富裕程度。给出这些输入的特征之后，神经网络的工作就是预测对应的价格。同时也注意到这些被叫做隐藏单元圆圈，在一个神经网络中，它们每个都从输入的四个特征获得自身输入，比如说，第一个结点代表家庭人口，而家庭人口仅仅取决于$x_1$和$x_2$特征。因此，我们说输入层和中间层被紧密的连接起来了。

值得注意的是神经网络给予了足够多的关于$x$和$y$的数据，给予了足够的训练样本有关$x$和$y$。神经网络非常擅长计算从$x$到$y$的精准映射函数。

# 1.3 神经网络的监督学习

关于神经网络也有很多的种类，考虑到它们的使用效果，有些使用起来恰到好处，但事实表明，到目前几乎所有由神经网络创造的经济价值，本质上都离不开一种叫做监督学习的机器学习类别，让我们举例看看。

在监督学习中你有一些输入$x$，你想学习到一个函数来映射到一些输出$y$，比如我们之前提到的房价预测的例子，你只要输入有关房屋的一些特征，试着去输出或者估计价格$y$。我们举一些其它的例子，来说明神经网络已经被高效应用到其它地方。


Supervised Learning

| Input(x) | Output (y) | Application |
|---|---|---|
| Home features | Price | Real Estate |
| Ad, user info | Click on ad? (0/1) | Online Advertising |
| Image | Object (1,…,1000) | Photo tagging |
| Audio | Text transcript | Speech recognition |
| English | Chinese | Machine translation |
| Image, Radar info | Position of other cars | Autonomous driving |

如今应用深度学习获利最多的一个领域，就是在线广告。这也许不是最鼓舞人心的，但真的很赚钱。具体就是通过在网站上输入一个广告的相关信息，因为也输入了用户的信息，于是网站就会考虑是否向你展示广告。

神经网络已经非常擅长预测你是否会点开这个广告，通过向用户展示最有可能点开的广告，这就是神经网络在很多家公司难以置信地提高获利的一种应用。因为有了这种向你展示你最有可能点击的广告的能力，而这一点击的行为的改变会直接影响到一些大型的在线广告公司的收入。

计算机视觉在过去的几年里也取得了长足的进步，这也多亏了深度学习。你可以输入一个图像，然后想输出一个索引，范围从 1 到 1000 来试着告诉你这张照片，它可能是，比方说，1000 个不同的图像中的任何一个，所以你可能会选择用它来给照片打标签。

深度学习最近在语音识别方面的进步也是非常令人兴奋的，你现在可以将音频片段输入神经网络，然后让它输出文本记录。得益于深度学习，机器翻译也有很大的发展。你可以利用神经网络输入英语句子，接着输出一个中文句子。

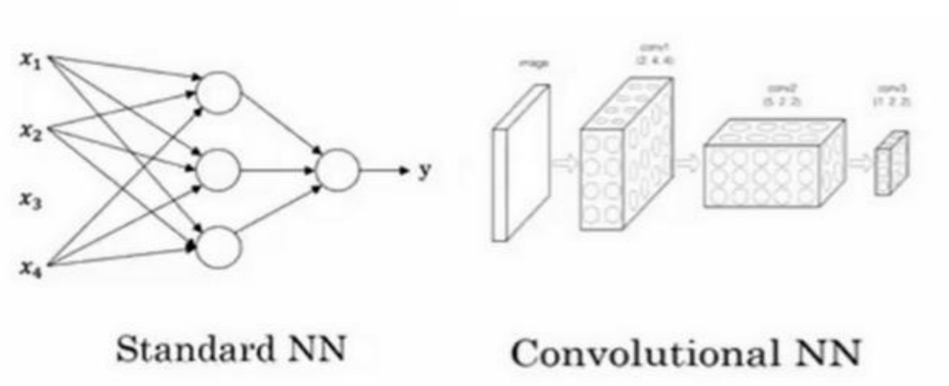
在自动驾驶技术中，你可以输入一幅图像，就好像一个信息雷达展示汽车前方有什么，据此，你可以训练一个神经网络，来告诉汽车在马路上面具体的位置，这就是神经网络在自动驾驶系统中的一个关键成分。

那么深度学习系统已经可以创造如此多的价值，通过智能的选择，哪些作为$x$哪些作为$y$，来针对于你当前的问题，然后拟合监督学习部分，往往是一个更大的系统，比如自动驾

驶。这表明神经网络类型的轻微不同，也可以产生不同的应用，比如说，应用到我们在上一个视频提到的房地产领域，我们不就使用了一个普遍标准神经网络架构吗？

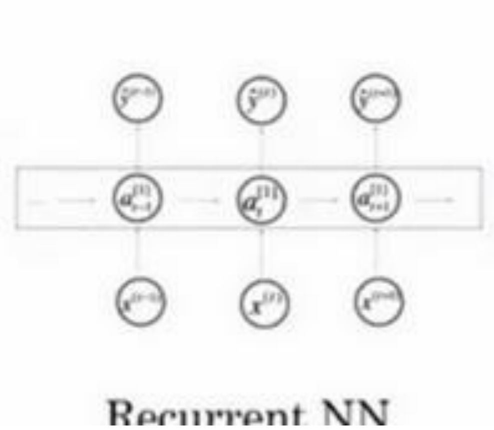
也许对于房地产和在线广告来说可能是相对的标准一些的神经网络，正如我们之前见到的。对于图像应用，我们经常在神经网络上使用卷积（**Convolutional Neural Network**），通常缩写为 **CNN**。对于序列数据，例如音频，有一个时间组件，随着时间的推移，音频被播放出来，所以音频是最自然的表现。作为一维时间序列（两种英文说法 **one-dimensional time series / temporal sequence**）。对于序列数据，经常使用 **RNN**，一种递归神经网络（**Recurrent Neural Network**），语言，英语和汉语字母表或单词都是逐个出现的，所以语言也是最自然的序列数据，因此更复杂的 **RNNs** 版本经常用于这些应用。

对于更复杂的应用比如自动驾驶，你有一张图片，可能会显示更多的 **CNN** 卷积神经网络结构，其中的雷达信息是完全不同的，你可能会有一个更定制的，或者一些更复杂的混合的神经网络结构。所以为了更具体地说明什么是标准的 **CNN** 和 **RNN** 结构，在文献中你可能见过左图这样的图片，这是一个标准的神经网络。而右图是一个卷积神经网络的例子。




我们会在后面的课程了解这幅图的原理和实现，卷积网络(**CNN**)通常用于图像数据。

你可能也会看到这样的图片，而且你将在以后的课程中学习如何实现它。

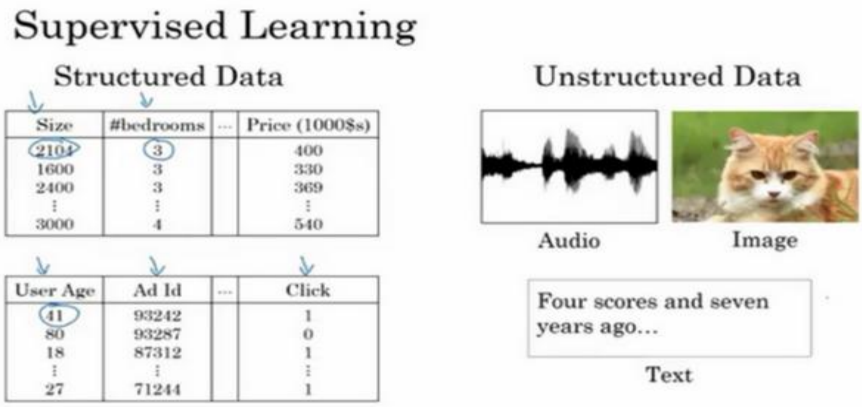



递归神经网络(**RNN**)非常适合这种一维序列，数据可能是一个时间组成部分。

你可能也听说过机器学习对于结构化数据和非结构化数据的应用，结构化数据意味着数据的基本数据库。例如在房价预测中，你可能有一个数据库，有专门的几列数据告诉你卧室的大小和数量，这就是结构化数据。或预测用户是否会点击广告，你可能会得到关于用户的

信息，比如年龄以及关于广告的一些信息，然后对你的预测分类标注，这就是结构化数据，意思是每个特征,比如说房屋大小卧室数量,或者是一个用户的年龄,都有一个很好的定义。

相反非结构化数据是指比如音频，原始音频或者你想要识别的图像或文本中的内容。这里的特征可能是图像中的像素值或文本中的单个单词。



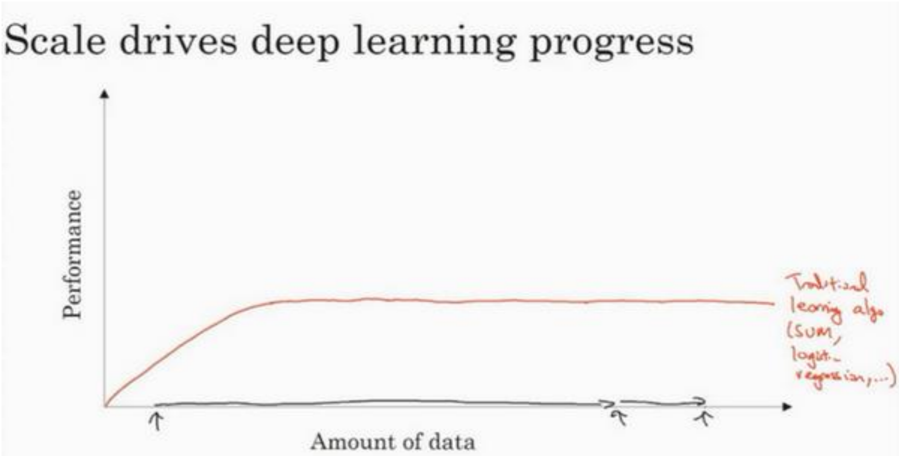

从历史经验上看，处理非结构化数据是很难的，与结构化数据比较，让计算机理解非结构化数据很难，而人类进化得非常善于理解音频信号和图像，文本是一个更近代的发明，但是人们真的很擅长解读非结构化数据。

神经网络的兴起就是这样最令人兴奋的事情之一，多亏了深度学习和神经网络，计算机现在能更好地解释非结构化数据，这是与几年前相比的结果，这为我们创造了机会。许多新的令人兴奋的应用被使用，语音识别、图像识别、自然语言文字处理，甚至可能比两三年前的还要多。因为人们天生就有本领去理解非结构化数据，你可能听说了神经网络更多在媒体非结构化数据的成功，当神经网络识别了一只猫时那真的很酷，我们都知道那意味着什么。

但结果也表明，神经网络在许多短期经济价值的创造，也是基于结构化数据的。比如更好的广告系统、更好的利润建议，还有更好的处理大数据的能力。许多公司不得不根据神经网络做出准确的预测。

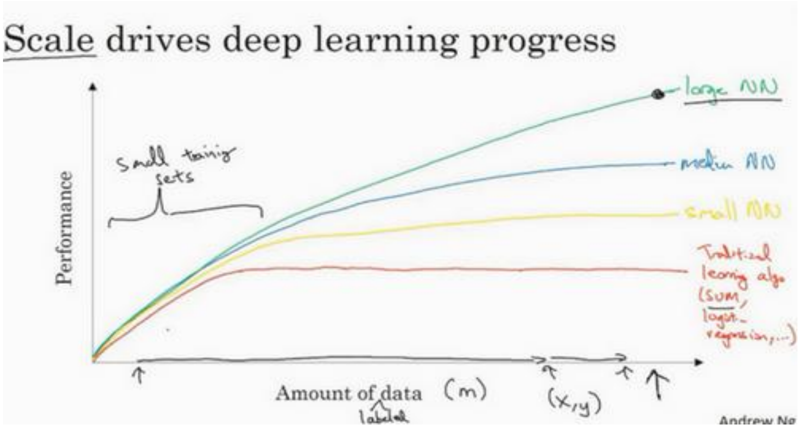
## 1.4 为什么深度学习会兴起？

当我回答这个问题时，我通常给他们画个图，在水平轴上画一个形状，在此绘制出所有任务的数据量，而在垂直轴上，画出机器学习算法的性能，比如说准确率体现在垃圾邮件过滤或者广告点击预测，或者是神经网络在自动驾驶汽车时判断位置的准确性，根据图像可以发现，如果你把一个传统机器学习算法的性能画出来，作为数据量的一个函数，你可能得到一个弯曲的线，就像图中这样，它的性能一开始在增加更多数据时会上升，但是一段变化后它的性能就会像一个平原一样。假设水平轴拉的很长很长，它们不知道如何处理规模巨大的数据，而过去十年的社会里，我们遇到的很多问题只有相对较少的数据量。




多亏数字化社会的来临，现在的数据量都非常巨大，我们花了很多时间活动在这些数字的领域，比如在电脑网站上、在手机软件上以及其它数字化的服务，它们都能创建数据，同时便宜的相机被配置到移动电话，还有加速仪及各类各样的传感器，同时在物联网领域我们也收集到了越来越多的数据。仅仅在过去的 20 年里对于很多应用，我们便收集到了大量的数据，远超过机器学习算法能够高效发挥它们优势的规模。

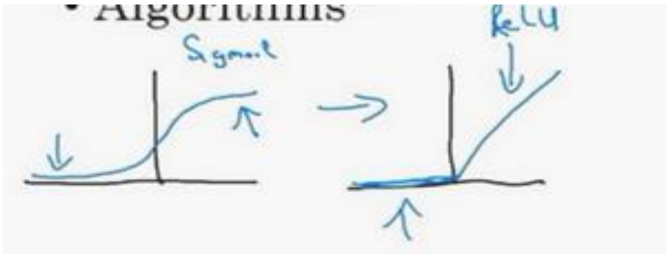
神经网络展现出的是，如果你训练一个小型的神经网络，那么这个性能可能会像下图黄色曲线表示那样；如果你训练一个稍微大一点的神经网络，比如说一个中等规模的神经网络（下图蓝色曲线），它在某些数据上面的性能也会更好一些；如果你训练一个非常大的神经网络，它就会变成下图绿色曲线那样，并且保持变得越来越好。因此可以注意到两点：如果你想要获得较高的性能体现，那么你有两个条件要完成，第一个是你需要训练一个规模足够大的神经网络，以发挥数据规模量巨大的优点，另外你需要能画到$x$轴的这个位置，所以你需要很多的数据。因此我们经常说规模一直在推动深度学习的进步，这里的规模指的也同时是神经网络的规模，我们需要一个带有许多隐藏单元的神经网络，也有许多的参数及关联性，就如同需要大规模的数据一样。为了使这个图更加从技术上讲更精确一点，我在$x$轴下面已经写明的数据量，这儿加上一个标签（label）量，通过添加这个标签量，也就是指在训练样本时，我们同时输入$x$和标签$y$，接下来引入一点符号，使用小写的字母$m$表示训练集的规模，或者说训练样本的数量，这个小写字母$m$就横轴结合其他一些细节到这个图像中。




在这个小的训练集中，各种算法的优先级事实上定义的也不是很明确，所以如果你没有大量的训练集，那效果会取决于你的特征工程能力，那将决定最终的性能。假设有些人训练出了一个 **SVM**（支持向量机）表现的更接近正确特征，然而有些人训练的规模大一些，可能在这个小的训练集中 **SVM** 算法可以做的更好。因此你知道在这个图形区域的左边，各种算法之间的优先级并不是定义的很明确，最终的性能更多的是取决于你在用工程选择特征方面的能力以及算法处理方面的一些细节，只是在某些大数据规模非常庞大的训练集，也就是在右边这个$m$会非常的大时，我们能更加持续地看到更大的由神经网络控制的其它方法。

在深度学习萌芽的初期，数据的规模以及计算量，局限在我们对于训练一个特别大的神经网络的能力，无论是在 **CPU** 还是 **GPU** 上面，那都使得我们取得了巨大的进步。但是渐渐地，尤其是在最近这几年，我们也见证了算法方面的极大创新。许多算法方面的创新，一直是在尝试着使得神经网络运行的更快。

作为一个具体的例子，神经网络方面的一个巨大突破是从 **sigmoid** 函数转换到一个 **ReLU** 函数，这个函数我们在之前的课程里提到过。



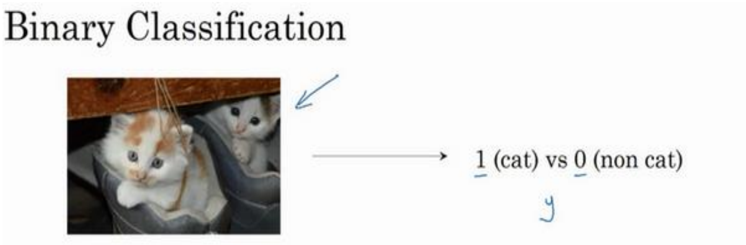
一个使用 **sigmoid** 函数的机器学习问题是，**sigmoid** 函数的梯度会接近零，所以学习的速度会变得非常缓慢，当你实现梯度下降以及梯度接近零的时候，参数会更新的很慢，所以学习的速率也会变的很慢，而通过改变激活函数，**ReLU** 它的梯度对于所有输入的负值都是零，因此梯度更加不会逐渐减少到零。而这里的梯度，这条线的斜率在这左边是零，仅仅通过将 **Sigmod** 函数转换成 **ReLU** 函数，便能够使得一个叫做梯度下降（**gradient descent**）的算法运行的更快，这就是一个相对比较简单的算法创新的例子。

# 第二周：神经网络的编程基础

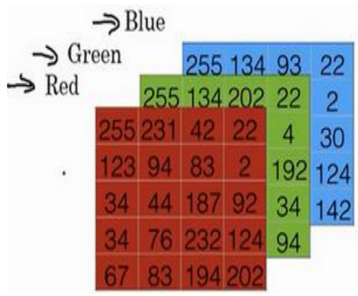
## 2.1 二分类

在神经网络的计算中，通常先有一个叫做前向暂停(**forward pause**)或叫做前向传播(**foward propagation**)的步骤，接着有一个叫做反向暂停(**backward pause**) 或叫做反向传播(**backward propagation**)的步骤。所以这周我也会向你介绍为什么神经网络的训练过程可以分为前向传播和反向传播两个独立的部分。

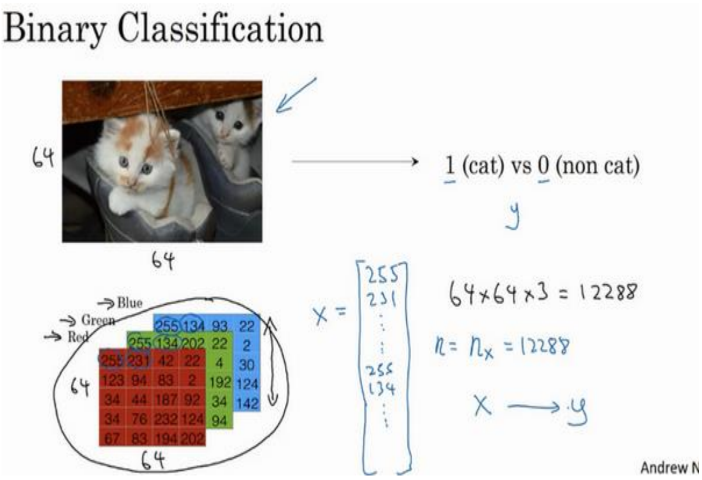
逻辑回归是一个用于二分类(**binary classification**)的算法。首先我们从一个问题开始说起，这里有一个二分类问题的例子，假如你有一张图片作为输入，比如这只猫，如果识别这张图片为猫，则输出标签 1 作为结果；如果识别出不是猫，那么输出标签 0 作为结果。现在我们可以用字母$y$来表示输出的结果标签，如下图所示：




我们来看看一张图片在计算机中是如何表示的，为了保存一张图片，需要保存三个矩阵，它们分别对应图片中的红、绿、蓝三种颜色通道，如果你的图片大小为 64x64 像素，那么你就有三个规模为 64x64 的矩阵，分别对应图片中红、绿、蓝三种像素的强度值。为了便于表示，这里我画了三个很小的矩阵，注意它们的规模为 5x4 而不是 64x64，如下图所示：




为了把这些像素值放到一个特征向量中，我们需要把这些像素值提取出来，然后放入一个特征向量$x$。为了把这些像素值转换为特征向量 $x$，我们需要像下面这样定义一个特征向量 $x$ 来表示这张图片，我们把所有的像素都取出来，直到得到一个特征向量，把图片中所有的红、绿、蓝像素值都列出来。如果图片的大小为 64x64 像素，那么向量 $x$ 的总维度，将是 64 乘以 64 乘以 3，3 是颜色通道。这是三个像素矩阵中像素的总量。在这个例子中结果为 12,288。现在我们用$n_x = 12{,}288$，来表示输入特征向量的维度，有时候为了简洁，我会直接用小写的$n$来表示输入特征向量$x$的维度。所以在二分类问题中，我们的目标就是习得一个分类器，它以图片的特征向量作为输入，然后预测输出结果$y$为 1 还是 0，也就是预测图片中是否有猫：

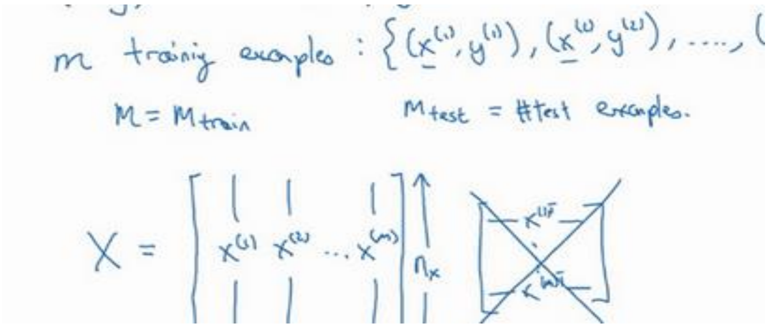



**符号定义 ：**

用一对$(x,y)$来表示一个单独的样本，$x$代表$n_x$维的特征向量，$y$ 表示标签(输出结果)只能为 0 或 1。 而训练集将由$m$个训练样本组成，其中$(x^{(1)},y^{(1)})$表示第一个样本的输入和输出，$(x^{(2)},y^{(2)})$表示第二个样本的输入和输出，直到最后一个样本$(x^{(m)},y^{(m)})$，然后所有的这些一起表示整个训练集。

- $x$：表示一个$n_x$维数据，为输入数据，维度为$(n_x,1)$；
- $y$：表示输出结果，取值为(0,1)；
- $(x^{(i)},y^{(i)})$：第$i$组数据，可能是训练数据，也可能是测试数据，此处默认训练数据；
- $X=[x^{(1)},x^{(2)},...,x^{(m)}]$：表示所有的训练数据集的输入值，放在一个 $n_x\times m$的矩阵中，其中$m$表示样本数目;
- $Y=[y^{(1)},y^{(2)},...,y^{(m)}]$：对应表示所有训练数据集的输出值，维度为$1\times m$。

最后为了能把训练集表示得更紧凑一点，我们会定义一个矩阵用大写$X$的表示，它由输入向量$x^{(1)}$、$x^{(2)}$等组成，如下图放在矩阵的列中，所以现在我们把$x^{(1)}$作为第一列放在矩阵中，$x^{(2)}$作为第二列，$x^{(m)}$放到第$m$列，然后我们就得到了训练集矩阵$X$。所以这个矩阵有$m$列，$m$是训练集的样本数量，然后这个矩的高度记为$n_x$，注意有时候可能因为其他某些原因，矩阵$X$会由训练样本按照行堆叠起来而不是列，如下图所示：$x^{(1)}$的转置直到$x^{(m)}$的转置，但是在实现神经网络的时候，使用左边的这种形式，会让整个实现的过程变得更加简单：

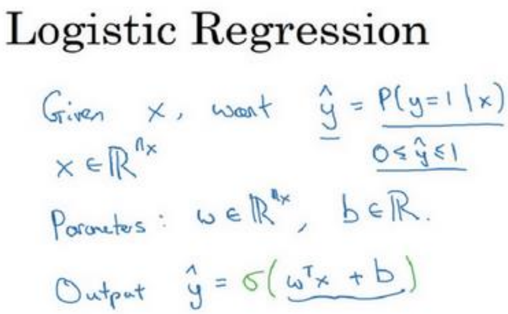


现在来简单温习一下：$X$是一个规模为$n_x$乘以$m$的矩阵，当你用 **Python** 实现的时候，你会看到`X.shape`，这是一条 **Python** 命令，用于显示矩阵的规模，即`X.shape` 等于$(n_x, m)$，$X$是一个规模为$n_x$乘以$m$的矩阵。所以综上所述，这就是如何将训练样本（输入向量$X$的集合）表示为一个矩阵。

那么输出标签$y$呢？同样的道理，为了能更加容易地实现一个神经网络，将标签$y$放在列中将会使得后续计算非常方便，所以我们定义大写的$Y$等于$y^{(1)}, y^{(m)}, \ldots, y^{(m)}$，所以在这里是一个规模为 1 乘以$m$的矩阵，同样地使用 **Python** 将表示为`Y.shape`等于$(1, m)$，表示这是一个规模为 1 乘以$m$的矩阵。

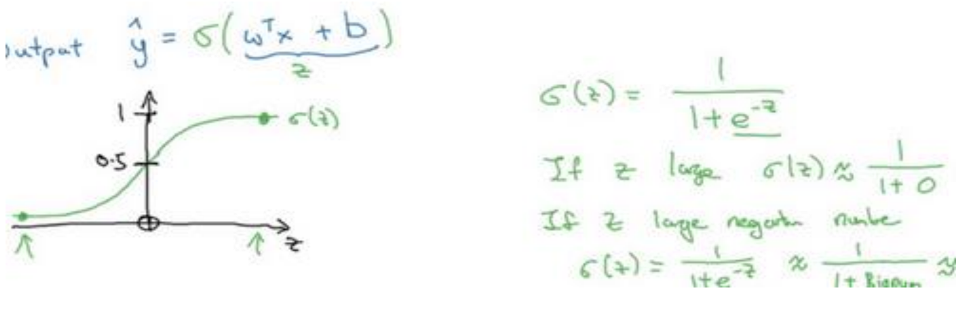
## 2.2 逻辑回归

对于二元分类问题来讲，给定一个输入特征向量$X$，它可能对应一张图片，你想识别这张图片识别看它是否是一只猫或者不是一只猫的图片，你想要一个算法能够输出预测，你能称之为$\hat{y}$，也就是你对实际值 $y$ 的估计。更正式地来说，如果$X$是我们在上个视频看到的图片，你想让 $\hat{y}$ 来告诉你这是一只猫的图片的机率有多大。$X$是一个$n_x$维的向量（相当于有$n_x$个特征的特征向量）。我们用$w$来表示逻辑回归的参数，这也是一个$n_x$维向量（因为$w$实际上是特征权重，维度与特征向量相同），参数里面还有$b$，这是一个实数（表示偏差）。所以给出输入$x$以及参数$w$和$b$之后，我们怎样产生输出预测值$\hat{y}$，一件你可以尝试却不可行的事是让$\hat{y} = w^T x + b$。




因为你想让$\hat{y}$表示实际值$y$等于 1 的机率的话，$\hat{y}$ 应该在 0 到 1 之间。但因为$w^T x + b$可能比 1 要大得多，或者甚至为一个负值。对于你想要的在 0 和 1 之间的概率来说它是没有意义的，因此在逻辑回归中，我们的输出应该是$\hat{y}$等于由上面得到的线性函数式子作为自变量的 **sigmoid** 函数中，公式如上图最下面所示，将线性函数转换为非线性函数。

下图是 **sigmoid** 函数的图像，如果我把水平轴作为$z$轴，那么关于$z$的 **sigmoid** 函数是这样的，它是平滑地从 0 走向 1，让我在这里标记纵轴，这是 0，曲线与纵轴相交的截距是 0.5，这就是关于$z$的 **sigmoid** 函数的图像。我们通常都使用$z$来表示$w^T x + b$的值。




关于 **sigmoid** 函数 $\sigma(z) = \frac{1}{1+e^{-z}}$，在这里$z$是一个实，你的工作就是去让机器学习参数$w$以及$b$这样才使得$\hat{y}$成为对$y = 1$这一情况的概率的一个很好的估计。

## 2.3 逻辑回归的代价函数

为了训练逻辑回归模型的参数$w$和参数$b$，我们需要一个代价函数，通过训练代价函数来得到参数$w$和参数$b$。先看一下逻辑回归的输出函数：

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b), \text{where } \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\text{Given } \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}, \text{ want } \hat{y}^{(i)} \approx y^{(i)}.$$

对训练集的预测值 $\hat{y}$，我们更希望它会接近于训练集中的$y$值，训练样本$i$所对应的预测值是$y^{(i)}$,是用训练样本的$w^T x^{(i)} + b$，然后通过 **sigmoid** 函数来得到，也可以把$z$定义为$z^{(i)} = w^T x^{(i)} + b$，上标$(i)$来指明数据表示$x$或者$y$或者$z$或者其他数据的第$i$个训练样本。

**损失函数：**$L(\hat{y}, y)$.

我们通过损失函数$L(\hat{y}, y)$，来衡量预测输出值和实际值有多接近。一般我们用预测值和实际值的平方差或者它们平方差的一半，但是通常在逻辑回归中我们不这么做，因为当我们在学习逻辑回归参数的时候，会发现我们的优化目标不是凸优化，只能找到多个局部最优值，梯度下降法很可能找不到全局最优值，虽然平方差是一个不错的损失函数，但是我们在逻辑回归模型中会定义另外一个损失函数。$L(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$

为什么要用这个函数作为逻辑损失函数？我们想让它尽可能地小，为了更好地理解这个损失函数怎么起作用，我们举两个例子：

- 当$y = 1$时损失函数$L = -\log(\hat{y})$，如果想要损失函数$L$尽可能得小，那么$\hat{y}$就要尽可能大，因为 **sigmoid** 函数取值[0,1]，所以$\hat{y}$会无限接近于 1。
- 当$y = 0$时损失函数$L = -\log(1-\hat{y})$，如果想要损失函数$L$尽可能得小，那么$\hat{y}$就要尽可能小，因为 **sigmoid** 函数取值[0,1]，所以$\hat{y}$会无限接近于 0。

为了衡量算法在全部训练样本上的表现如何，我们需要定义一个算法的代价函数，算法的代价函数是对$m$个样本的损失函数求和然后除以$m$:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}\log\hat{y}^{(i)} - (1-y^{(i)})\log(1-\hat{y}^{(i)})\right)$$

损失函数只适用于像这样的单个训练样本，而代价函数是参数的总代价，所以在训练逻辑回归模型时候，我们需要找到合适的$w$和$b$，来让代价函数 $J$ 的总代价降到最低。

## 2.4 梯度下降法

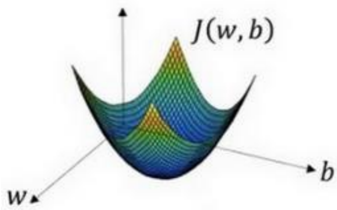
梯度下降法可以做什么？在你测试集上，通过最小化代价函数（成本函数）$J(w,b)$来训练的参数$w$和$b$，

Gradient Descent

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})$$

在这个图中，横轴表示你的空间参数$w$和$b$，在实践中，$w$可以是更高的维度，但是为了更好地绘图，我们定义$w$和$b$，都是单一实数，代价函数（成本函数）$J(w,b)$是在水平轴$w$和$b$上的曲面，因此曲面的高度就是$J(w,b)$在某一点的函数值。

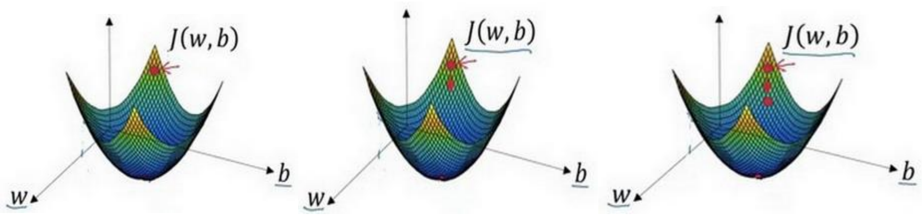



我们所做的就是找到使得代价函数$J(w,b)$函数值是最小值，对应的参数$w$和$b$。如图，代价函数（成本函数）$J(w,b)$是一个凸函数(convex function)，像一个大碗一样。

如图，这就与刚才的图有些相反，因为它是非凸的并且有很多不同的局部最小值。

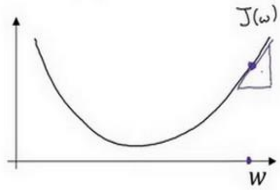
由于逻辑回归的代价函数（成本函数）$J(w,b)$特性，我们必须定义代价函数（成本函数）$J(w,b)$为凸函数。初始化$w$和$b$，可以用如图那个小红点来初始化参数$w$和$b$，也可以采用随机初始化的方法，对于逻辑回归几乎所有的初始化方法都有效，因为函数是凸函数，无论在哪里初始化，应该达到同一点或大致相同的点。

1. 我们以如图的小红点的坐标来初始化参数$w$和$b$。
2. 朝最陡的下坡方向走一步，不断地迭代
3. 直到走到全局最优解或者接近全局最优解的地方



通过以上的三个步骤我们可以找到全局最优解，也就是代价函数（成本函数）$J(w,b)$这个凸函数的最小值点。

假定代价函数（成本函数）$J(w)$ 只有一个参数$w$，即用一维曲线代替多维曲线，这样可以更好画出图像。

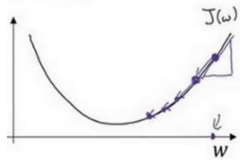



迭代就是不断重复做如图的公式:

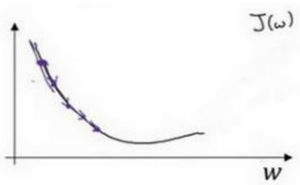
$$w := w - a\frac{dJ(w)}{dw}$$

$a$ 表示学习率（**learning rate**），用来控制步长（**step**），即向下走一步的长度，$\frac{dJ(w)}{dw}$就是函数$J(w)$对$w$ 求导（**derivative**），在代码中我们会使用$dw$表示这个结果。

对于导数更加形象化的理解就是斜率（**slope**），如图该点的导数就是这个点相切于 $J(w)$的小三角形的高除宽。假设我们以如图点为初始化点，该点处的斜率的符号是正的，即$\frac{dJ(w)}{dw} > 0$，所以接下来会向左走一步。整个梯度下降法的迭代过程就是不断地向左走，直至逼近最小值点。




假设我们以如图点为初始化点，该点处的斜率的符号是负的，即$\frac{dJ(w)}{dw} < 0$，所以接下来会向右走一步。



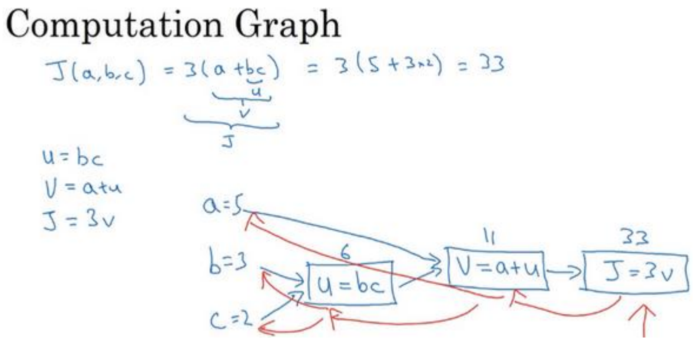

梯度下降法的细节化说明（两个参数），逻辑回归的代价函数$J(w,b)$是含有两个参数的。

$$w := w - a\frac{\partial J(w,b)}{\partial w} \qquad b := b - a\frac{\partial J(w,b)}{\partial b}$$

$\partial$ 表示求偏导符号，可以读作 **round**， $\frac{\partial J(w,b)}{\partial w}$ 就是函数$J(w,b)$ 对$w$ 求偏导，在代码中我们会使用$dw$ 表示这个结果， $\frac{\partial J(w,b)}{\partial b}$ 就是函数$J(w,b)$对$b$ 求偏导，在代码中我们会使用$db$ 表示这个结果， 小写字母$d$ 用在求导数（**derivative**），即函数只有一个参数， 偏导数符号$\partial$ 用在求偏导（**partial derivative**），即函数含有两个以上的参数。

# 2.5 计算图

我们将举一个例子说明计算图是什么。让我们举一个比逻辑回归更加简单的，或者说不那么正式的神经网络的例子。




我们尝试计算函数$J$，$J$是由三个变量$a,b,c$组成的函数，这个函数是$3(\mathrm{a}+\mathrm{bc})$ 。计算这个函数实际上有三个不同的步骤，首先是计算 $b$ 乘以 $c$，我们把它储存在变量$u$中，因此$u=bc$； 然后计算$v=a+u$；最后输出$J=3v$，这就是要计算的函数$J$。我们可以把这三步画成如下的计算图，我先在这画三个变量$a,b,c$，第一步就是计算$u=bc$，我在这周围放个矩形框，它的输入是$b,c$，接着第二步$v=a+u$，最后一步$J=3v$。

举个例子: $a=5,b=3,c=2$，$u=bc$就是 6，v 就是 5+6=11。$J$是 3 倍的 v，即$3\times(5+3\times2)$。如果你把它算出来，实际上得到 33 就是$J$的值。 当有不同的或者一些特殊的输出变量时，例如本例中的$J$和逻辑回归中你想优化的代价函数$J$，因此计算图用来处理这些计算会很方便。从这个小例子中我们可以看出，通过一个从左向右的过程，你可以计算出$J$的值。为了计算导数，从右到左（红色箭头，和蓝色箭头的过程相反）的过程是用于计算导数最自然的方式。

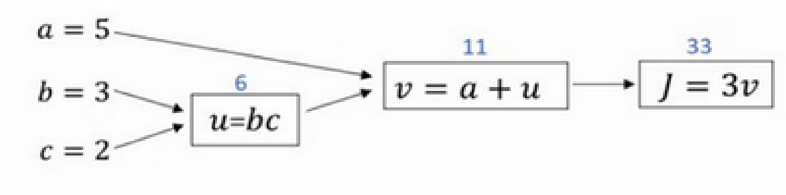
概括一下：计算图组织计算的形式是用蓝色箭头从左到右的计算，让我们看看下一个视频中如何进行反向红色箭头(也就是从右到左)的导数计算，让我们继续下一个视频的学习。

## 2.6 使用计算图求导数

在上一个视频中，我们看了一个例子使用流程计算图来计算函数$J$。现在我们看看流程图的描述，看看如何利用它计算出函数$J$的导数。下面用到的公式：

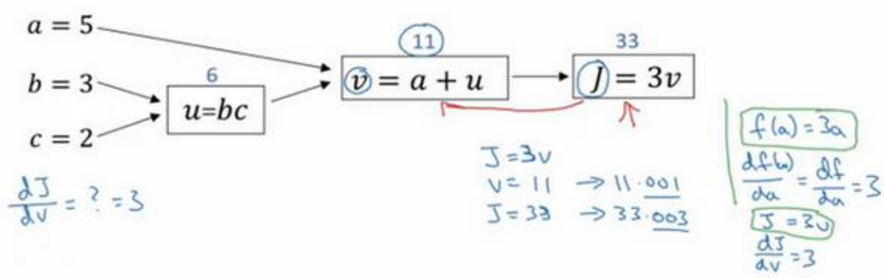
$$\frac{dJ}{du}=\frac{dJ}{dv}\frac{dv}{du}\ ,\quad \frac{dJ}{db}=\frac{dJ}{du}\frac{du}{db}\ ,\quad \frac{dJ}{da}=\frac{dJ}{du}\frac{du}{da}$$

这是一个流程图：



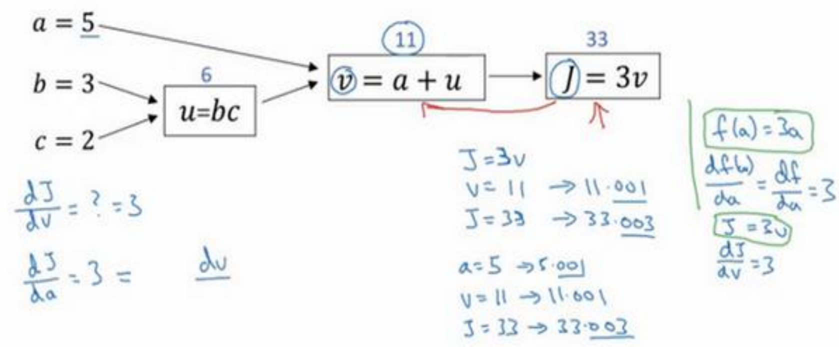

假设你要计算$\frac{dJ}{dv}$，那要怎么算呢？定义$J=3v$，现在$v=11$，如果让$v$增加一点点，比如到 11.001，那么$J=3v=33.003$，所以我这里$v$增加了 0.001，然后最终结果是$J$上升到原来的 3 倍，所以$\frac{dJ}{dv}=3$，因为对于任何 $v$ 的增量$J$都会有 3 倍增量。

在反向传播算法中的术语，如果你想计算最后输出变量的导数，使用你最关心的变量对$v$的导数，那么我们就做完了一步反向传播，在这个流程图中是一个反向步。



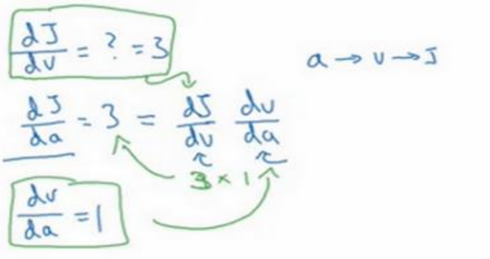

我们来看另一个例子，$\frac{dJ}{da}$是多少呢？如果我们提高$a$的数值，对$J$的数值有什么影响？

变量$a=5$，我们让它增加到 5.001，那么对$v$的影响就是$a+u$，之前$v=11$，现在变成 11.001，现在$J$就变成 33.003 了，所以$J$的增量是 3 乘以$a$的增量，意味着这个导数是 3。



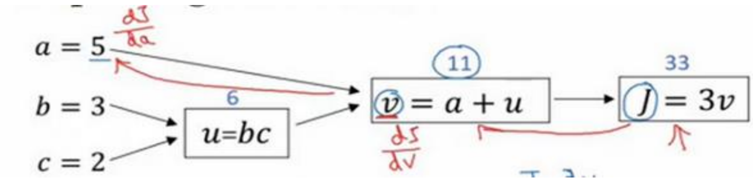

首先$a$增加了，$v$也会增加，$v$增加多少呢？这取决于$\frac{dv}{da}$，然后$v$的变化导致$J$也在增加，

这在微积分里叫链式法则（chain rule），如果$a$影响到$v$，$v$影响到$J$，那么当你让$a$变大时，$J$的变化量就是当你改变$a$时，$v$的变化量乘以改变$v$时$J$的变化量，在微积分里这叫链式法则。

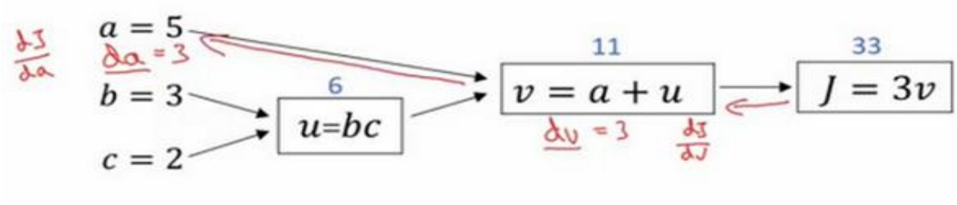



我们从这个计算中看到，如果你让$a$增加 0.001，$v$也会变化相同的大小，所以$\frac{dv}{da} = 1$。事实上，如果你代入进去，我们之前算过$\frac{dJ}{dv} = 3$，$\frac{dv}{da} = 1$，所以这个乘积 3×1，实际上就给出了正确答案，$\frac{dJ}{da} = 3$。

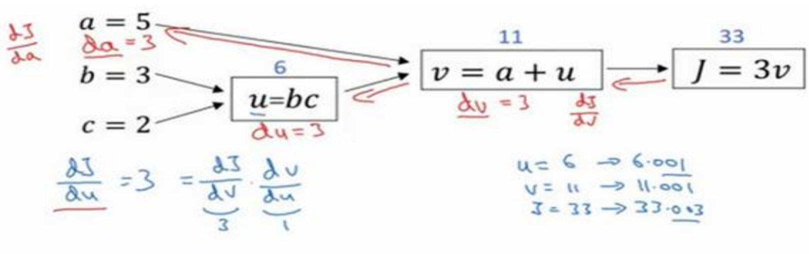
这张图表示了如何计算，$\frac{dJ}{dv}$就是$J$对变量$v$的导数，它可以帮助你计算$\frac{dJ}{da}$，所以这是另一步反向传播计算。




我们清理出一张新的流程图，我们回顾一下，到目前为止，我们一直在往回传播，并计算$dv = 3$，再次，$dv$是代码里的变量名，其真正的定义是$\frac{dJ}{dv}$。我发现$da = 3$，再次，$da$是代码里的变量名，其实代表$\frac{dJ}{da}$的值。




大概手算了一下，两条直线怎么计算反向传播。$\frac{dJ}{du}$是多少呢？通过和之前类似的计算，现在我们从$u = 6$出发，如果你令$u$增加到 6.001，那么$v$之前是 11，现在变成 11.001 了，$J$就从 33 变成 33.003，所以$J$ 增量是 3 倍，所以$\frac{\mathrm{dJ}}{du} = 3$。对$u$的分析很类似对 a 的分析，实际上这计算起来就是$\frac{dJ}{dv} \cdot \frac{dv}{du}$，有了这个，我们可以算出$\frac{dJ}{dv} = 3$，$\frac{dv}{du} = 1$，最终结果是$3 \times 1 = 3$。




现在，我们仔细看看最后一个例子，那么$\frac{dJ}{db}$呢？想象一下，如果你改变了$b$的值，你想要然后变化一点，让$J$值到达最大或最小，那么导数是什么呢？这个$J$函数的斜率，当你稍微改变$b$值之后。事实上，使用微积分链式法则，这可以写成两者的乘积，就是$\frac{dJ}{du}\cdot\frac{du}{db}$，理由是，如果你改变$b$一点点，所以$b$变化比如说 3.001，它影响 J 的方式是，首先会影响$u$，它对$u$的影响有多大？好，$u$的定义是$b \cdot c$，所以$b = 3$时这是 6，现在就变成 6.002 了，对吧，因为在我们的例子中$c = 2$，所以这告诉我们$\frac{du}{db} = 2$当你让$b$增加 0.001 时，$u$就增加两倍。所以$\frac{du}{db} = 2$，现在我想$u$的增加量已经是$b$的两倍，那么$\frac{dJ}{du}$是多少呢？我们已经弄清楚了，这等于 3，所以让这两部分相乘，我们发现$\frac{dJ}{db} = 6$。

## 2.7 逻辑回归中的梯度下降

假设样本只有两个特征$x_1$和$x_2$，为了计算$z$，我们需要输入参数$w_1$、$w_2$ 和$b$，除此之外还有特征值$x_1$和$x_2$。因此$z$的计算公式为： $z = w_1x_1 + w_2x_2 + b$

逻辑回归的公式定义如下： $\hat{y} = a = \sigma(z)$ 其中$z = w^Tx + b$ ， $\sigma(z) = \frac{1}{1+e^{-z}}$

损失函数： $L(\hat{y}^{(i)}, y^{(i)}) = -y^{(i)}\log\hat{y}^{(i)} - (1 - y^{(i)})\log(1 - \hat{y}^{(i)})$

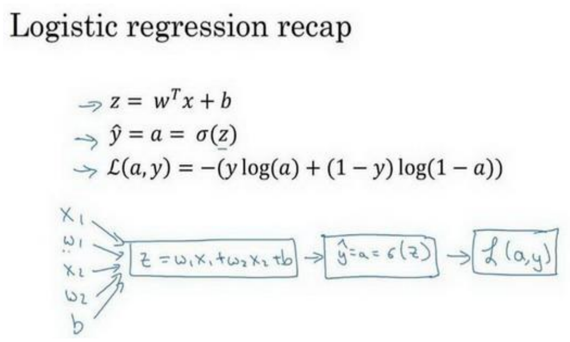
代价函数： $J(w, b) = \frac{1}{m}\sum_i^m L(\hat{y}^{(i)}, y^{(i)})$

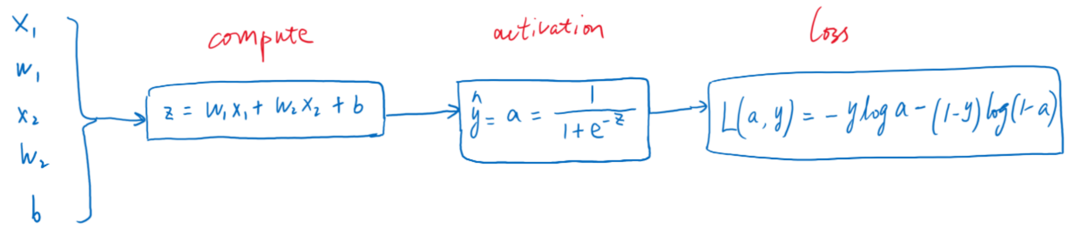
假设只考虑单个样本的情况，单个样本的代价函数定义如下：

$$L(a, y) = -y\log(a) + (1 - y)\log(1 - a))$$

其中$a$是逻辑回归的输出，$y$是样本的标签值。现在让我们画出表示这个计算的计算图。这里先复习下梯度下降法，$w$和$b$的修正量可以表达如下：

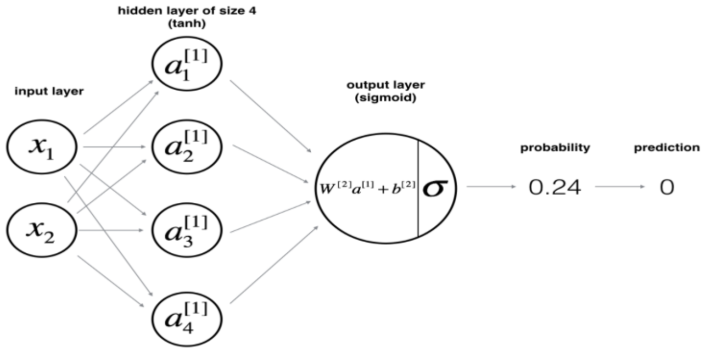
$$w := w - a\frac{\partial J(w,b)}{\partial w}，\ b := b - a\frac{\partial J(w,b)}{\partial b}$$




$$\frac{\partial L(a,y)}{w_1} = \frac{\partial L}{\partial a}\cdot\frac{\partial a}{\partial z}\cdot\frac{\partial z}{\partial w_1} = \underbrace{\left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\cdot\left[a(1-a)\right]}_{(a-y)}\cdot x_1 = (a-y)\cdot x_1$$

$$\frac{\partial L(a,y)}{\partial b} = \frac{\partial L}{\partial a}\cdot\frac{\partial a}{\partial z}\cdot\frac{\partial z}{\partial b} = \underbrace{\left(-\frac{y}{a} + \frac{1-y}{1-a}\right)\cdot\left[a(1-a)\right]}_{(a-y)}\cdot 1 = (a-y)$$




**Mathematically**:

For one example $x^{(i)}$:

$$
\begin{aligned}
z^{[1](i)} &= W^{[1]}x^{(i)} + b^{[1]}\\
a^{[1](i)} &= \tanh(z^{[1](i)})\\
z^{[2](i)} &= W^{[2]}a^{[1](i)} + b^{[2]}\\
\hat{y}^{(i)} &= a^{[2](i)} = \sigma(z^{[2](i)})\\
y^{(i)}_{prediction} &= \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

现在进行最后一步反向推导，也就是计算$w$和$b$变化对代价函数$L$的影响，特别地，可以用:

$$dw_1 = \frac{1}{m}\sum_i^m x_1^{(i)}\,(a^{(i)} - y^{(i)})$$

$$dw_2 = \frac{1}{m}\sum_i^m x_2^{(i)}\,(a^{(i)} - y^{(i)})$$

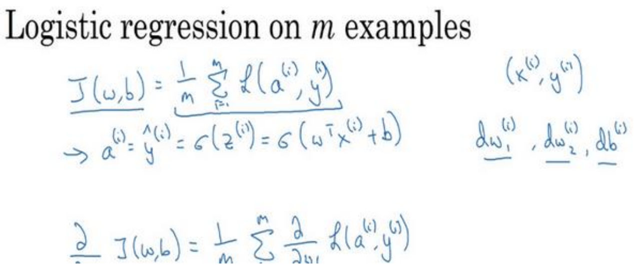
$$db = \frac{1}{m}\sum_i^m (a^{(i)} - y^{(i)})$$

视频中， $dw_1$ 表示$\frac{\partial L}{\partial w_1} = x_1 \cdot dz$， $dw_2$ 表示$\frac{\partial L}{\partial w_2} = x_2 \cdot dz$， $db = dz$。

然后：更新$w_1 = w_1 - a dw_1$， 更新$w_2 = w_2 - a dw_2$， 更新$b = b - \alpha db$。

# 2.8 m 个样本的梯度下降

在之前的视频中，你已经看到如何计算导数，以及应用梯度下降在逻辑回归的一个训练样本上。现在我们想要把它应用在$m$个训练样本上。

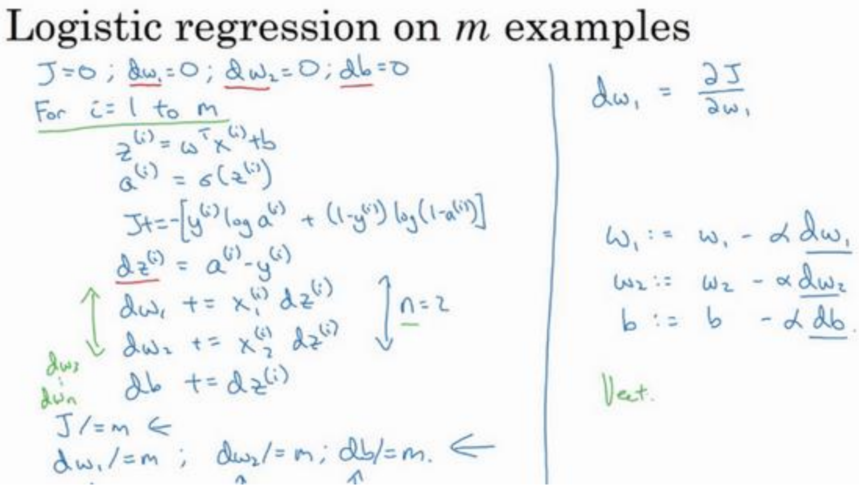



首先，让我们时刻记住有关于损失函数$J(w,b)$的定义。

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$$

当你的算法输出关于样本$y$的$a^{(i)}$，$a^{(i)}$是训练样本的预测值，即：$\sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$。 前面的是对于任意单个训练样本，如何计算微分当你只有一个训练样本。因此$dw_1$，$dw_2$和$db$ 添上上标$i$表示你求得的相应的值。

现在你知道带有求和的全局代价函数，实际上是 1 到$m$项各个损失的平均。 所以它表明全局代价函数对$w_1$的微分，对$w_1$的微分也同样是各项损失对$w_1$微分的平均。




但之前我们已经演示了如何计算这项，即之前幻灯中演示的如何对单个训练样本进行计算。所以你真正需要做的是计算这些微分，如我们在之前的训练样本上做的。并且求平均，这会给你全局梯度值，你能够把它直接应用到梯度下降算法中。

当你应用深度学习算法，你会发现在代码中显式地使用 **for** 循环使你的算法很低效，同时在深度学习领域会有越来越大的数据集。所以能够应用你的算法且没有显式的 **for** 循环会是重要的，并且会帮助你适用于更大的数据集。所以这里有一些叫做向量化技术，它可以允许你的代码摆脱这些显式的 **for** 循环。

## 2.9 向量化

向量化是非常基础的去除代码中 **for** 循环的艺术，因为深度学习算法处理大数据集效果很棒，所以你的代码运行速度非常重要，否则如果在大数据集上，你的代码可能花费很长时间去运行。

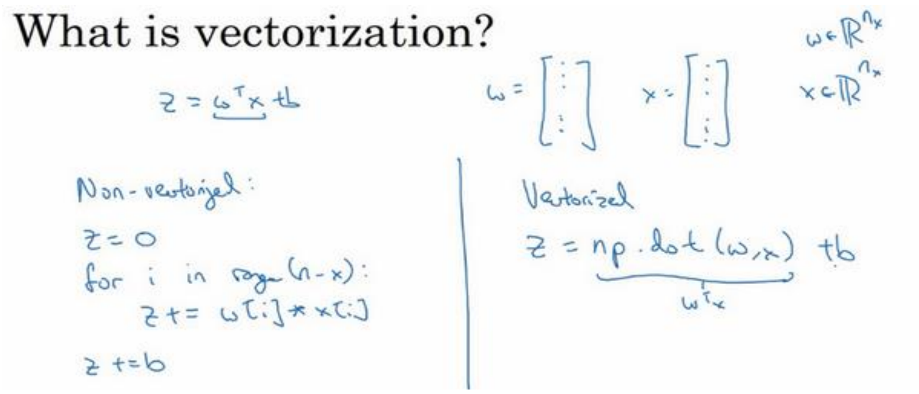
在逻辑回归中你需要去计算$z = w^T x + b$，$w$、$x$都是列向量。如果你有很多的特征那么就会有一个非常大的向量，所以$w \in \mathbb{R}^{n_x}$ ，$x \in \mathbb{R}^{n_x}$，所以如果你想使用非向量化方法去计算$w^T x$，你需要用如下方式（**python**）

```
z = 0
for i in range(n_x)
   z += w[i]*x[i]
z += b
```

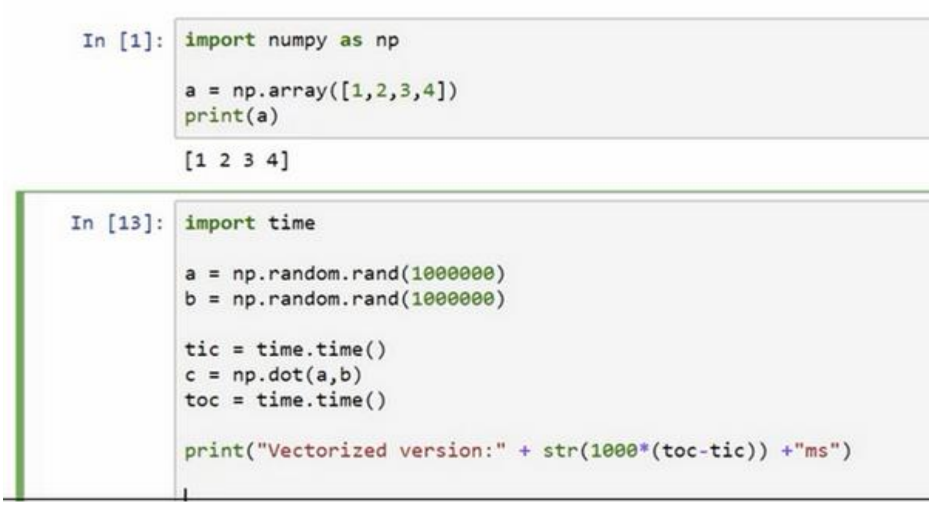
这是一个非向量化的实现，你会发现这真的很慢，作为一个对比，向量化实现将会非常直接计算$w^T x$，代码如下：

```
z = np.dot(w.T, x) + b
```

这是向量化计算$w^T x$的方法，你将会发现这个非常快




让我们用一个小例子说明一下:



```
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) +"ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")
```

```
250286.989866
Vectorized version:1.5027523040771484ms
250286.989866
For loop:474.29513931274414ms
```

向量化和非向量化计算了相同的值，向量化版本花费了 1.5 毫秒，非向量化版本的 **for** 循环花费了大约几乎 500 毫秒，非向量化版本多花费了 300 倍时间。所以仅仅是向量化你的代码，就会运行 300 倍快。这意味着如果向量化方法需要花费一分钟去运行的数据，**for** 循环将会花费 5 个小时去运行。一句话总结，以上都是再说和 **for** 循环相比，向量化可以快速得到结果。

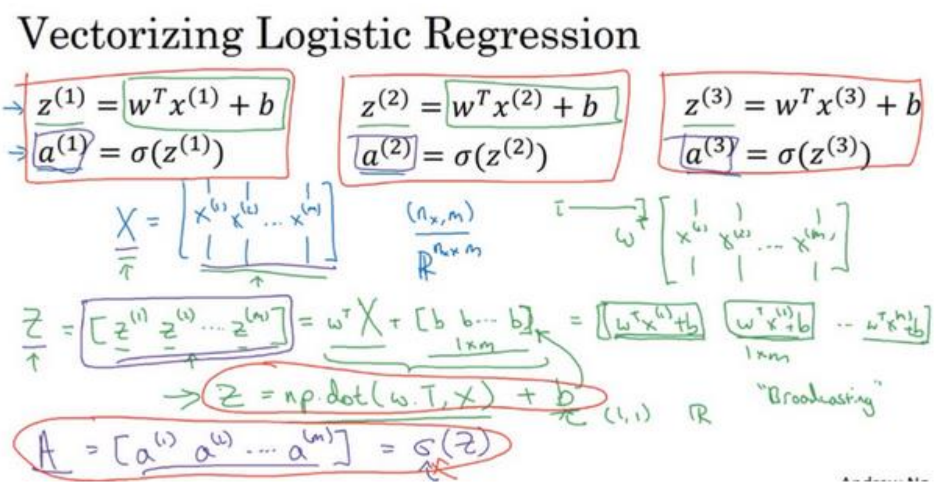
# 2.10 向量化逻辑回归

首先我们回顾一下逻辑回归的前向传播步骤。如果你有 $m$ 个训练样本，然后对第一个样本进行预测，你需要这样计算。计算 $z$，我正在使用这个熟悉的公式 $z^{(1)} = w^T x^{(1)} + b$ 。然后计算激活函数 $a^{(1)} = \sigma(z^{(1)})$ ，计算第一个样本的预测值 $y$ 。

对第二个样本进行预测，你需要计算 $z^{(2)} = w^T x^{(2)} + b$ ， $a^{(2)} = \sigma(z^{(2)})$ 。

对第三个样本进行预测，你需要计算 $z^{(3)} = w^T x^{(3)} + b$ ， $a^{(3)} = \sigma(z^{(3)})$ ，依次类推。如果你有 $m$ 个训练样本，你可能需要这样做 $m$ 次，可以看出，为了完成前向传播步骤，即对我们的 $m$ 个样本都计算出预测值。有一个办法可以并且不需要任何一个明确的 **for** 循环。让我们来看一下你该怎样做。

首先，回忆一下我们曾经定义了一个矩阵 $X$ 作为你的训练输入，(如下图中蓝色 $X$ )像这样在不同的列中堆积在一起。这是一个 $n_x$ 行 $m$ 列的矩阵。我现在将它写为 **Python numpy** 的形式 $(n_x, m)$ ，这只是表示 $X$ 是一个 $n_x$ 乘以 $m$ 的矩阵 $R^{n_x \times m}$。




总结一下，在这张幻灯片中我们已经看到，不需要 **for** 循环，利用 $m$ 个训练样本一次性计算出小写 $z$ 和小写 $a$，用一行代码即可完成。

```
Z = np.dot(w.T,X) + b
```

这一行代码：$A = [a^{(1)} a^{(2)} ... a^{(m)}] = \sigma(Z)$ ，通过恰当地运用$\sigma$一次性计算所有 $a$。这就是在同一时间内你如何完成一个所有 $m$ 个训练样本的前向传播向量化计算。

概括一下，你刚刚看到如何利用向量化在同一时间内高效地计算所有的激活函数的所有$a$值。接下来，可以证明，你也可以利用向量化高效地计算反向传播并以此来计算梯度。让我们在下一个视频中看该如何实现。

## 2.11 向量化 logistic 回归的梯度输出

如何向量化计算的同时，对整个训练集预测结果$a$，这是我们之前已经讨论过的内容。在本次视频中我们将学习如何向量化地计算$m$个训练数据的梯度，本次视频的重点是如何**同时**计算 $m$ 个数据的梯度，并且实现一个非常高效的逻辑回归算法**(Logistic Regression)**。

之前我们在讲梯度计算的时候，列举过几个例子， $dz^{(1)}=a^{(1)}-y^{(1)}$，$dz^{(2)}=a^{(2)}-y^{(2)}$ ……等等一系列类似公式。现在，对 $m$个训练数据做同样的运算，我们可以定义一个新的变量 $dZ=[dz^{(1)},dz^{(2)}...dz^{(m)}]$ ，所有的 $dz$ 变量横向排列，因此，$dZ$ 是一个 $1\times m$ 的矩阵，或者说，一个 $m$ 维行向量。在之前的幻灯片中，我们已经知道如何计算$A$，即$[a^{(1)},a^{(2)}...a^{(m)}]$，我们需要找到这样的一个行向量 $Y=[y^{(1)}\ y^{(2)}\ ...\ y^{(m)}]$ ，由此，我们可以这样计算 $dZ=A-Y=[a^{(1)}-y^{(1)}\ a^{(2)}-y^{(2)}\ ...\ a^{(m)}-y^{(m)}]$，不难发现第一个元素就是 $dz^{(1)}$，第二个元素就是 $dz^{(2)}$ ……所以我们现在仅需一行代码，就可以同时完成这所有的计算。

在之前的实现中，我们已经去掉了一个 **for** 循环，但我们仍有一个遍历训练集的循环，如下所示：

$$dw=0$$

$$dw+=x^{(1)}*dz^{(1)}$$

$$dw+=x^{(2)}\ *dz^{(2)}$$

…………

$$dw+=x^{(m)}*dz^{(m)}$$

$$dw=\frac{dw}{m}$$

$$db=0$$

$$db+=dz^{(1)}$$

$$db+=dz^{(2)}$$

…………

$$db+=dz^{(m)}$$

$$db=\frac{db}{m}$$

上述（伪）代码就是我们在之前实现中做的，我们已经去掉了一个 **for** 循环，但用上述方法计算 $dw$ 仍然需要一个循环遍历训练集，我们现在要做的就是将其向量化！

首先我们来看 $db$，不难发现 $db=\frac{1}{m}\sum_{i=1}^{m}dz^{(i)}$ ， 之前的讲解中，我们知道所有的$dz^{i)}$已经组成一个行向量 $dZ$了，所以在 **Python** 中，我们很容易地想到$db=\frac{1}{m}*np.sum(dZ)$；接下来看$dw$，我们先写出它的公式 $dw=\frac{1}{m}*X*dz^{T}$ 其中，$X$ 是一个行向量。因此展开后

$dw=\frac{1}{m}*(x^{(1)}dz^{(1)}+x^{(2)}dz^{(2)}+\ldots+x^{m}dz^{m})$ 。因此我们可以仅用两行代码进行计算：$db=\frac{1}{m}*np.sum(dZ)$， $dw=\frac{1}{m}*X*dz^{T}$。这样，我们就避免了在训练集上使用 for 循环。

现在，让我们回顾一下，看看我们之前怎么实现的逻辑回归，可以发现，没有向量化是非常低效的，如下图所示代码：

```
J = 0, dw_1 = 0, dw_2 = 0, db = 0
for i = 1 to m:
    z^(i) = w^T x^(i) + b
    a^(i) = σ(z^(i))
    J += -[y^(i) log a^(i) + (1 - y^(i)) log(1 - a^(i))]
    dz^(i) = a^(i) - y^(i)
    dw_1 += x_1^(i) dz^(i)
    dw_2 += x_2^(i) dz^(i)
    db += dz^(i)
J = J/m, dw_1 = dw_1/m, dw_2 = dw_2/m
db = db/m
```

我们的目标是不使用 **for** 循环，而是向量，我们可以这么做：

$$Z=w^{T}X+b=np.dot(w.T,X)+b$$

$$A=\sigma(Z)$$

$$dZ=A-Y$$

$$dw=\frac{1}{m}*X*dz^{T}$$

$$db=\frac{1}{m}*np.sum(dZ)$$

$$w:=w-a*dw$$

$$b:=b-a*db$$

现在我们利用前五个公式完成了前向和后向传播，也实现了对所有训练样本进行预测和求导，再利用后两个公式，梯度下降更新参数。我们的目的是不使用 **for** 循环，所以我们就通过一次迭代实现一次梯度下降，但如果你希望多次迭代进行梯度下降，那么仍然需要 **for** 循环，放在最外层。不过我们还是觉得一次迭代就进行一次梯度下降，避免使用任何循环比较舒服一些。

最后，我们得到了一个高度向量化的、非常高效的逻辑回归的梯度下降算法，我们将在下次视频中讨论 **Python** 中的 **Broadcasting** 技术。

## 2.12 Python 中的广播

这是一个不同食物(每 100g)中不同营养成分的卡路里含量表格，表格为 3 行 4 列，列表示不同的食物种类，从左至右依次为苹果，牛肉，鸡蛋，土豆。行表示不同的营养成分，从上到下依次为碳水化合物，蛋白质，脂肪。

Calories from Carbs, Proteins, Fats in 100g of different foods:

$$\begin{array}{r}
\\ \text{Carb} \\ \text{Protein} \\ \text{Fat}
\end{array}
\begin{array}{c}
\begin{array}{cccc} \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \end{array} \\
\begin{bmatrix} 56.0 & 0.0 & 4.4 & 68.0 \\ 1.2 & 104.0 & 52.0 & 8.0 \\ 1.8 & 135.0 & 99.0 & 0.9 \end{bmatrix}
\end{array}$$

那么，我们现在想要计算不同食物中不同营养成分中的卡路里百分比。

现在计算苹果中的碳水化合物卡路里百分比含量，首先计算苹果（100g）中三种营养成分卡路里总和 56+1.2+1.8 = 59，然后用 56/59 = 94.9%算出结果。

可以看出苹果中的卡路里大部分来自于碳水化合物，而牛肉则不同。

对于其他食物，计算方法类似。首先，按列求和，计算每种食物中（100g）三种营养成分总和，然后分别用不用营养成分的卡路里数量除以总和，计算百分比。

那么，能否不使用 for 循环完成这样的一个计算过程呢？假设上图的表格是一个 3 行 4 列的矩阵$A$，记为 $A_{3\times4}$，接下来我们要使用 **Python** 的 **numpy** 库完成这样的计算。我们打算使用两行代码完成，第一行代码对每一列进行求和，第二行代码分别计算每种食物每种营养成分的百分比。

在 **jupyter notebook** 中输入如下代码，按 shift+Enter 运行，输出如下。

```
In [6]:
import numpy as np

A = np.array([[56.0,0.0,4.4,68.0],
              [1.2,104.0,52.0,8.0],
              [1.8,135.0,99.0,0.9]])

print(A)
```

```
[[  56.     0.     4.4   68. ]
 [   1.2  104.    52.     8. ]
 [   1.8  135.    99.     0.9]]
```

下面使用如下代码计算每列的和，可以看到输出是每种食物(100g)的卡路里总和。

```
In [7]:
cal = A.sum(axis=0)
print(cal)
```

```
[  59.   239.   155.4   76.9]
```

其中 `sum` 的参数 `axis=0` 表示求和运算按列执行，之后会详细解释。

接下来计算百分比，这条指令将 $3\times4$的矩阵$A$除以一个$1\times4$的矩阵，得到了一个 $3\times4$ 的结果矩阵，这个结果矩阵就是我们要求的百分比含量。

```
In [8]: percentage = 100*A/cal.reshape(1,4)
        print(percentage)
```

```
[[ 94.91525424   0.           2.83140283  88.42652796]
 [  2.03389831  43.51464435  33.46203346  10.40312094]
 [  3.05084746  56.48535565  63.70656371   1.17035111]]
```

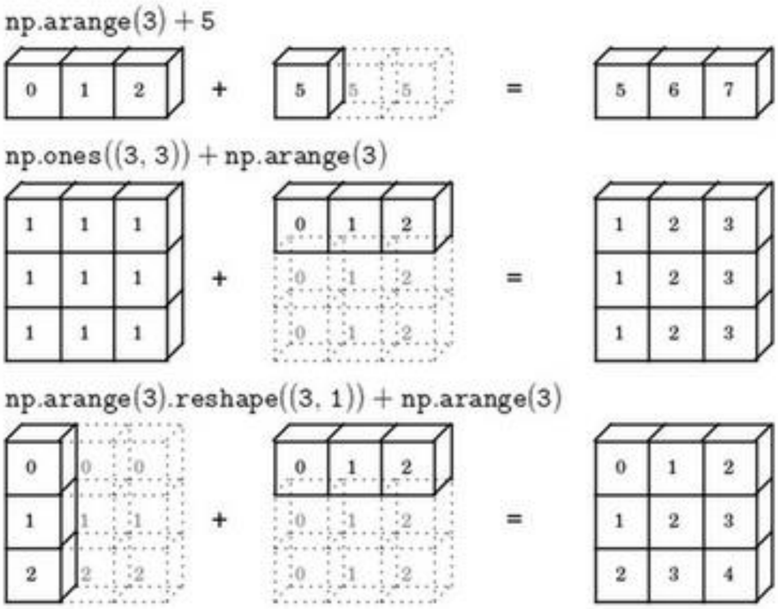
下面再来解释一下 A.sum(axis = 0)中的参数 axis。**axis 用来指明将要进行的运算是沿着哪个轴执行，在 numpy 中，0 轴是垂直的，也就是列，而 1 轴是水平的，也就是行。**

而第二个 A/cal.reshape(1,4)指令则调用了 **numpy** 中的广播机制。这里使用 $3 \times 4$ 的矩阵$A$除以 $1 \times 4$的矩阵$cal$。技术上来讲，其实并不需要再将矩阵$cal$ reshape(重塑)成 $1 \times 4$，因为矩阵$cal$本身已经是 $1 \times 4$了。但是当我们写代码时不确定矩阵维度的时候，通常会对矩阵进行重塑来确保得到我们想要的列向量或行向量。重塑操作 reshape 是一个常量时间的操作，时间复杂度是$O(1)$，它的调用代价极低。

那么一个 $3 \times 4$ 的矩阵是怎么和 $1 \times 4$的矩阵做除法的呢？让我们来看一些更多的广播的例子。在 **numpy** 中，当一个 $4 \times 1$的列向量与一个常数做加法时，实际上会将常数扩展为一个 $4 \times 1$的列向量，然后两者做逐元素加法。结果就是右边的这个向量。这种广播机制对于行向量和列向量均可以使用。再看下一个例子。

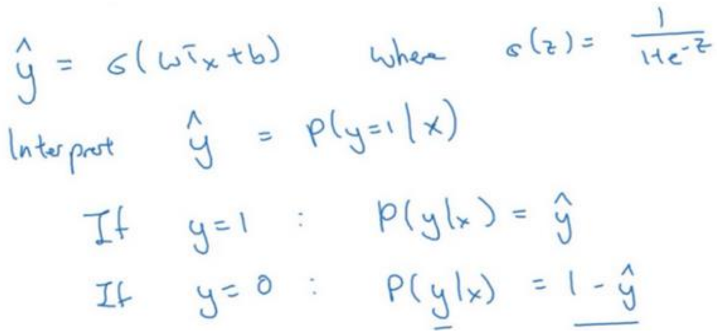
$$\underset{(m,n)\quad(2,3)}{\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}} + \underset{(1,n)\rightsquigarrow(m,n)\quad(2,3)}{\begin{bmatrix}100&200&300\\100&200&300\end{bmatrix}} = \begin{bmatrix}101&202&303\\104&205&306\end{bmatrix}$$

**如果两个数组的后缘维度的轴长度相符或其中一方的轴长度为 1，则认为它们是广播兼容的。广播会在缺失维度和轴长度为 1 的维度上进行。**总结一下 broadcasting，可以看看下面的图：



## 2.13 （选修）logistic 损失函数的解释

在前面的视频中，我们已经分析了逻辑回归的损失函数表达式，在这节选修视频中，我将给出一个简洁的证明来说明逻辑回归的损失函数为什么是这种形式。




回想一下，在逻辑回归中，需要预测的结果$\hat{y}$,可以表示为$\hat{y} = \sigma(w^T x + b)$，$\sigma$是我们熟悉的$S$型函数 $\sigma(z) = \sigma(w^T x + b) = \frac{1}{1+e^{-z}}$ 。我们约定 $\hat{y} = p(y = 1|x)$ ，即算法的输出$\hat{y}$ 是给定训练样本 $x$ 条件下 $y$ 等于 1 的概率。

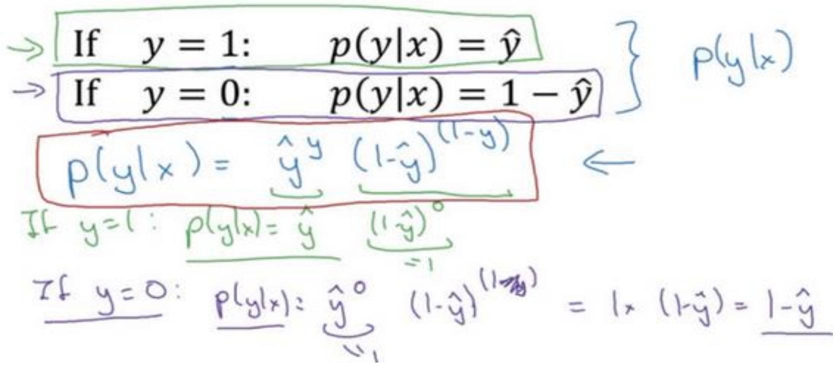
换句话说，如果$y = 1$，在给定训练样本 $x$ 条件下$y = \hat{y}$；反过来说，如果$y = 0$，在给定训练样本$x$条件下 $(y = 1 - \hat{y})$，因此，如果 $\hat{y}$ 代表 $y = 1$ 的概率，那么$1 - \hat{y}$就是 $y = 0$的概率。接下来，我们就来分析这两个条件概率公式。

$$\text{If} \quad y = 1: \quad p(y|x) = \hat{y}$$
$$\text{If} \quad y = 0: \quad p(y|x) = 1 - \hat{y}$$

这两个条件概率公式定义形式为 $p(y|x)$并且代表了 $y = 0$ 或者 $y = 1$ 这两种情况，我们可以将这两个公式合并成一个公式。需要指出的是我们讨论的是二分类问题的损失函数，因此，$y$的取值只能是 0 或者 1。上述的两个条件概率公式可以合并成如下公式：

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{(1-y)}$$

接下来我会解释为什么可以合并成这种形式的表达式：$(1 - \hat{y})$的$(1 - y)$次方这行表达式包含了上面的两个条件概率公式，我来解释一下为什么。



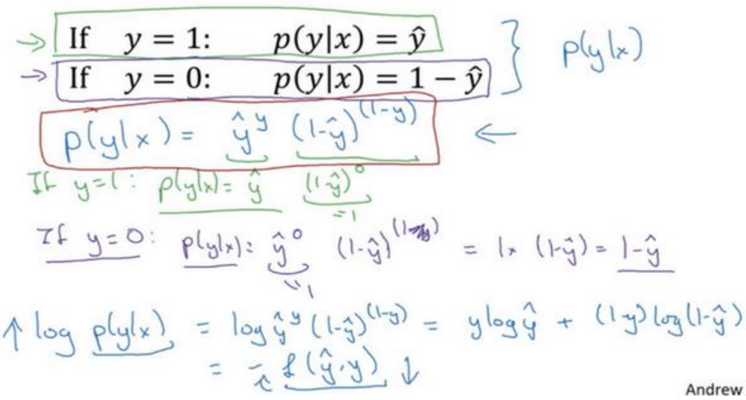

第一种情况，假设 $y = 1$，由于$y = 1$，那么$(\hat{y})^y = \hat{y}$，因为 $\hat{y}$的 1 次方等于$\hat{y}$，$1 - (1 - \hat{y})^{(1-y)}$的指数项$(1 - y)$等于 0，由于任何数的 0 次方都是 1，$\hat{y}$乘以 1 等于$\hat{y}$。因此当$y = 1$时 $p(y|x) = \hat{y}$（图中绿色部分）。

第二种情况，当 $y=0$ 时 $p(y|x)$ 等于多少呢？假设$y=0$，$\hat{y}$的$y$次方就是 $\hat{y}$ 的 0 次方，任何数的 0 次方都等于 1，因此 $p(y|x)=1\times(1-\hat{y})^{1-y}$ ，前面假设 $y=0$ 因此$(1-y)$就等于 1，因此 $p(y|x)=1\times(1-\hat{y})$。因此在这里当$y=0$时，$p(y|x)=1-\hat{y}$。这就是这个公式(第二个公式，图中紫色字体部分)的结果。

因此，刚才的推导表明 $p(y|x)=\hat{y}^{(y)}(1-\hat{y})^{(1-y)}$，就是 $p(y|x)$ 的完整定义。由于 log 函数是严格单调递增的函数，最大化 $log(p(y|x))$ 等价于最大化 $p(y|x)$ 并且地计算 $p(y|x)$ 的 log 对数，就是计算 $log(\hat{y}^{(y)}(1-\hat{y})^{(1-y)})$ (其实就是将 $p(y|x)$ 代入)，通过对数函数化简为：

$$ylog\,\hat{y}+(1-y)log(1-\hat{y})$$

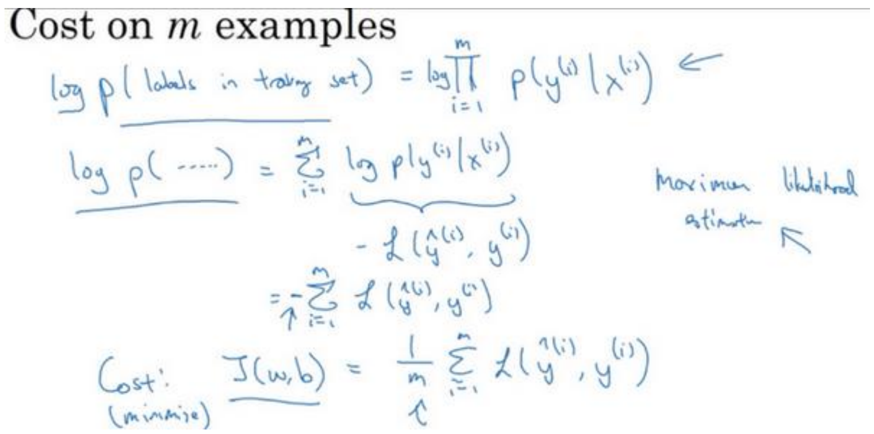
而这就是我们前面提到的损失函数的负数 $(-L(\hat{y},y))$ ，前面有一个负号的原因是当你训练学习算法时需要算法输出值的概率是最大的（以最大的概率预测这个值），然而在逻辑回归中我们需要最小化损失函数，因此最小化损失函数与最大化条件概率的对数 $log(p(y|x))$ 关联起来了，因此这就是单个训练样本的损失函数表达式。




在 $m$个训练样本的整个训练集中又该如何表示呢，让我们一起来探讨一下。

让我们一起来探讨一下，整个训练集中标签的概率，更正式地来写一下。假设所有的训练样本服从同一分布且相互独立，也即独立同分布的，所有这些样本的联合概率就是每个样本概率的乘积:

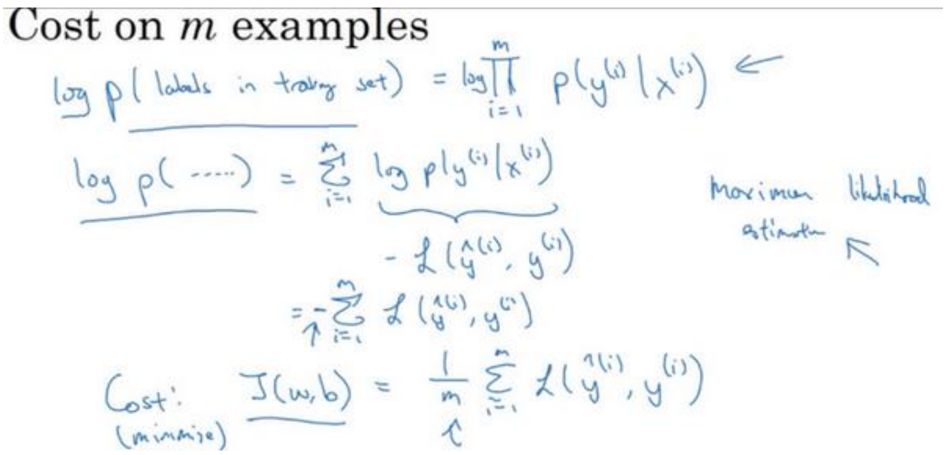
$$P(\text{labels in training set})=\prod_{i=1}^{m}P(y^{(i)}|x^{(i)})。$$



如果你想做最大似然估计，需要寻找一组参数，使得给定样本的观测值概率最大，但令这个概率最大化等价于令其对数最大化，在等式两边取对数：

$$logp(\text{labels in training set}) = log\prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} logP(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} -L(\hat{y}^{(i)}, y^{(i)})$$

在统计学里面，有一个方法叫做最大似然估计，即求出一组参数，使这个式子取最大值，也就是说，使得这个式子取最大值，$\sum_{i=1}^{m} -L(\hat{y}^{(i)}, y^{(i)})$，可以将负号移到求和符号的外面，$-\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$，这样我们就推导出了前面给出的 logistic 回归的成本函数$J(w,b) = \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{\hat{(i)}})$。




由于训练模型时，目标是让成本函数最小化，所以我们不是直接用最大似然概率，要去掉这里的负号，最后为了方便，可以对成本函数进行适当的缩放，我们就在前面加一个额外的常数因子$\frac{1}{m}$，即：

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})。$$

总结一下，为了最小化成本函数$J(w,b)$，我们从 **logistic** 回归模型的最大似然估计的角度出发，假设训练集中的样本都是独立同分布的条件下。尽管这节课是选修性质的，但还是感谢观看本节视频。我希望通过本节课您能更好地明白逻辑回归的损失函数，为什么是那种形式，明白了损失函数的原理，希望您能继续完成课后的练习，前面课程的练习以及本周的测验，在课后的小测验和编程练习中,祝您好运。

# 第三周：浅层神经网络

## 3.1 神经网络概述

公式 3.1：

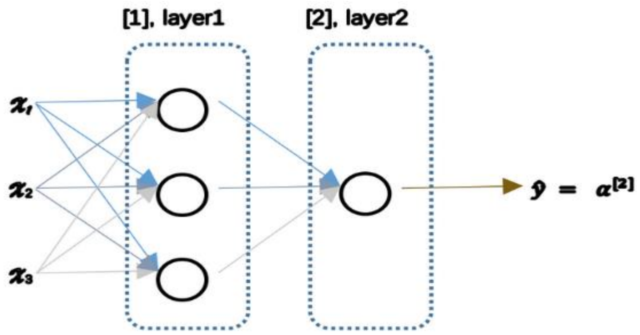
$$\left.\begin{array}{l} x \\ w \\ b \end{array}\right\} \Longrightarrow z = w^T x + b$$

如上所示，首先你需要输入特征$x$，参数$w$和$b$，通过这些你就可以计算出$z$，

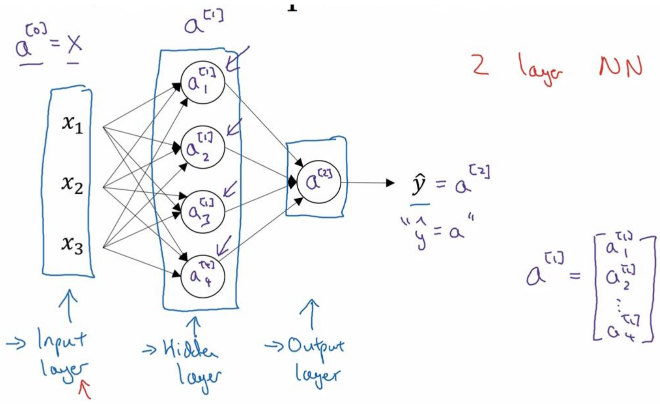
公式 3.2：

$$\left.\begin{array}{l} x \\ w \\ b \end{array}\right\} \Longrightarrow z = w^T x + b \Longrightarrow \alpha = \sigma(z)$$
$$\Longrightarrow L(a, y)$$

神经网络看起来是如下这个样子。你可以把许多 **sigmoid** 单元堆叠起来形成一个神经网络。对于图 3.1.1 中的节点，它包含了之前讲的计算的两个步骤：首先通过公式 3.1 计算出值$z$，然后通过$\sigma(z)$计算值$a$。



## 3.2 神经网络的表示




我们有输入特征$x_1$、$x_2$、$x_3$，它们被竖直地堆叠起来，这叫做神经网络的**输入层**。它包含了神经网络的输入；然后另外一层我们称之为**隐藏层**（图 3.2.1 的四个结点）。在本例中最后一层只由一个结点构成，而这个只有一个结点的层被称为**输出层**，它负责产生预测值。解释隐藏层的含义：在一个神经网络中，使用监督学习训练它的时候，训练集包含了输入$x$也包含了目标输出$y$，所以术语隐藏层的含义是在训练集中，这些中间结点的准确值我们是不知道的，也就是说看不见它们在训练集中应具有的值。你能看见输入的值，你也能看见输出的值，但是隐藏层中的东西，在训练集中你是无法看到的。所以这也解释了词语隐藏层，只是表示你无法在训练集中看到他们。

这里的记号$a^{[0]}$可以用来表示输入特征。$a$表示激活，它意味着网络中不同层的值会传递到它们后面的层中，输入层将$x$传递给隐藏层，所以我们将输入层的激活值称为$a^{[0]}$；下一层即隐藏层也同样会产生一些激活值，那么我将其记作$a^{[1]}$，第一个单元或结点我们将其表示为$a_1^{[1]}$，第二个结点的值我们记为$a_2^{[1]}$，以此类推。所以这是一个四维的向量，如果写成 Python 代码，那么它是一个规模为 4x1 的矩阵或一个大小为 4 的列向量，因为在本例中有四个结点或者单元，或者称为四个隐藏层单元；

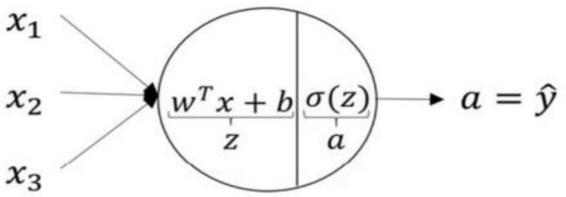
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

最后输出层将产生某个数值$a$，它只是一个单独的实数，所以的$\hat{y}$值将取为$a^{[2]}$。这与逻辑回归很相似，在逻辑回归中，我们有$\hat{y}$直接等于$a$，在逻辑回归中我们只有一个输出层，所以我们没有用带方括号的上标。但是在神经网络中，我们将使用这种带上标的形式来明确地指出这些值来自于哪一层，有趣的是在约定俗成的符号传统中，在这里你所看到的这个例子，只能叫做一个两层的神经网络。原因是当我们计算网络层数时，输入层是不算入总层数内，所以隐藏层是第一层，输出层是第二层。第二个惯例是我们将输入层称为第零层，所以在技术上，这仍然是一个三层的神经网络，因为这里有输入层、隐藏层，还有输出层。

# 3.3 计算神经网络的输出

**神经网络的计算**

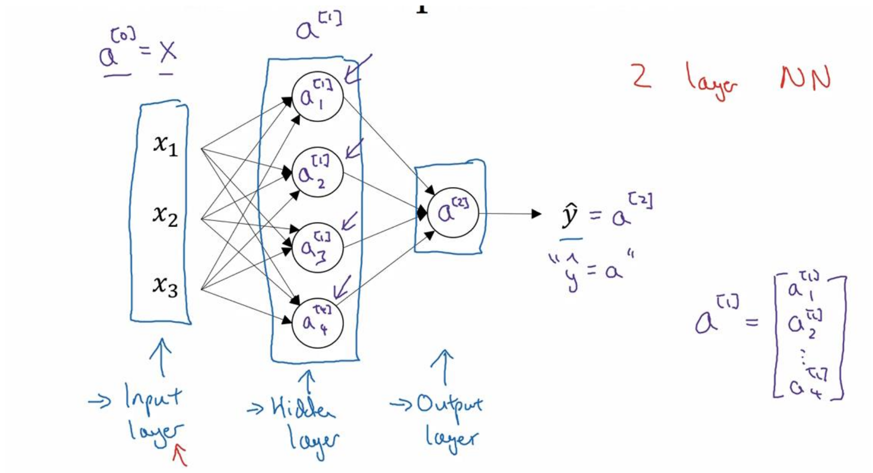
关于神经网络是怎么计算的，从我们之前提及的逻辑回归开始，用圆圈表示神经网络的计算单元，逻辑回归的计算有两步，首先你按步骤计算出$z$，然后你以 **sigmoid** 函数为激活函数计算$z$（得出$a$），一个神经网络只是这样子做了好多次重复计算。




$$z = w^Tx + b$$

$$a = \sigma(z)$$

现在回顾下只有一个隐藏层的简单两层**神经网络结构**：




$x$表示输入特征，$a$表示每个神经元的输出，$W$表示特征的权重，上标表示神经网络的层数（隐藏层为 1），下标表示该层的第几个神经元。这是神经网络的**符号惯例**。

我们从隐藏层的第一个神经元开始计算，从上图可以看出，输入与逻辑回归相似，这个神经元的计算与逻辑回归一样分为两步，小圆圈代表了计算的两个步骤。

第一步，计算$z_1^{[1]}$，$z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]}$。

第二步，通过激活函数计算$a_1^{[1]}$，$a_1^{[1]} = \sigma(z_1^{[1]})$。

隐藏层的第二个以及后面两个神经元的计算过程一样，只是注意符号表示不同，最终分别得到$a_2^{[1]}$、$a_3^{[1]}$、$a_4^{[1]}$，详细结果见下:

$W_1^{[1]}: (3 \times 1)$

$$z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]}, a_1^{[1]} = \sigma(z_1^{[1]})$$

$W_1^{[1]T}: (1 \times 3)$

$x: (3 \times 1)$

$$z_2^{[1]} = w_2^{[1]T}x + b_2^{[1]}, a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T}x + b_3^{[1]}, a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T}x + b_4^{[1]}, a_4^{[1]} = \sigma(z_4^{[1]})$$

**向量化计算**

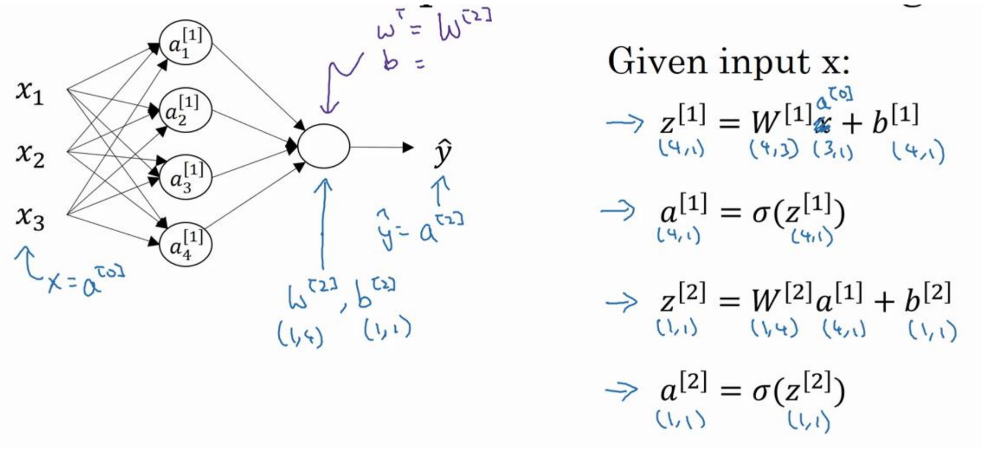
向量化过程是将神经网络中的一层神经元参数纵向堆积起来，如隐藏层中的 $w.T$ 纵向堆积起来变成一个(4,3)的矩阵，用符号$W^{[1]}$表示。

$$\overset{4\times 1}{\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}} = \underset{1\times 3}{\overset{4\times 3}{\overbrace{\begin{bmatrix} \ldots W_1^{[1]T} \ldots \\ \ldots W_2^{[1]T} \ldots \\ \ldots W_3^{[1]T} \ldots \\ \ldots W_4^{[1]T} \ldots \end{bmatrix}}^{W^{[1]}}}} * \overset{3\times 1}{\overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}^{input}} + \overset{4\times 1}{\overbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}^{b^{[1]}}}, \qquad a^{[1]} = \overset{4\times 1}{\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}} = \sigma(z^{[1]})$$

因此，$\color{red}{\boldsymbol{z^{[n]} = w^{[n]}x + b^{[n]}}}$，$\color{red}{\boldsymbol{a^{[n]} = \sigma(z^{[n]})}}$

$$\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} & w_{33}^{(1)} \\ w_{14}^{(1)} & w_{24}^{(1)} & w_{34}^{(1)} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

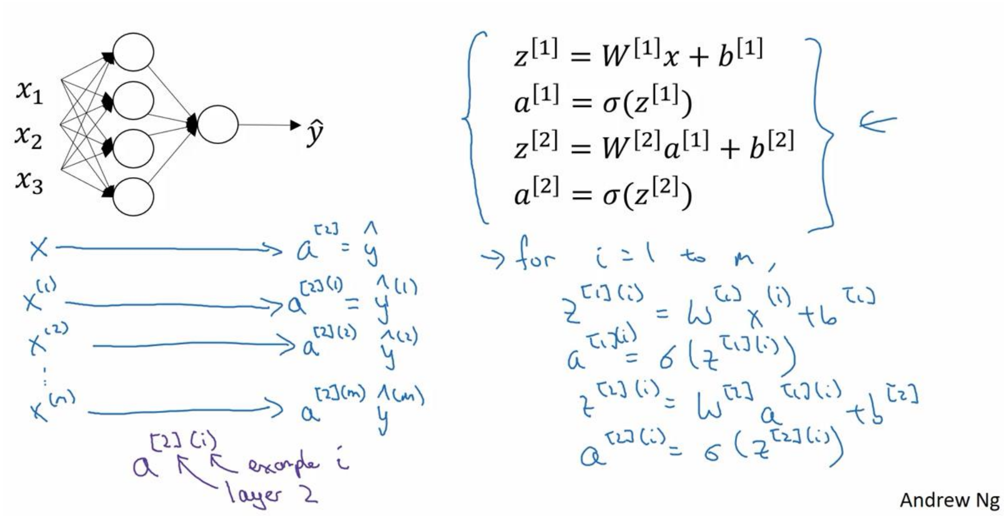
对于神经网络的第一层，给予一个输入$x$，得到$a^{[1]}$，$x$可以表示为$a^{[0]}$。通过相似的衍生你会发现，后一层的表示同样可以写成类似的形式，得到$a^{[2]}$，$\hat{y} = a^{[2]}$。



# 3.4 多样本向量化

在上一个视频，了解到如何针对于单一的训练样本，在神经网络上计算出预测值。在这个视频，将会了解到如何向量化多个训练样本，并计算出结果。

逻辑回归是将各个训练样本组合成矩阵，对矩阵的各列进行计算。神经网络是通过对逻辑回归中的等式简单的变形，让神经网络计算出输出值。这种计算是所有的训练样本同时进行的，以下是实现它具体的步骤：




对于一个给定的输入特征向量$X$，这四个等式可以计算出$\alpha^{[2]}$等于$\hat{y}$。这是针对于单一的训练样本。如果有$m$个训练样本，那么就需要重复这个过程。用第一个训练样本$x^{[1]}$来计算出预测值$\hat{y}^{[1]}$，就是第一个训练样本上得出的结果。然后，用$x^{[2]}$来计算出预测值$\hat{y}^{[2]}$，……循环往复，直至用$x^{[m]}$计算出$\hat{y}^{[m]}$。用激活函数表示法，如上图左下所示，它写成$a^{[2](1)}$、$a^{[2](2)}$和$a^{[2](m)}$。【注】：$a^{[2](i)}$，$(i)$是指第$i$个训练样本而[2]是指第二层。

如果有一个非向量化形式的实现，而且要计算出它的预测值，对于所有训练样本，需要让$i$从 1 到$m$实现这四个等式：

$$z^{[1](i)} = W^{[1](i)}x^{(i)} + b^{[1](i)}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2](i)}a^{[1](i)} + b^{[2](i)}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

对于上面的这个方程中的 $^{(i)}$，是所有依赖于训练样本的变量，即将$(i)$添加到$x$，$z$和$a$。

所以，希望通过这个细节可以更快地正确实现这些算法。接下来讲讲如何向量化这些：

公式 3.12：

$$X = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

公式 3.13：

$$Z^{[1]}=\begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

公式 3.14：

$$A^{[1]}=\begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ \alpha^{[1](1)} & \alpha^{[1](2)} & \cdots & \alpha^{[1](m)} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

公式 3.15：

$$\left.\begin{aligned} z^{[1](i)} = W^{[1](i)}x^{(i)} + b^{[1]} \\ \alpha^{[1](i)} = \sigma(z^{[1](i)}) \\ z^{[2](i)} = W^{[2](i)}\alpha^{[1](i)} + b^{[2]} \\ \alpha^{[2](i)} = \sigma(z^{[2](i)}) \end{aligned}\right\} \Longrightarrow \begin{cases} A^{[1]} = \sigma(z^{[1]}) \\ z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} = \sigma(z^{[2]}) \end{cases}$$

以此类推，从小写的向量$x$到这个大写的矩阵$X$，只是通过组合$x$向量在矩阵的各列中。

## 3.5 向量化实现的解释

我们先手动对几个样本计算一下前向传播，看看有什么规律：

$$z^{[1](1)} = W^{[1]}x^{(1)} + b^{[1]}$$

$$z^{[1](2)} = W^{[1]}x^{(2)} + b^{[1]}$$

$$z^{[1](3)} = W^{[1]}x^{(3)} + b^{[1]}$$

这里，为了描述的简便，我们先忽略掉 $b^{[1]}$后面你将会看到利用 **Python** 的广播机制，可以很容易的将$b^{[1]}$ 加进来。

现在 $W^{[1]}$ 是一个矩阵，$x^{(1)}, x^{(2)}, x^{(3)}$都是列向量，矩阵乘以列向量得到列向量，下面将它们用图形直观的表示出来: 公式 3.17：

$$W^{[1]}x = \begin{bmatrix} \cdots \\ \cdots \\ \cdots \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x^{(1)} & x^{(2)} & x^{(3)} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ w^{(1)}x^{(1)} & w^{(1)}x^{(2)} & w^{(1)}x^{(3)} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} =$$

$$\begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} = Z^{[1]}$$

从图中可以看出，当加入更多样本时，只需向矩阵$X$中加入更多列。从这里我们也可以了解到，为什么之前我们对单个样本的计算要写成 $z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$ 这种形式，因为当有不同的训练样本时，将它们堆到矩阵$X$的各列中，那么它们的输出也就会相应的堆叠到矩阵 $Z^{[1]}$ 的各列中。现在我们就可以直接计算矩阵 $Z^{[1]}$ 加上$b^{[1]}$，因为列向量 $b^{[1]}$ 和矩阵 $Z^{[1]}$ 的列向量有着相同的尺寸，而 **Python** 的广播机制对于这种矩阵与向量直接相加的处理方式是，将向量与矩阵的每一列相加。 所以这一节只是说明了为什么公式 $Z^{[1]} = W^{[1]}X + b^{[1]}$ 是前向传播的第一步计算的正确向量化实现，但事实证明，类似的分析可以发现，前向传播的其它步也可以使用非常相似的逻辑，即如果将输入按列向量横向堆叠进矩阵，那么通过公式计算之后，也能得到成列堆叠的输出。

## 3.6 激活函数

在神经网路的前向传播中，$a^{[1]}=\sigma(z^{[1]})$和$a^{[2]}=\sigma(z^{[2]})$这两步会使用到 **sigmoid** 函数。**sigmoid** 函数在这里被称为激活函数。更通常的情况下，使用不同的函数$g(z^{[1]})$，$g$可以是除了 **sigmoid** 函数意外的非线性函数。**tanh** 函数或者双曲正切函数是总体上都优于 **sigmoid** 函数的激活函数。

如图，$a=tan(z)$的值域是位于+1 和-1 之间。**tanh** 函数是 **sigmoid** 的向下平移和伸缩后的结果。对它进行了变形后，穿过了(0,0)点，并且值域介于+1 和-1 之间。

$$a=tanh(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}},\qquad a'(z)=1-[f(z)]^2$$

结果表明，如果在隐藏层上使用函数 $g(z^{[1]})=tanh(z^{[1]})$ 效果总是优于 **sigmoid** 函数。因为函数值域在-1 和+1 的激活函数，其均值是更接近零均值的。在训练一个算法模型时，如果使用 **tanh** 函数代替 **sigmoid** 函数中心化数据，使得数据的平均值更接近 0 而不是 0.5.

在讨论优化算法时，我基本已经不用 **sigmoid** 激活函数了，**tanh** 函数在所有场合都优于 **sigmoid** 函数。但有一个例外：在二分类的问题中，对于输出层，因为$y$的值是 0 或 1，所以想让$\hat{y}$的数值介于 0 和 1 之间，而不是在-1 和+1 之间。需要使用 **sigmoid** 激活函数。

在这个例子里看到的是，对隐藏层使用 **tanh** 激活函数，输出层使用 **sigmoid** 函数。

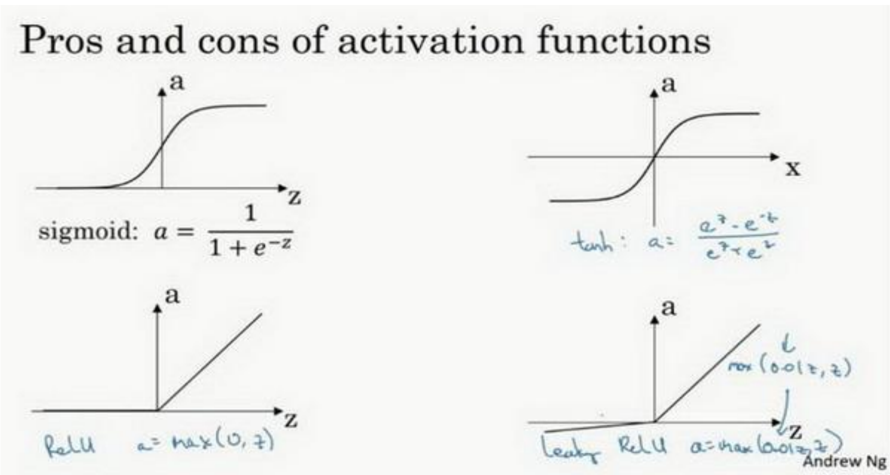
sigmoid 函数和 tanh 函数两者共同的缺点是，在$z$特别大或者特别小的情况下，导数的梯度或者函数的斜率会变得特别小，最后就会接近于 0，导致降低梯度下降的速度。

另一个激活函数是：修正线性单元的函数（**Recified Linear Unit**）。

$$a=max(0,z),\qquad a'(z)=\begin{cases}1, & z>0\\ 0, & z\le 0\end{cases}$$

所以，只要$z$是正值的情况下，导数恒等于 1，当$z$是负值的时候，导数恒等于 0。从实际上来说，当使用$z$的导数时，$z$=0 的导数是没有定义的。

这里也有另一个版本的 **Relu** 被称为 **Leaky Relu**。当$z$是负值时，这个函数的值不是等于 0，而是轻微的倾斜。这个函数通常比 **Relu** 激活函数效果要好，尽管在实际中 **Leaky ReLu** 使用的并不多。



第一，在$z$的区间变动很大的情况下，激活函数的导数或者激活函数的斜率都会远大于0，在程序实现就是一个 **if-else** 语句，而 **sigmoid** 函数需要进行浮点四则运算，在实践中，使用 **ReLu** 激活函数神经网络通常会比使用 **sigmoid** 或者 **tanh** 激活函数学习的更快。

第二，**sigmoid** 和 **tanh** 函数的导数在正负饱和区的梯度都会接近于 0，这会造成梯度弥散，而 **Relu** 和 **Leaky ReLu** 函数大于 0 部分都为常数，不会产生梯度弥散现象。(同时应该注意到的是，**Relu** 进入负半区的时候，梯度为 0，神经元此时不会训练，产生所谓的稀疏性，而 **Leaky ReLu** 不会有这问题)。$z$在 **ReLu** 的梯度一半都是 0，但是，有足够的隐藏层使得 z 值大于 0，所以对大多数的训练数据来说学习过程仍然可以很快。

## 3.7 为什么需要非线性激活函数？

这是神经网络正向传播的方程，现在我们去掉函数$g$，然后令$a^{[1]} = z^{[1]}$，或者我们也可以令$g(z) = z$，这个有时被叫做线性激活函数（更学术点的名字是恒等激励函数，因为它们就是把输入值输出）。为了说明问题我们把$a^{[2]} = z^{[2]}$，那么这个模型的输出$y$或仅仅只是输入特征$x$的线性组合。

如果我们改变前面的式子，令：

(1) $a^{[1]} = z^{[1]} = W^{[1]}x + b^{[1]}$

(2) $a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ 将式子(1)代入式子(2)中，则： $a^{[2]} = z^{[2]} = W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]}$

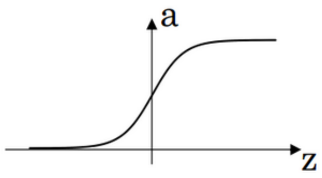
(3) $a^{[2]} = z^{[2]} = W^{[2]}W^{[1]}x + W^{[2]}b^{[1]} + b^{[2]}$

简化多项式得 $a^{[2]} = z^{[2]} = W'x + b'$，如果你是用线性激活函数或者叫恒等激励函数，那么神经网络只是把输入线性组合再输出。总而言之，不能在隐藏层用线性激活函数，可以用 **ReLU** 或者 **tanh** 或者 **leaky ReLU** 或者其他的非线性激活函数，唯一可以用线性激活函数的通常就是输出层。在这之外，在隐层使用线性激活函数非常少见。因为房价都是非负数，所以我们也可以在输出层使用 **ReLU** 函数这样你的$\hat{y}$都大于等于 0。

# 3.8 激活函数的导数

在神经网络中反向传播中，你真的需要计算激活函数的斜率或者导数。针对以下四种激活，求其导数如下：

1）**sigmoid activation function**




$$g(z) = \frac{1}{1+e^{-z}}$$

其具体的求导如下：

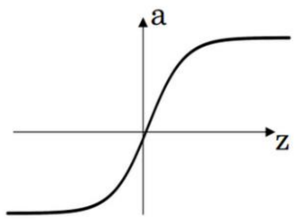
$$\frac{d}{dz}g(z) = \frac{1}{1+e^{-z}}(1-\frac{1}{1+e^{-z}}) = g(z)(1-g(z))$$

注：在神经网络中

$$a = g(z);$$

$$g(z)' = \frac{d}{dz}g(z) = a(1-a)$$

2) **Tanh activation function**



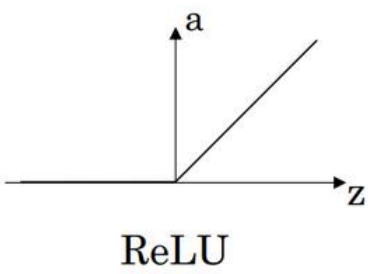

$$g(z) = \tanh(z)$$

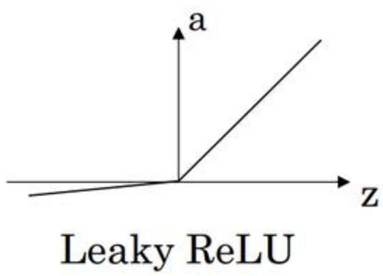
其具体的求导如下：

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

$$\frac{d}{\mathrm{dz}}g(z) = 1-(tanh(z))^2$$

3）**Rectified Linear Unit (ReLU)**




$$g(z) = max(0,z)$$

$$g(z)' = \begin{cases} 0 & \text{if z } < \text{ 0} \\ 1 & \text{if z } > \text{ 0} \\ undefined & \text{if z } = \text{ 0} \end{cases}$$

4）**Leaky linear unit (Leaky ReLU)** 与 **ReLU** 类似

$$g(z) = max(0.01z, z)$$

$$g(z)' = \begin{cases} 0.01 & \text{if z } < \text{ 0} \\ 1 & \text{if z } > \text{ 0} \\ undefined & \text{if z } = \text{ 0} \end{cases}$$

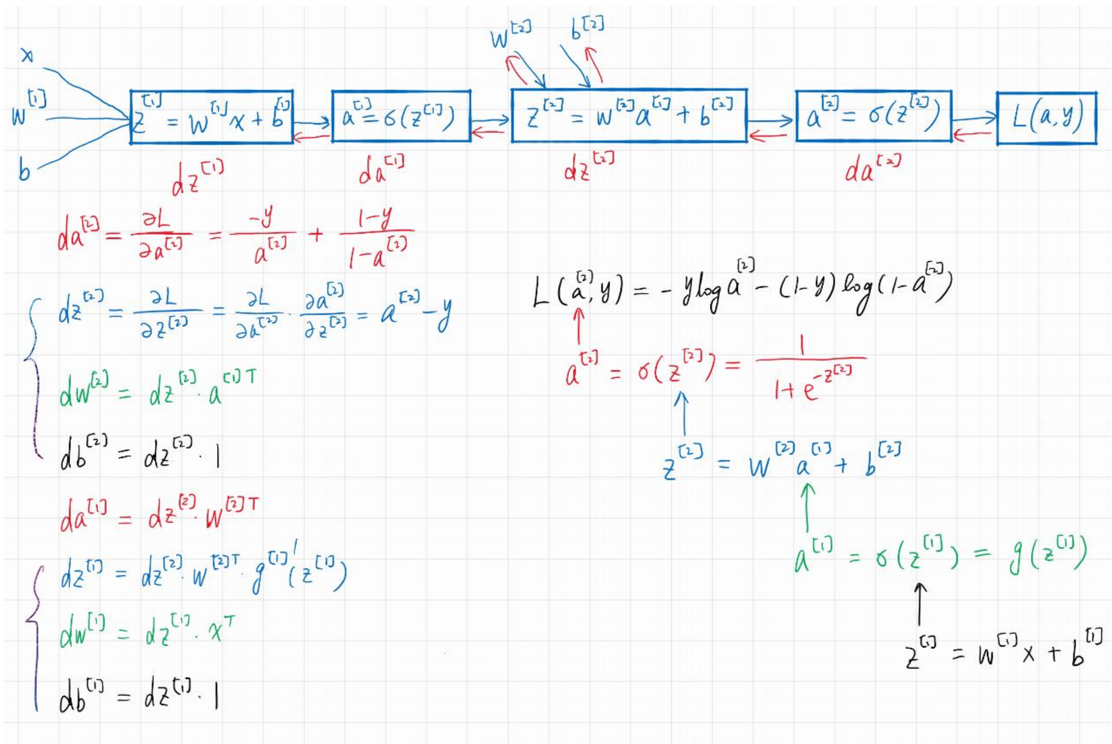
## 3.9 神经网络的梯度下降

你的单隐层神经网络会有$W^{[1]}$，$b^{[1]}$，$W^{[2]}$，$b^{[2]}$这些参数，还有个$n_x$表示输入特征的个数，$n^{[1]}$表示隐藏单元个数，$n^{[2]}$表示输出单元个数。

矩阵$W^{[1]}$的维度就是$(n^{[1]},n^{[0]})$，$b^{[1]}$就是$n^{[1]}$维向量，可以写成$(n^{[1]},1)$，就是一个的列向量。 矩阵$W^{[2]}$的维度就是$(n^{[2]},n^{[1]})$，$b^{[2]}$的维度就是$(n^{[2]},1)$维度。

你还有一个神经网络的成本函数，假设你在做二分类任务，那么成本函数 **Cost function**:

$J(W^{[1]},b^{[1]},W^{[2]},b^{[2]})=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y},y)$ ，**loss function** 和 **logistic** 回归完全一样。




**Summary of gradient descent**

| | |
|---|---|
| $dz^{[2]}=a^{[2]}-y$ | $dZ^{[2]}=A^{[2]}-Y$ |
| $dW^{[2]}=dz^{[2]}a^{[1]^T}$ | $dW^{[2]}=\frac{1}{m}dZ^{[2]}A^{[1]^T}$ |
| $db^{[2]}=dz^{[2]}$ | $db^{[2]}=\frac{1}{m}np.sum(dZ^{[2]},axis=1,keepdims=True)$ |
| $dz^{[1]}=W^{[2]T}dz^{[2]}*g^{[1]\prime}(z^{[1]})$ | $dZ^{[1]}=W^{[2]T}dZ^{[2]}*g^{[1]\prime}(Z^{[1]})$ |
| $dW^{[1]}=dz^{[1]}x^T$ | $dW^{[1]}=\frac{1}{m}dZ^{[1]}X^T$ |
| $db^{[1]}=dz^{[1]}$ | $db^{[1]}=\frac{1}{m}np.sum(dZ^{[1]},axis=1,keepdims=True)$ |

训练参数需要做梯度下降，在训练神经网络的时候，随机初始化参数很重要，而不是初始化成全零。当你参数初始化成某些值后，每次梯度下降都会循环计算以下预测值：

$\hat{y}^{(i)},(i=1,2,\ldots,m)$

公式 3.28： $dW^{[1]}=\frac{dJ}{dW^{[1]}},db^{[1]}=\frac{dJ}{db^{[1]}}$

公式 3.29： $dW^{[2]}=\frac{dJ}{dW^{[2]}},db^{[2]}=\frac{dJ}{db^{[2]}}$

其中

公式 3.30： $W^{[1]}\Rightarrow W^{[1]}-adW^{[1]},b^{[1]}\Rightarrow b^{[1]}-adb^{[1]}$

公式 3.31： $W^{[2]}\Rightarrow W^{[2]}-adW^{[2]},b^{[2]}\Rightarrow b^{[2]}-adb^{[2]}$

正向传播方程如下（之前讲过）：

**forward propagation**：

(1) $z^{[1]}=W^{[1]}x+b^{[1]}$

(2) $a^{[1]}=\sigma(z^{[1]})$

(3) $z^{[2]}=W^{[2]}a^{[1]}+b^{[2]}$

(4) $a^{[2]}=g^{[2]}(z^{[z]})=\sigma(z^{[2]})$

反向传播方程如下:

**back propagation**：

公式 3.32： $dz^{[2]}=A^{[2]}-Y,Y=\begin{bmatrix}y^{[1]} & y^{[2]} & \cdots & y^{[m]}\end{bmatrix}$

公式 3.33： $dW^{[2]}=\frac{1}{m}dz^{[2]}A^{[1]T}$

公式 3.34： $db^{[2]}=\frac{1}{m}np.sum(dz^{[2]},axis=1,keepdims=True)$

公式 3.35：

$$dz^{[1]}=\underbrace{W^{[2]T}\mathrm{d}z^{[2]}}_{(n^{[1]},m)} \quad * \quad \underbrace{g^{[1]'}}_{activation\ function\ of\ hidden\ layer} \quad * \quad \underbrace{(z^{[1]})}_{(n^{[1]},m)}$$

公式 3.36： $dW^{[1]}=\frac{1}{m}dz^{[1]}x^{T}$

公式 3.37： $\underbrace{db^{[1]}}_{(n^{[1]},1)}=\frac{1}{m}np.sum(dz^{[1]},axis=1,keepdims=True)$

上述是反向传播的步骤，注：这些都是针对所有样本进行过向量化，$Y$是$1\times m$的矩阵；这里 np.sum 是 **python** 的 **numpy** 命令，axis=1 表示水平相加求和，keepdims 是防止 python 输出那些古怪的秩数$(n,)$，加上这个确保阵矩阵$db^{[2]}$这个向量输出的维度为$(n,1)$这样标准的形式。

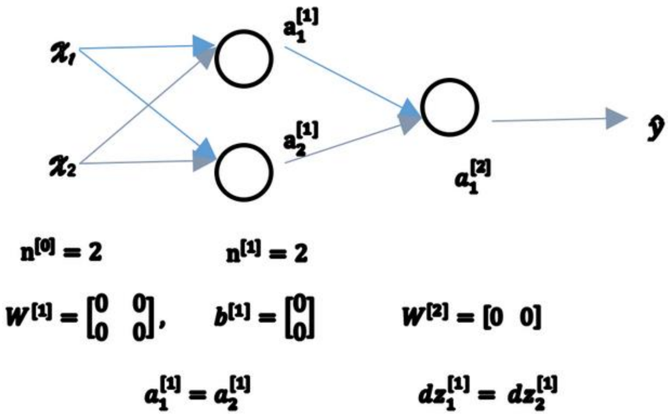
目前为止，我们计算的都和 **Logistic** 回归十分相似，但当你开始计算反向传播时，你需要计算，是隐藏层函数的导数，输出在使用 **sigmoid** 函数进行二元分类。这里是进行逐个元素乘积，因为$W^{[2]T}dz^{[2]}$和$(z^{[1]})$这两个都为$(n^{[1]},m)$矩阵；

以上就是正向传播的 4 个方程和反向传播的 6 个方程。如果你要实现这些算法，你必须正确执行正向和反向传播运算，你必须能计算所有需要的导数，用梯度下降来学习神经网络的参数；你也可以像深度学习从业者一样直接实现这个算法，不去了解其中的知识。

## 3.10 随机初始化

对于逻辑回归，把权重初始化为 0 当然也是可以的。但是对于一个神经网络，如果你把权重或者参数都初始化为 0，那么梯度下降将不会起作用。

让我们看看这是为什么。有两个输入特征，$n^{[0]}=2$，2 个隐藏层单元$n^{[1]}$就等于 2。因此与一个隐藏层相关的矩阵，或者说$W^{[1]}$是 2*2 的矩阵，假设把它初始化为 0 的 2*2 矩阵，$b^{[1]}$也等于 $[0\ 0]^T$，把偏置项$b$初始化为 0 是合理的，但是把$w$初始化为 0 就有问题了。如果按照这样初始化的话，你总会发现$a_1^{[1]}$ 和 $a_2^{[1]}$相等。因为两个隐含单元计算同样的函数，当你做反向传播计算时，这会导致$\mathrm{dz}_1^{[1]}$ 和 $\mathrm{dz}_2^{[1]}$也会一样，对称这些隐含单元会初始化得一样，这样输出的权值也会一模一样，由此$W^{[2]}$等于$[0\ 0]$；




如果这样初始化这个神经网络，那么这两个隐含单元就会完全一样，因此他们完全对称，也就意味着计算同样的函数，并且最终经过每次训练的迭代，这两个隐含单元仍然是同一个函数。$dW$会是一个这样的矩阵，每一行有同样的值因此我们做权重更新把权重$W^{[1]}\Longrightarrow W^{[1]}-adW$每次迭代后的$W^{[1]}$，第一行等于第二行。

由此可以推导，如果你把权重都初始化为 0，那么由于隐含单元开始计算同一个函数，所有的隐含单元就会对输出单元有同样的影响。一次迭代后同样的表达式结果仍然是相同的，即隐含单元仍是对称的。通过推导，两次、三次、无论多少次迭代，不管你训练网络多长时间，隐含单元仍然计算的是同样的函数。因此这种情况下超过 1 个隐含单元也没什么意义，因为他们计算同样的东西。

如果你要初始化成 0，由于所有的隐含单元都是对称的，无论你运行梯度下降多久，他们一直计算同样的函数，这没有任何帮助，因为想要两个不同的隐含单元计算不同的函数，这个问题的解决方法就是随机初始化参数。

你应该这么做：把$W^{[1]}$设为 `np.random.randn(2,2)`(生成高斯分布)，通常再乘上一个小的数，比如 0.01，这样把它初始化为很小的随机数。然后$b$没有这个对称的问题（叫做 **symmetry breaking problem**），所以可以把 $b$ 初始化为 0，因为只要随机初始化$W$你就有不同的隐含单元计算不同的东西，因此不会有 **symmetry breaking** 问题了。相似的，对于$W^{[2]}$

你可以随机初始化，$b^{[2]}$可以初始化为 0。

$W^{[1]} = np.random.randn(2,2) \ * \ 0.01$ ,

$b^{[1]} = np.zeros((2,1))$

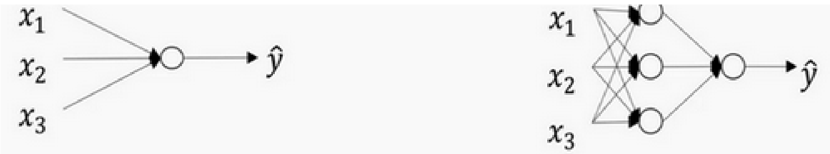
$W^{[2]} = np.random.randn(2,2) \ * \ 0.01$ , $b^{[2]} = 0$

你也许会疑惑，这个常数从哪里来，为什么是 0.01，而不是 100 或者 1000。我们通常倾向于初始化为很小的随机数。因为如果你用 **tanh** 或者 **sigmoid** 激活函数，或者说只在输出层有一个 **Sigmoid**，如果（数值）波动太大，当你计算激活值时$z^{[1]} = W^{[1]}x + b^{[1]}$ , $a^{[1]} = \sigma(z^{[1]}) = g^{[1]}(z^{[1]})$如果$W$很大，$z$就会很大。$z$的一些值$a$就会很大或者很小，因此这种情况下你很可能停在 **tanh/sigmoid** 函数的平坦的地方(见图 3.8.2)，这些地方梯度很小也就意味着梯度下降会很慢，因此学习也就很慢。

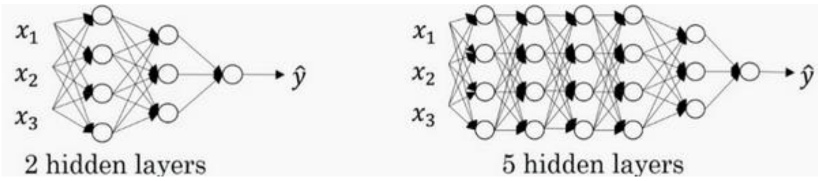
# 第四周：深层神经网络

## 4.1 深层神经网络

我们学习了**只有一个单独隐藏层**的神经网络的正向传播和反向传播，还有逻辑回归，并且你还学到了向量化。本周所要做的是把这些理念集合起来，就可以执行你自己的深度神经网络。我们已知逻辑回归，结构如下图左边。一个隐藏层的神经网络，结构下图右边：

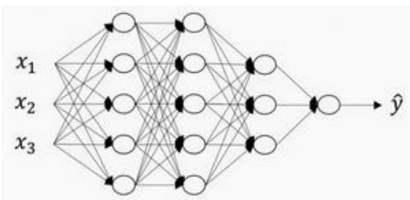



注意：神经网络的层数是这么定义的：**从左到右，由 0 开始定义**，比如上边右图，$x_1$、$x_2$、$x_3$，这层是第 0 层，这层右边的隐藏层是第 1 层，由此类推。如下图左边是两个隐藏层的神经网络，右边是 5 个隐藏层的神经网络。




但是在过去的几年中，**DLI**（深度学习学院 **deep learning institute**）已经意识到有一些函数，只有非常深的神经网络能学会，而更浅的模型则办不到。尽管对于任何给定的问题很难去提前预测到底需要多深的神经网络，所以先去尝试逻辑回归，尝试一层隐藏层，然后两层隐含层，然后把**隐含层的数量看做是一个可以自由选择大小的超参数**，然后再保留交叉验证数据上评估，或者用你的开发集来评估。

我们再看下深度学习的符号定义：下图是一个四层的神经网络，有三个隐藏层。第一层 5 个神经元数目，第二层 5 个，第三层 3 个。我们用 L 表示层数，$L=4$，输入层的索引为“0”，$n^{[0]}=n_x=3$，第一个隐藏层$n^{[1]}=5$，表示有 5 个隐藏神经元，同理$n^{[2]}=5$，$n^{[3]}=3$，$n^{[4]}$=$n^{[L]}=1$（输出单元为 1）。

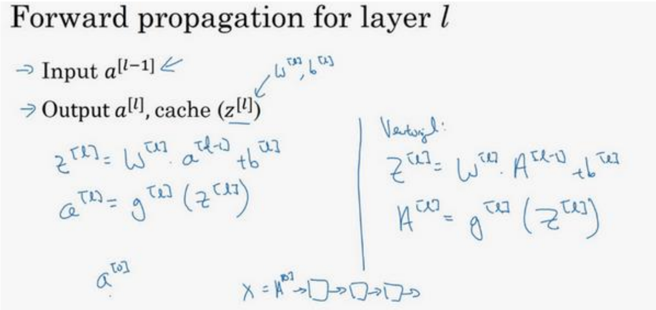



在不同层所拥有的神经元的数目，对于每层 *l* 都用$a^{[l]}$来记作 *l* 层激活后结果，我们会在后面看到在正向传播时，最终能你会计算出$a^{[l]}$，等于这个神经网络所预测的输出结果。

通过用激活函数 $g$ 计算$z^{[l]}$，激活函数也被索引为层数$l$，然后我们用$w^{[l]}$来记作在 *l* 层计算$z^{[l]}$值的权重。类似的，$z^{[l]}$里的方程$b^{[l]}$也一样。

## 4.2 前向传播和反向传播

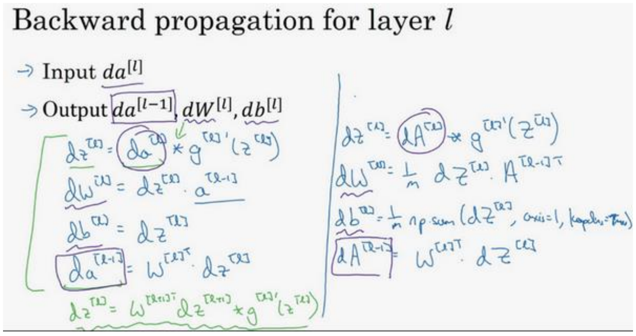
先讲前向传播，输入$a^{[l-1]}$，输出是$a^{[l]}$，缓存为$z^{[l]}$；从实现的角度来说我们可以缓存下$w^{[l]}$和$b^{[l]}$，这样更容易在不同的环节中调用函数。




- 前向传播的步骤可以写成： $z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]}$，$a^{[l]} = g^{[l]}\left(z^{[l]}\right)$
- 向量化实现过程可以写成： $z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$，$A^{[l]} = g^{[l]}(Z^{[l]})$

前向传播需要喂入$A^{[0]}$也就是$X$，来初始化第一层的输入值。$a^{[0]}$对应于一个训练样本的输入特征，而$A^{[0]}$对应于一整个训练样本的输入特征，所以这就是这条链的第一个前向函数的输入，重复这个步骤就可以从左到右计算前向传播。

下面讲反向传播的步骤：输入为$da^{[l]}$，输出为$da^{[l-1]}$，$dw^{[l]}$, $db^{[l]}$




- 反向传播的步骤可以写成：

  [1] $dz^{[l]} = da^{[l]} * g^{[l]}{'}(z^{[l]})$

  [2] $dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$

  [3] $db^{[l]} = dz^{[l]}$

  [4] $da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$

  [5] $dz^{[l]} = w^{[l+1]T} dz^{[l+1]} \cdot\ g^{[l]}{'}(z^{[l]})$，由式[4]带入式[1]得到。

- 向量化实现过程可以写成：

  [1] $dZ^{[l]} = dA^{[l]} * g^{[l]}{'}\left(Z^{[l]}\right)$

  [2] $dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$

  [3] $db^{[l]} = \frac{1}{m}\ np.sum(dz^{[l]}, axis = 1, keepdims = True)$

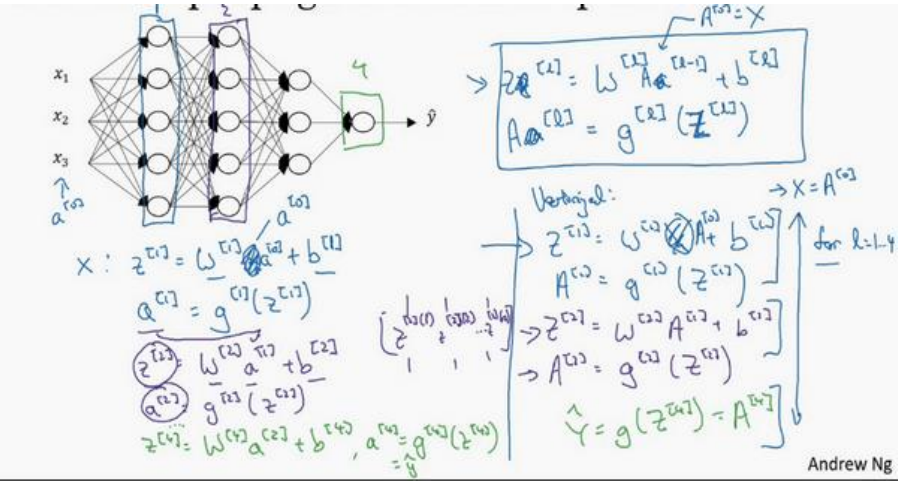
  [4] $dA^{[l-1]} = W^{[l]T}.dZ^{[l]}$

## 4.3 核对矩阵的维数

当实现深度神经网络时，我常用的检查代码是否有错的方法就是拿出一张纸过一遍算法中矩阵的维数。

- $w$的维度是（下一层的维数，前一层的维数），即$w^{[l]}: (n^{[l]},\ n^{[l-1]})$；
- $b$的维度是（下一层的维数，1），即$b^{[l]}: (n^{[l]},\ 1)$；
- $z^{[l]}$，$a^{[l]}$：$(n^{[l]}, 1)$；

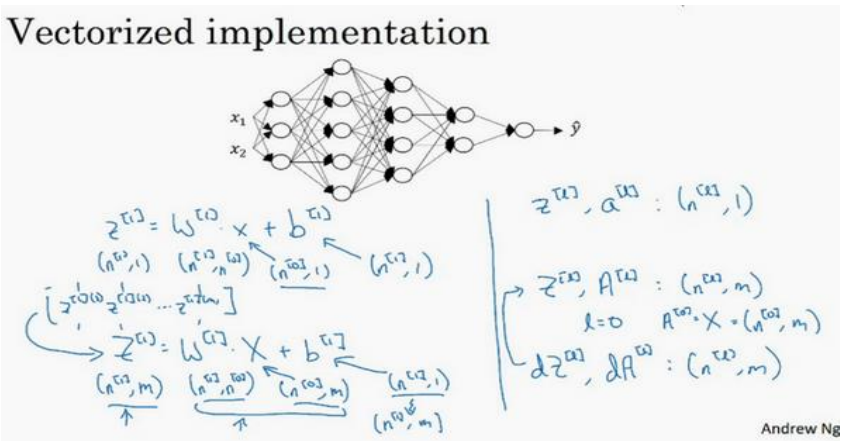
$dw^{[l]}$和$w^{[l]}$维度相同，$db^{[l]}$和$b^{[l]}$维度相同，且$w$和$b$向量化维度不变，但$z$，$a$以及$x$的维度会向量化后发生变化。




向量化后：

$Z^{[l]}$可以看成由每一个单独的$Z^{[l]}$叠加而得到，$Z^{[l]} = (z^{[l][1]}，z^{[l][2]}，z^{[l][3]}，\ldots，z^{[l][m]})$，$m$为训练集大小，所以$Z^{[l]}$的维度不再是$(n^{[l]}, 1)$，而是$(n^{[l]}, m)$。

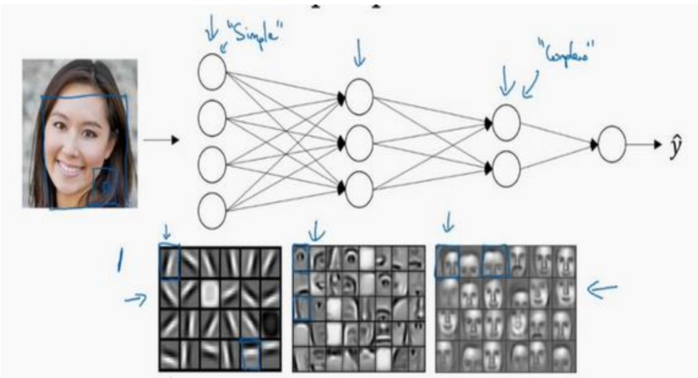
$A^{[l]}$：$(n^{[l]}, m)$，$A^{[0]} = X = (n^{[l]}, m)$




在你做深度神经网络的反向传播时，一定要确认所有的矩阵维数是前后一致的，可以大大提高代码通过率。

## 4.4 为什么使用深层表示？

我们都知道深度神经网络能解决好多问题，其实并不需要很大的神经网络，但是得有深度，得有比较多的隐藏层，这是为什么呢？我们一起来看几个例子来帮助理解，为什么深度神经网络会很好用。

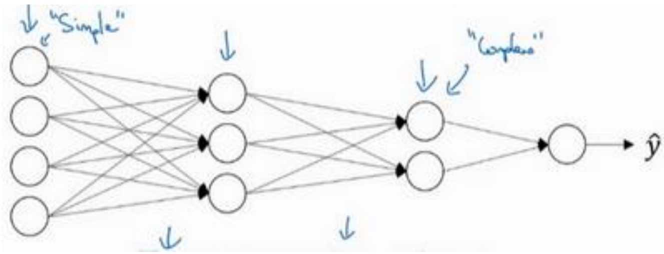



首先，深度网络究竟在计算什么？如果你在建一个人脸识别或是人脸检测系统，深度神经网络所做的就是，当你输入一张脸部的照片，然后你可以把深度神经网络的第一层，当成一个特征探测器或者边缘探测器。在这个例子里，我会建一个大概有 20 个隐藏单元的深度神经网络。隐藏单元就是这些图（第一张图）里这些小方块，举个例子，这个小方块（第一行第一列）就是一个隐藏单元，它会去找这张照片里“|”垂直方向。那么这个隐藏单元（第四行第四列），可能是在找（“—”）水平方向。你可以先把神经网络的第一层当作看图，然后去找这张照片的各个边缘。我们可以把照片里组成边缘的像素们放在一起看，然后它可以把被探测到的边缘组合成面部的不同部分（第二张图）。比如说，可能有一个神经元会去找眼睛的部分，另外还有别的在找鼻子的部分，然后把这许多的边缘结合在一起，就可以开始检测人脸的不同部分。最后再把这些部分放在一起，比如鼻子眼睛下巴，就可以识别或是探测不同的人脸（第三张图）。

你可以直觉上把这种神经网络的前几层当作探测简单的函数，比如边缘，之后把它们跟后几层结合在一起，那么总体上就能学习更多复杂的函数。这些图的意义，我们在学习卷积神经网络的时候再深入了解。还有一个技术性的细节需要理解的是，边缘探测器其实相对来说都是针对照片中非常小块的面积。就像这块（第一行第一列），都是很小的区域。面部探测器就会针对于大一些的区域，但是主要的概念是，一般你会从比较小的细节入手，比如边缘，然后再一步步到更大更复杂的区域，比如一只眼睛或是一个鼻子，再把眼睛鼻子装一块组成更复杂的部分。


这种从简单到复杂的金字塔状表示方法或者组成方法，也可以应用在图像或者人脸识别以外的其他数据上。比如当你想要建一个语音识别系统的时候，需要解决的就是如何可视化语音，比如你输入一个音频片段，那么神经网络的第一层可能就会去先开始试着探测比较低

层次的音频波形的一些特征，比如音调是变高了还是低了，分辨白噪音，咝咝咝的声音，或者音调，可以选择这些相对程度比较低的波形特征，然后把这些波形组合在一起就能去探测声音的基本单元。在语言学中有个概念叫做音位，比如说单词 ca，c 的发音，“嗑”就是一个音位，a 的发音“啊”是个音位，t 的发音“特”也是个音位，有了基本的声音单元以后，组合起来，你就能识别音频当中的单词，单词再组合起来就能识别词组，再到完整的句子。



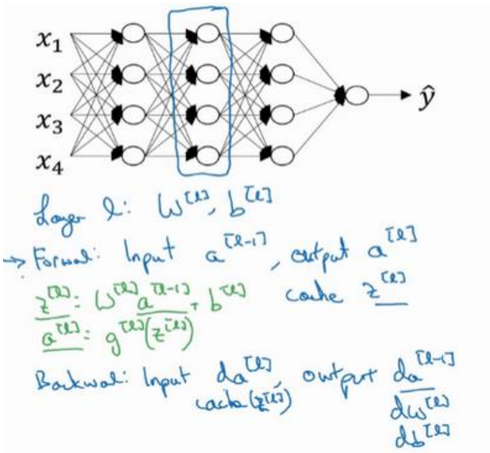

所以深度神经网络的这许多隐藏层中，较早的前几层能**学习一些低层次的简单特征，等到后几层，能把简单的特征结合起来，去探测更加复杂的东西。**比如你录在音频里的单词、词组或是句子，然后就能运行语音识别了。同时我们所计算的之前的几层，也就是相对简单的输入函数，比如图像单元的边缘什么的。到网络中的深层时，你实际上就能做很多复杂的事，比如探测面部或是探测单词、短语或是句子。

**Small**：隐藏单元的数量相对较少

**Deep**：隐藏层数目比较多

深层的网络隐藏单元数量相对较少，隐藏层数目较多，如果浅层的网络想要达到同样的计算结果则需要指数级增长的单元数量才能达到。

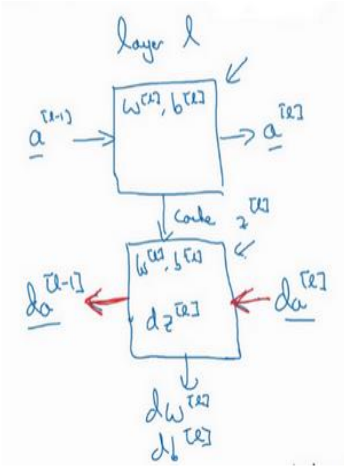
## 4.5 搭建神经网络块



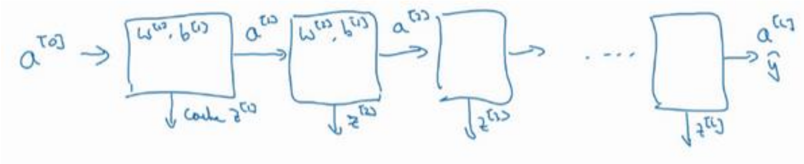

这是一个层数较少的神经网络，我们选择其中一层（方框部分），从这一层的计算着手。在第$l$层你有参数$W^{[l]}$和$b^{[l]}$，正向传播里有输入的激活函数，输入是前一层$a^{[l-1]}$，输出是$a^{[l]}$，我们之前讲过$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$,$a^{[l]} = g^{[l]}(z^{[l]})$，那么这就是你如何从输入$a^{[l-1]}$走到输出的$a^{[l]}$。之后你就可以把$z^{[l]}$的值缓存起来，我在这里也会把这包括在缓存中，因为缓存的$z^{[i]}$对以后的正向反向传播的步骤非常有用。

然后是反向步骤或者说反向传播步骤，同样也是第$l$层的计算，你会需要实现一个函数输入为$da^{[l]}$，输出$da^{[l-1]}$的函数。一个小细节需要注意，输入在这里其实是$da^{[l]}$以及所缓存的$z^{[l]}$值，之前计算好的$z^{[l]}$值，除了输出$da^{[l-1]}$的值以外，也需要输出你需要的梯度$dW^{[l]}$和$db^{[l]}$，这是为了实现梯度下降学习。

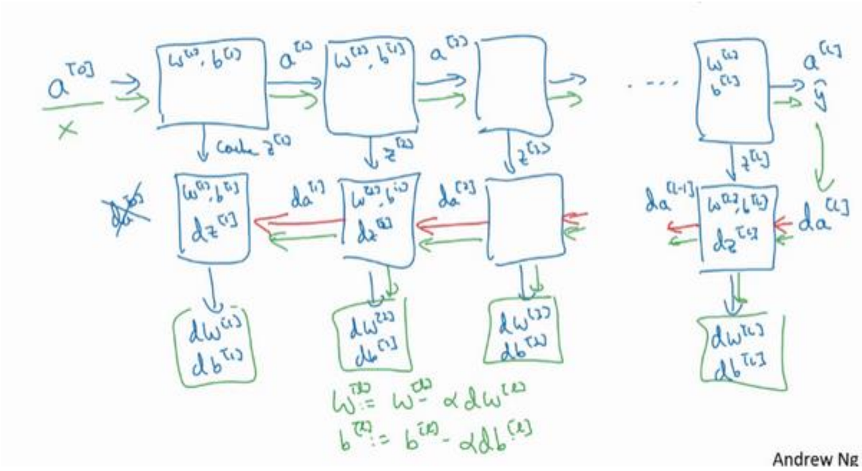
这就是基本的正向步骤的结构，我把它成为称为正向函数，类似的在反向步骤中会称为反向函数。总结起来就是，在 l 层，你会有正向函数，输入$a^{[l-1]}$并且输出$a^{[l]}$，为了计算结果你需要用$W^{[l]}$和$b^{[l]}$，以及输出到缓存的$z^{[l]}$。然后用作反向传播的反向函数，是另一个函数，输入$da^{[l]}$，输出$da^{[l-1]}$，你就会得到对激活函数的导数，也就是希望的导数值$da^{[l]}$。$a^{[l-1]}$是会变的，前一层算出的激活函数导数。在这个方块（第二个）里你需要$W^{[l]}$和$b^{[l]}$，最后你要算的是$dz^{[l]}$。然后这个方块（第三个）中，这个反向函数可以计算输出$dW^{[l]}$和$db^{[l]}$。我会用红色箭头标注标注反向步骤，如果你们喜欢，我可以把这些箭头涂成红色。



然后如果实现了这两个函数（正向和反向），然后神经网络的计算过程会是这样的：



把输入特征$a^{[0]}$，放入第一层并计算第一层的激活函数，用$a^{[1]}$表示，你需要$W^{[1]}$和$b^{[1]}$来计算，之后也缓存$z^{[l]}$值。之后喂到第二层，第二层里，需要用到$W^{[2]}$和$b^{[2]}$，你会需要计算第二层的激活函数$a^{[2]}$。后面几层以此类推，直到最后你算出了$a^{[L]}$，第$L$层的最终输出值$\hat{y}$。在这些过程里我们缓存了所有的$z$值，这就是正向传播的步骤。

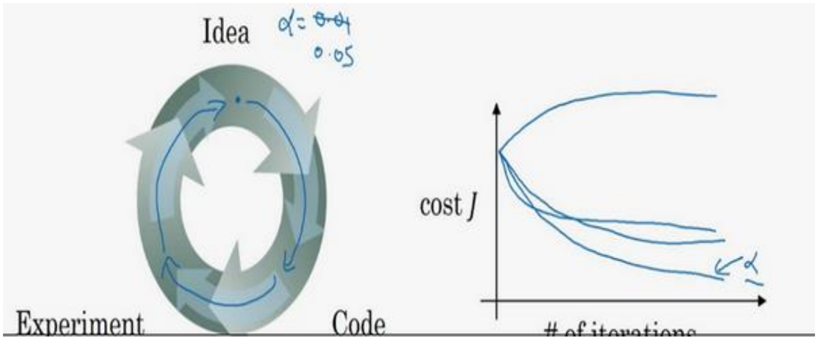



对反向传播的步骤而言，我们需要算一系列的反向迭代，就是这样反向计算梯度，你需要把$da^{[L]}$的值放在这里，然后这个方块会给我们$da^{[L-1]}$的值，以此类推，直到我们得到$da^{[2]}$和$da^{[1]}$，你还可以计算多一个输出值，就是$da^{[0]}$，但这其实是你的输入特征的导数，并不重要，起码对于训练监督学习的权重不算重要，你可以止步于此。反向传播步骤中也会输出$dW^{[l]}$和$db^{[l]}$，这会输出$dW^{[3]}$和$db^{[3]}$等等。目前为止你算好了所有需要的导数，稍微填一下这个流程图。

## 4.6 参数 VS 超参数

想要深度神经网络起很好的效果，还需要规划好参数以及超参数。什么是超参数？比如算法中的 **learning rate** $a$（学习率）、**iterations**(梯度下降法循环的数量)、$L$（隐藏层数目）、$n^{[l]}$（隐藏层单元数目）、**choice of activation function**（激活函数的选择）都需要你来设置，这些数字实际上控制最后的参数$W$和$b$的值，所以它们被称作超参数。

实际上深度学习有很多不同的超参数，之后我们也会介绍一些其他的超参数，如 **momentum**、**mini batch size**、**regularization parameters** 等等。如何寻找超参数的最优值？




走 **Idea—Code—Experiment—Idea** 这个循环，尝试各种不同的参数，实现模型并观察是否成功，然后再迭代。

今天的深度学习应用领域，还是很经验性的过程，比如你可能大致知道一个最好的学习率值，可能说$a = 0.01$最好，我会想先试试看，然后你可以实际试一下，训练一下看看效果如何。然后基于尝试的结果，你觉得学习率设定再提高到 0.05 会比较好。如果你不确定什么值是最好的，你大可以先试试一个学习率$a$，再看看损失函数 J 的值有没有下降。然后你可以试一试大一些的值，发现损失函数的值增加并发散了，可能试试其他数，看结果是否下降的很快或者收敛到在更高的位置。

然而，当你开始开发新应用时，预先很难确切知道，究竟超参数的最优值应该是什么。所以通常必须尝试很多不同的值，并走这个循环，试试各种参数。试试看 5 个隐藏层，这个数目的隐藏单元，实现模型并观察是否成功，然后再迭代。一个很大程度基于经验的过程，凭经验的过程就是试直到你找到合适的数值。

另一个近来深度学习的影响是它用于解决很多问题，从计算机视觉到语音识别，到自然语言处理，到很多结构化的数据应用，比如网络广告或是网页搜索或产品推荐等等，这些领域中的一个，尝试了不同的设置，有时候这种设置超参数的直觉可以推广，但有时又不会。所以建议人们，特别是刚开始应用于新问题的人们，去试一定范围的值看看结果如何。下一门课程，我们会用系统性的尝试各种超参数取值。然后，甚至是你已经用了很久的模型，可能你在做网络广告应用，在你开发途中，很**有可能学习率的最优数值或是其他超参数的最优值是会变的**，所以即使你每天都在用当前最优的参数调试你的系统，你还是会发现，最优值过一年就会变化，因为电脑的基础设施，**CPU** 或是 **GPU** 可能会变化很大。