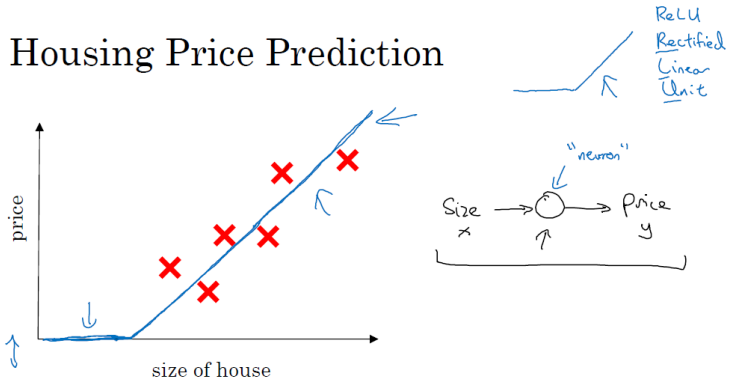
神经网络基础

介绍

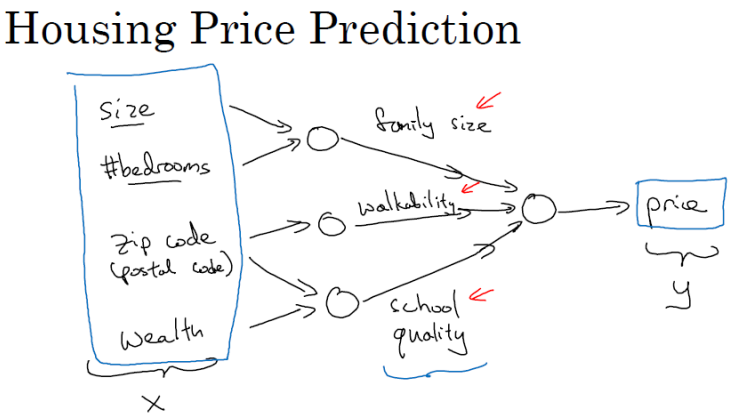
直观感受

深度学习指的是特别大规模的神经网络训练

这是一个以预测房价的例子:



如果以直线拟合，当然房价不可能为负数；所以出现了图中函数：ReLu激活函数；(可以理解为max(0,x)); 图中的蓝色标记的神经元就是激活函数ReLU激活函数的实现；这可以看成是一个简单的神经网络；下面我们看一个复杂的神经网络:



此例中输入值为四个，四个输入间接决定了一些影响房子价格的“直接因素”；形成了图中的复杂神经网络，其实可看成是单个神经网络堆叠形成的；

神经网络&监督学习

监督学习（Supervised Learning）

监督学习指的是我们已经知道输入和正确的输出是什么，我们要寻找的是输入和输出之间的关系；（用已知数据去发现数据间的规律）

- 回归（regression）：trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function
- 分类（classification）：trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories

神经网络在监督学习场景中的应用见下图:

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

当然机器学习也应用于结构化数据和非结构化的数据：

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Unstructured Data

Audio Image

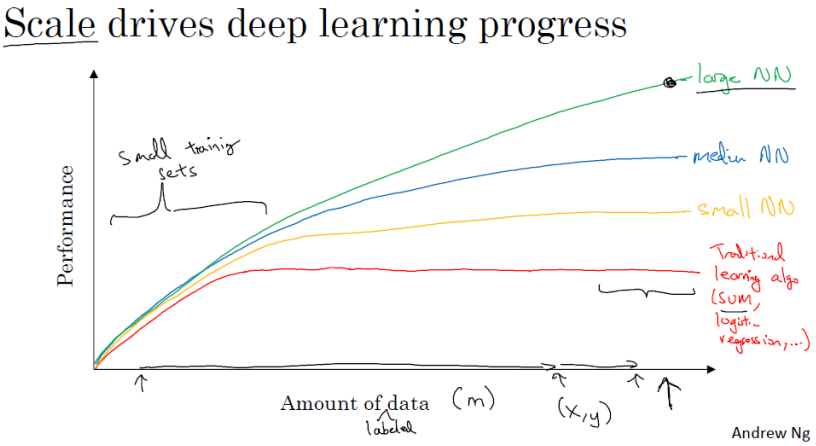
Four scores and seven years ago...

Text

神经网络的兴起

数据规模、计算量和算法的创新推动了深度学习的发展

下图是各种算法性能在数据集上的比较：

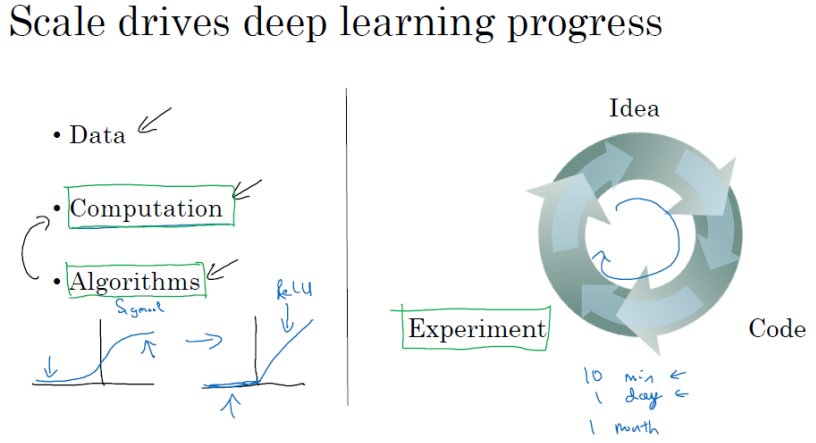


在数据集规模较小的应用上，传统算法与神经网络所表现的性能可能相差不大，甚至会优于神经网络；但是随着数据规模的增大，神经网络所表现出的性能会远远优于其他算法，至少目前来看是这样的，不知道以后还会不会有更好地算法；

为了提高神经网络在大数据上的性能：

- Being able to train a big enough neural network
- Huge amount of labeled data

训练神经网络的过程是迭代的：



我们需要花费大量的时间来训练神经网络，以提高我们的生产力；当然，更快的计算帮助我们迭代和使用更优的算法

神经网络编程基础

- 实现一个神经网络时，如果需要遍历整个训练集，并不需要直接使用 for 循环
- 神经网络的计算过程中，通常有一个正向过程（forward pass）或者叫正向传播步骤（forward propagation step），接着会有一个反向过程（backward pass）或者叫反向传播步骤（backward propagation step）

符号约定

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

· superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

·m : number of examples in the dataset

·n_x : input size

·n_y : output size (or number of classes)

·$n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h{}^{[\text{number of layers }+1]}$.

·L : number of layers in the network.

Objects:

·$X \in \mathbb{R}^{n_x \times m}$ is the input matrix

·$x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector

·$Y \in \mathbb{R}^{n_y \times m}$ is the label matrix

·$y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example

·$W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer

·$b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer

·$\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

Common forward propagation equation examples:

$a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function

$\hat{y}^{(i)} = softmax(W_h h + b_2)$

· General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$

· $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

Examples of cost function:

· $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$

· $J_1(\hat{y}, y) = \sum_{i=0}^{m} | \, y^{(i)} - \hat{y}^{(i)} \, |$

2 Deep Learning representations

For representations:

· nodes represent inputs, activations or outputs

· edges represent weights or biases

Here are several examples of Standard deep learning representations

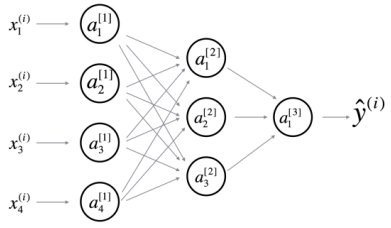


Figure 1: Comprehensive Network: representation commonly used for Neural Networks. For better aesthetic, we omitted the details on the parameters ($w_{ij}^{[l]}$ and $b_i^{[l]}$ etc...) that should appear on the edges

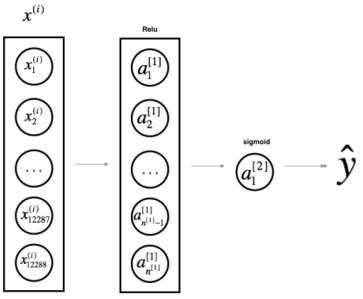
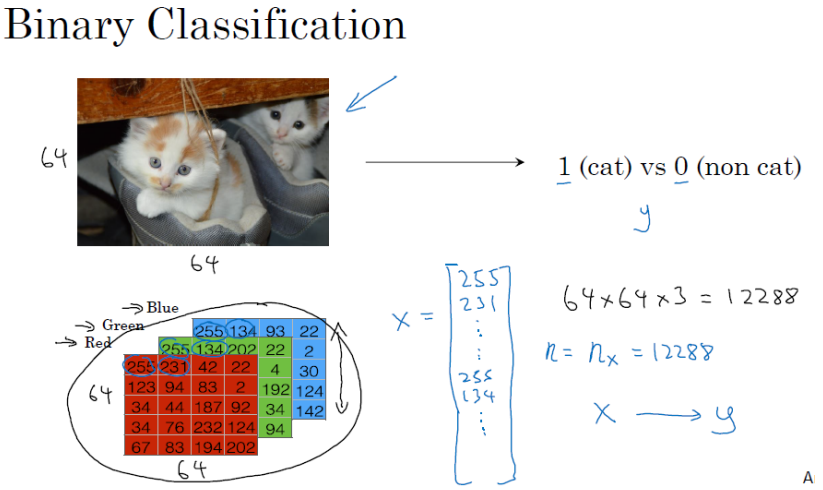


Figure 2: Simplified Network: a simpler representation of a two layer neural network, both are equivalent.

二分类

In a binary classification problem, the result is a discrete value output.

给出一些输入，输入结果是离散量；下面是一个用逻辑回归实现的猫分类器：



一张图片作为输入，输出结果用y表示；y=1表示是猫，y=0表示非猫；计算机为了表示输入的图片，需要保存三个矩阵，对应红黄蓝三种颜色通道；如果图片大小为64*64像素，则需要保存三个64*64的矩阵，分别对应图中红黄蓝三种颜色的强度；为了简单表示，我们将输入提取为一个64*64*3的一个特征向量x，如图所示，表示输入的图片；我们的目标就是学习一个分类器，预测输入的图片是否是猫。当然，实际中我们也不这样干，太费劲了，此处只为说明问题

逻辑回归

Logistic Regression 是一个常用于二分类的算法

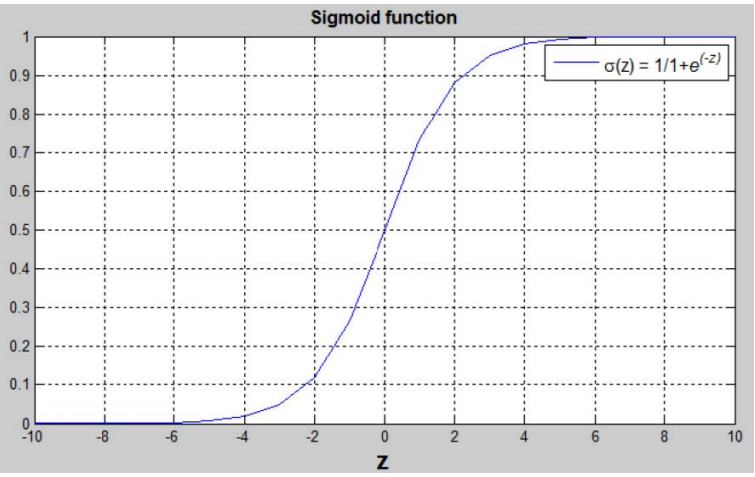
Logistic regression is a learning algorithm used in a supervised learning problem when the output y are all either zero or one. The goal of logistic regression is to minimize the error between its predictions and training data.

逻辑回归（Logistic Regression，也译作“对数几率回归”）是离散选择法模型之一，属于多重变量分析范畴，是社会学、生物统计学、临床、数量心理学、计量经济学、市场营销等统计实证分析的常用方法。

参数表示

- 输入的特征向量： $x \in \mathbb{R}^{n_x}$; n_x 是特征数量
- 用于训练的标签： $y \in 0, 1$
- 权重： $w \in \mathbb{R}^{n_x}$
- 偏置： $b \in \mathbb{R}$

- 输出：$\hat{y} = \sigma(w^T x + b)$
- Sigmoid函数：$s = \sigma(w^T x + b) = \sigma(z) = \frac{1}{1+e^{-z}}$



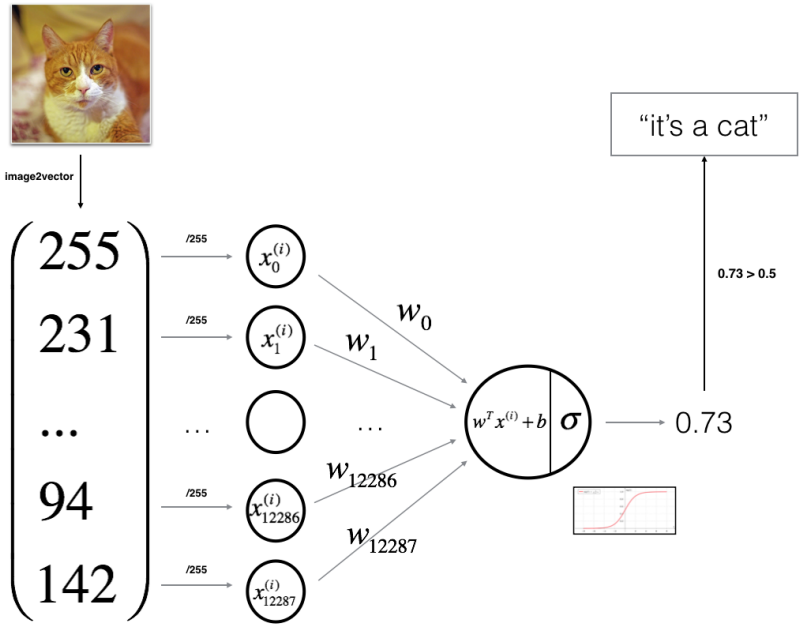
由图可知，Sigmoid函数有着下面的性质：

- 当 z 趋向于 无穷大时，$\sigma(z) = 1$
- 当 z 趋向于 负无穷大时，$\sigma(z) = 0$
- 当 z = 0 时，$\sigma(z) = 0.5$

因此，猫分类器要实现的就是：给定一个 n_x 维特征的输入特征向量 x, 标签为 y 的一张图片，估计这张图片为猫图的概率 $\hat{y}$, 即：

$$\hat{y} = P(y = 1|x), 0 \leq \hat{y} \leq 1$$

我们要做的就是根据输入特征向量 X 和标签数据 y，寻找一个函数，来表示出 $\hat{y}$；如果直接使用线性拟合 $\hat{y} = w^T X + b$的话，得到的 $\hat{y}$ 可能非常大，但实际是我们只希望 $\hat{y}$ 在 0 到 1 之间, 所以我们考虑使用 Sigmoid 函数对线性预测值进行约束，将其预测值限制到 0 到 1 之间，此为逻辑回归
下面这张图表示了整个过程：



代价（成本）函数

我们上文提到过，Logistic回归的目的是要最小化预测值和真实值的误差，怎么定义这个误差，即为代价函数；此外，代价函数还为我们提供了一个计算模型中参数 w 和 b 的一个手段，我们可以通过训练代价函数来得到参数值

损失（误差）函数：：

- 单个训练样本的预测值和真实值之间的误差 $L(\hat{y}, y)$

- 代价函数衡量的是全体样本的表现，可看成是所有损失函数的和：$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})$
- 常用的损失函数如平方误差：$L(\hat{y},y)=\frac{1}{2}(\hat{y}-y)^2$

在逻辑回归里面将不适用平方误差，因为这个损失函数在训练的过程中会得到一个非凸函数，最终会存在很多局部最优解，梯度下降可能找不到全局最优

在这里，我们考虑的是满足以下条件概率：$p(y|x)=\begin{cases}\hat{y} & \text{y=1}\\ 1-\hat{y} & \text{y=0}\end{cases}$
合并成一个式子如下：$p(y|x)=\hat{y}^{y}(1-\hat{y})^{(1-y)}$
对两边求对数，化简如下：$logp(y|x)=ylog(\hat{y})+(1-y)log(1-\hat{y})$
因为 log(x) 函数是严格单调递增的，我们要最大化 p(y|x)，同时还要最小化损失函数，所以在上式添加一个负号即可得到应用很广的**交叉熵**损失函数：

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

- 当 y = 1 时，$L=-log(\hat{y})$, 如果要让 L 尽可能小，那么 $\hat{y}$ 就要尽可能大，Sigmoid函数使得 $\hat{y}$ 就要尽可能接近于 1
- 当 y = 0 时，$L=-log(1-\hat{y})$, 如果要让 L 尽可能小，那么 $\hat{y}$ 就要尽可能小，Sigmoid函数使得 $\hat{y}$ 就要尽可能接近于 0

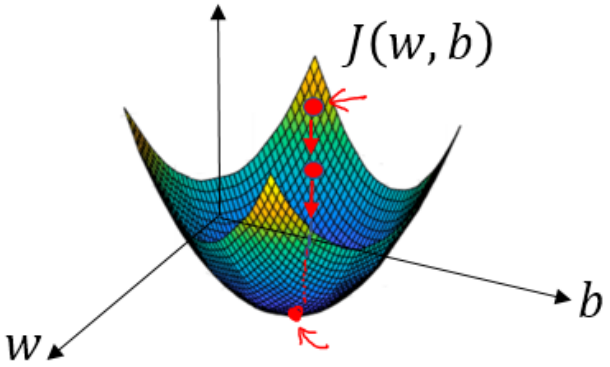
以上为单个样本的损失函数表达式，我们根据**极大似然估计（Maximum Likelihood Estimation）**推导 m 个样本的代价函数
假设所有样本独立同分布，则联合概率为所有样本的概率：$P=\prod_{i=1}^{m}p(y^{(i)}|x^{(i)})$
对上式进行最大似然估计，即两边取对数得到：$logP=\sum_{i=1}^{m}logp(y^{(i)}|x^{(i)})=-\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})$
去上次再取平均值，得到代价函数为：

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

梯度下降

函数的梯度（gradient）指出了函数变化最快的方向。即是说，按梯度的方向走，函数增长得就越快。那么按梯度的负方向走，函数值自然就降低得最快了，我们的目标是寻找合适的 w 和 b 以最小化代价函数；

简单起见，假设 w 和 b 都是一维实数，代价函数 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，如下图所示：

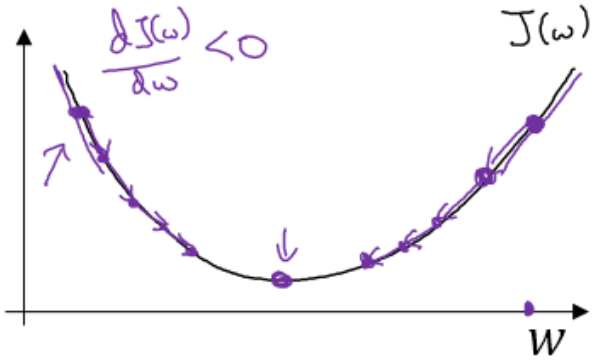


可以看到，代价函数为凸函数，与非凸函数的区别在于其不含多个局部最优解；为了训练找到合适的 w 和 b 的值，我们选取图中的小红点为 w 和 b 初始值所在点，当然也可以随机初始化，因为逻辑回归代价函数为凸函数的特性，保证了无论初始值在哪，都能到达同一个点或者大致相同的点。
梯度下降从起始点开始，**沿着最陡峭的下坡方向走，不断迭代**，最终到达全局最优解或接近全局最优解的地方。

$$\begin{cases}w:=w-\alpha\frac{dJ(w,b)}{dw}\\ b:=b-\alpha\frac{dJ(w,b)}{db}\end{cases}$$

- α 表示学习速度，即每次更新参数的步长
- α 不宜过大，也不宜过小；过小的话迭代次数会很多，导致训练时间过长，过大的话可能导致错过最优解

下图是一个参数的梯度下降细节说明：

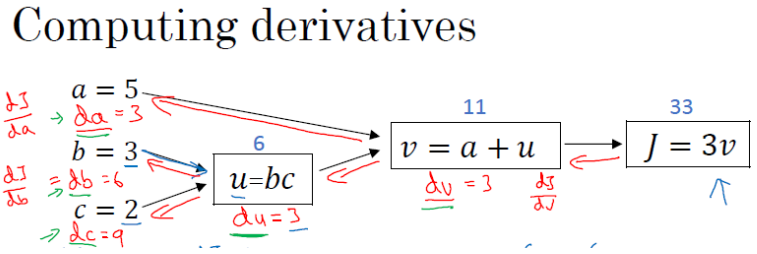


- 沿着导数的方向下降速度最快
- 当 w 大于最优解 w' 时，导数大于 0，那么 w 就会向更小的方向更新，即图中向左直逼最小值点
- 当 w 小于最优解 w' 时，导数小于 0，那么 w 就会向更大的方向更新，即图中向右直逼最小值点

计算图

一个神经网络的计算，都是按照前向或反向传播过程组织的。首先我们计算出一个新的网络的输出（前向过程），紧接着进行一个反向传输操作，。后者我们用来计算出对应的梯度或导数，即是当我们需要计算最终值相对于某个特征变量的导数时，我们需要利用计算图中上一步的结点定义。计算图解释了为什么我们用这种方式组织这些计算过程。

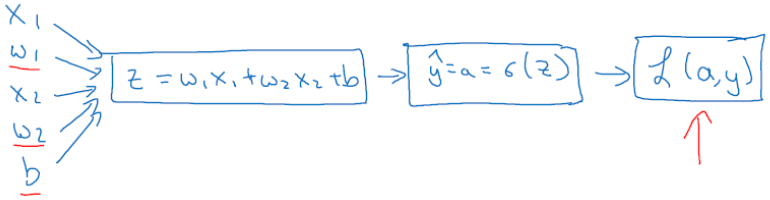
下面是一个计算图计算导数的例子：形象地说明了神经网络计算的过程：正向传播计算代价函数，反向传播计算导数，用到了求导的链式法则



逻辑回归的梯度下降

假设输入的特征向量维度为 2，即输入参数共有 x1, w1, x2, w2, b 这五个。可以推导出如下的计算图：

$$z = w^T x + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$



首先反向求出 L 对于 a 的导数：

$$da = \frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

接着继续反向求出 L 对于 z 的导数：

$$dz = \frac{dL(a,y)}{dz} = \frac{dL}{da}\frac{da}{dz} = a - y$$

依此类推求出最终的损失函数相较于原始参数的导数：

$$\begin{cases} dw1 = \frac{dL}{dw1} = x1dz = x1(a-y) \\ dw2 = \frac{dL}{dw2} = x2dz = x2(a-y) \\ db = dz = a - y \end{cases}$$

根据如下公式进行参数更新：

$$\begin{cases} w1 := w1 - \alpha dw1 \\ w2 := w2 - \alpha dw2 \\ b := b - \alpha db \end{cases}$$

接下来我们需要将对于单个样本的损失函数扩展到整个训练集的代价函数：

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

根据上面单个样本的求导计算，很容易求得整体的导数计算，只需要加上上标 (i) 即可
下图是 m 个样本的逻辑回归梯度下降：

$J = 0,\ dw1 = 0,\ dw2 = 0,\ db = 0$
for i = 1 to m:
$z^{(i)} = w^T x^{(i)} + b$
$a^{(i)} = \sigma(z^{(i)})$
$J \mathrel{+}= -\left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right]$
$dz^{(i)} = a^{(i)}(1 - a^{(i)})$
$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$
$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$
$db \mathrel{+}= dz^{(i)}$
$J = J/m,\ dw_1 = dw_1/m,\ dw_2 = dw_2/m,\ db = db/m$

我们求得了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，即我们要对参数求平均

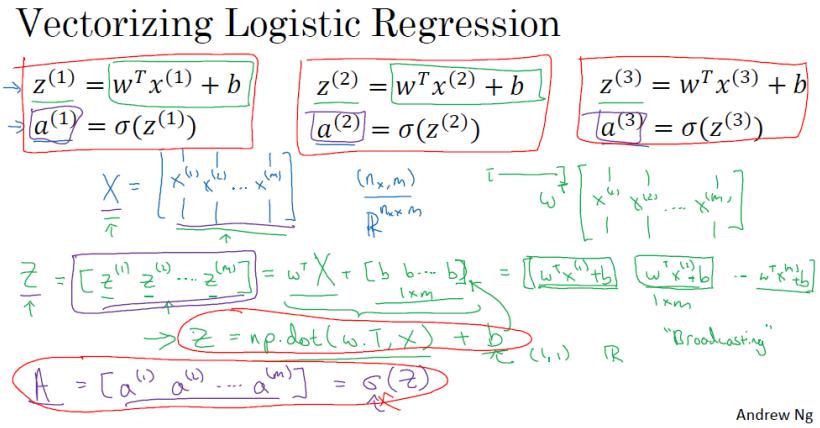
- 上述过程在计算时有一个缺点：你需要编写两个 for 循环。第一个 for 循环遍历 m 个样本，而第二个 for 循环遍历所有特征。如果有大量特征，在代码中显式使用 for 循环会使算法很低效。**向量化**可以用于解决显式使用 for 循环的问题。

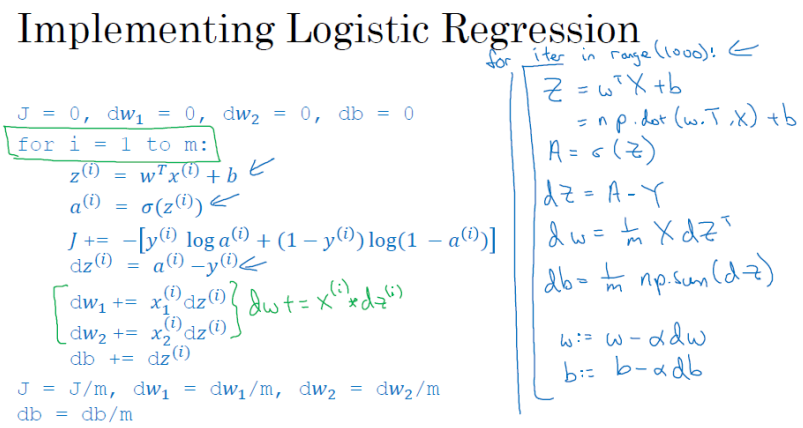
向量化

向量化是非常基础的去除代码中for循环的艺术

如计算 $z = w^T x + b$直接用 $z = np.dot(w.T, x) + b$而可以不用 for 循环

向量化逻辑回归





说白了，向量化即是将多步相同的操作整合到一步，变成一个向量同步去处理，去掉了显式的 for 循环。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。

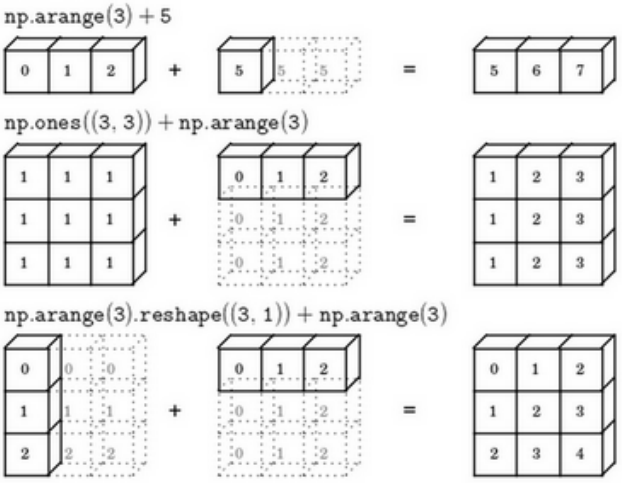
广播

Numpy 的 Universal functions 中要求输入的数组 shape 是一致的。当数组的 shape 不相等的时候，则会使用广播机制，调整数组使得 shape 一样，满足规则，则可以运算，否则就出错。

四条规则：

- 让所有输入数组都向其中 shape 最长的数组看齐，shape 中不足的部分都通过在前面加 1 补齐；
- 输出数组的 shape 是输入数组 shape 的各个轴上的最大值；
- 如果输入数组的某个轴和输出数组的对应轴的长度相同或者其长度为 1 时，这个数组能够用来计算，否则出错；
- 当输入数组的某个轴的长度为 1 时，沿着此轴运算时都用此轴上的第一组值。

用一张图来说明：



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。即同类型低维向高维填充

Numpy使用技巧

转置对秩为 1 的数组无效。因此，应该避免使用秩为 1 的数组，用 n * 1 的矩阵代替。例如，用 np.random.randn(5,1) 代替 np.random.randn(5)。