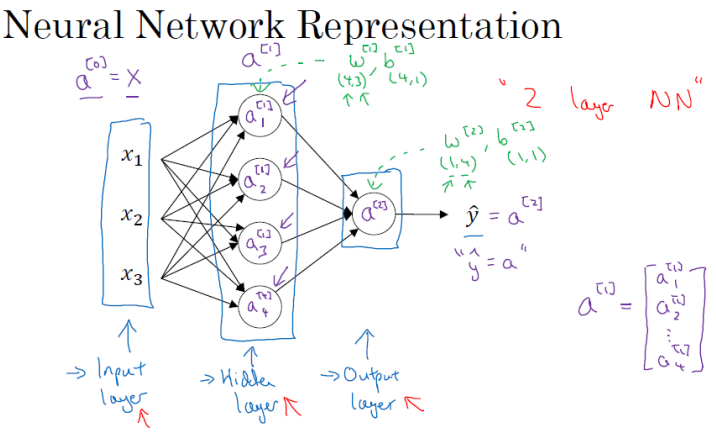
1 神经网络

1.1 浅层神经网络

1.1.1 神经网络的表示

- 竖向堆叠起来的输入特征被称作神经网络的输入层（the input layer）。
- 神经网络的隐藏层（a hidden layer）。“隐藏”的含义是在训练集中，这些中间节点的真正数值是无法看到的。
- 输出层（the output layer）负责输出预测值。

如下图是一个双层神经网络，也称作单隐层神经网络：



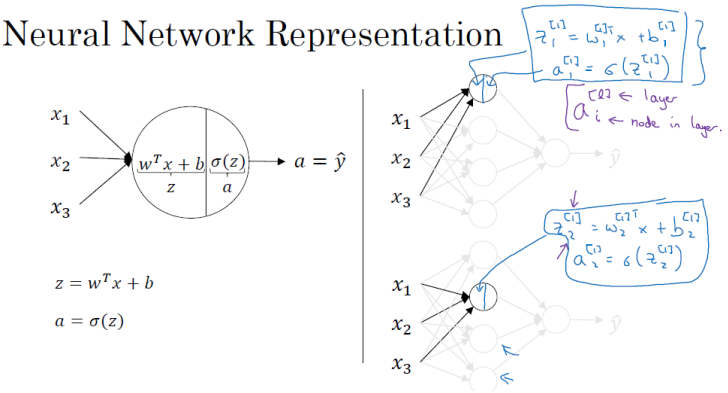
当我们计算网络的层数时，通常不考虑输入层，因此图中隐藏层是第一层，输出层是第二层。

约定俗成的符号表示是：

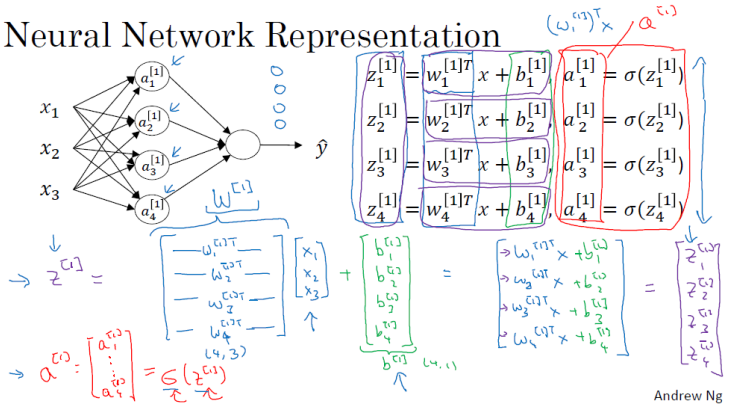
- 输入层 X 的激活值为 $a^{[0]}$ [] 里面的数字代表第几层，a代表激活
- 隐藏层和以及最后的输出层都是带参数 w 和 b 的，我们也以类似的上标 [] 方式表示与哪一层有关

1.1.2 神经网络的计算

实际上，神经网络的计算就是将 Logistic 回归的计算步骤重复很多次



详细计算过程如下图：



对于单个样本进行向量化之后可以得到：

$$\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = \overbrace{\begin{bmatrix} \dots W_1^{[1]T} \dots \\ \dots W_2^{[1]T} \dots \\ \dots W_3^{[1]T} \dots \\ \dots W_4^{[1]T} \dots \end{bmatrix}}^{W^{[1]}} * \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}^{input} + \overbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}^{b^{[1]}} \tag{1}$$

然后通过如下激活函数进行传递计算：

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]}) \tag{2}$$

即第一个隐藏层我们有如下公式：

$$\begin{cases} z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]} \\ a_1^{[1]} = \sigma(z_1^{[1]}) \end{cases} \tag{3}$$

以后的以此类推

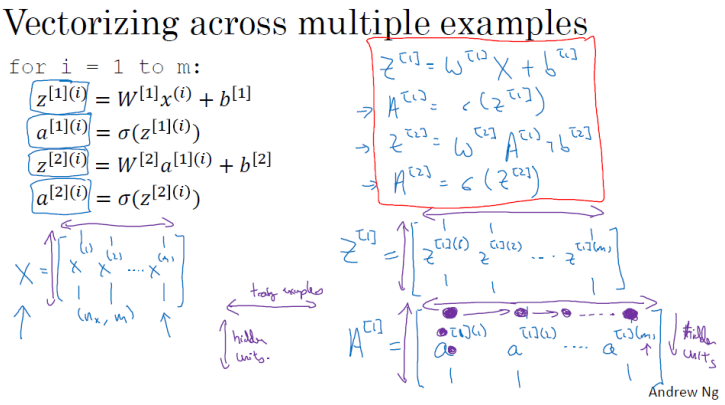
更一般的，有如下规律：

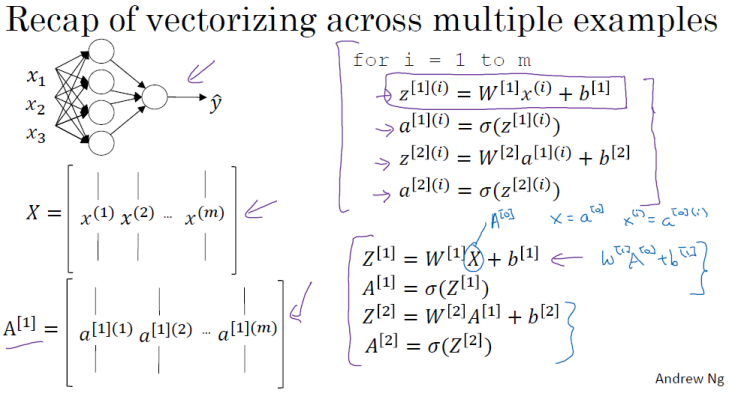
- $z^{[n]} = w^{[n]} x + b^{[n]}$
- $a^{[n]} = \sigma(z^{[n]})$
- 在一个的共有 L 层，且第 L 层有 $n^{[l]}$ 个节点的神经网络中，参数矩阵 $W^{[l]}$ 的大小为 $n^{[l]} * n^{[l-1]}$，$B^{[l]}$的大小为 $n^{[l]} * 1$

对于 m 个样本进行向量化即是加上上标 (i)

$$\left.\begin{aligned} z^{[1](i)} &= W^{[1](i)} x^{(i)} + b^{[1]} \\ a^{[1](i)} &= \sigma(z^{[1](i)}) \\ z^{[2](i)} &= W^{[2](i)} a^{[1](i)} + b^{[2]} \\ a^{[2](i)} &= \sigma(z^{[2](i)}) \end{aligned}\right\} \Longrightarrow \begin{cases} A^{[1]} = \sigma(z^{[1]}) \\ z^{[2]} = W^{[2]} A^{[1]} + b^{[2]} \\ A^{[2]} = \sigma(z^{[2]}) \end{cases} \tag{4}$$

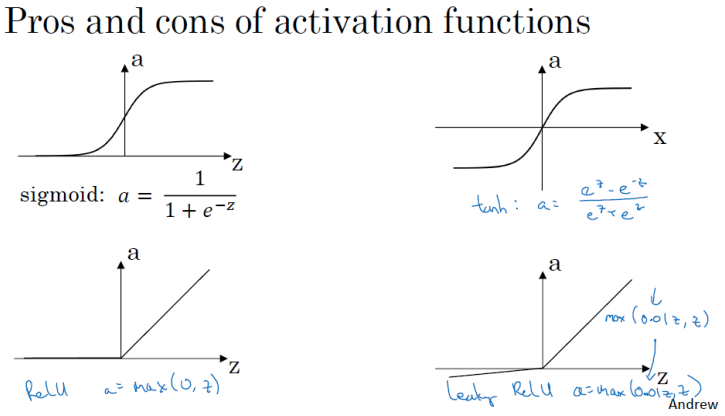
下图详细地解释了 m 个样本的向量化过程：





1.1.3 激活函数

使用一个神经网络时，需要决定使用哪种激活函数用在隐藏层上，哪种用在输出节点上。先前我们用sigmoid激活函数，但是，有时其他的激活函数效果会更好 下面是一些常见的激活函数选择：



- tanh 函数其实是 sigmoid 函数的移位版本。对于隐藏单元，选用 tanh 函数作为激活函数的话，效果总比 sigmoid 函数好，因为 tanh 函数的值在 -1 到 1 之间，最后输出的结果的平均值更趋近于 0，而不是采用 sigmoid 函数时的 0.5，这实际上可以使得下一层的学习变得更加轻松。对于二分类问题，为确保输出在 0 到 1 之间，将仍然采用 sigmiod 函数作为输出的激活函数
- sigmoid 函数和 tanh 函数都具有的缺点是：在 z 接近无穷大或无穷小时，这两个函数的导数也就是梯度变得非常小，此时梯度下降的速度也会变得非常慢
- ReLU 函数，当 z > 0 时，ReLU函数的导数一直为 1，所以采用 ReLU 函数作为激活函数时，随机梯度下降的收敛速度会比 sigmoid 及 tanh 快得多；然而当 z < 0 时，梯度一直为 0，但是实际的运用中，该缺陷的影响不是很大。当 z = 0 时，导数值没定义，但是也不影响，取 0 或者 1 都可以
- Leaky ReLU 保证在 z < 0 的时候，梯度仍然不为 0（通常取 0.01）。理论上来说，Leaky ReLU 有 ReLU 的所有优点，但在实际操作中没有证明总是好于 ReLU，因此不常用

激活函数的导数

- sigmoid 激活函数：$a = \sigma(z) = \frac{1}{1+e^{-z}}$

$$\frac{d}{dz}g(z) = \frac{1}{1+e^{-z}}(1-\frac{1}{1+e^{-z}}) = g(z)(1-g(z)) \tag{5}$$

在神经网络中：

$$a = g(z); g(z)' = \frac{d}{dz}g(z) = a(1-a) \tag{6}$$

- tanh 激活函数：$a = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

$$\frac{d}{dz}g(z) = 1 - (tanh(z))^2 \tag{7}$$

- ReLu 激活函数：$a = max(0, z)$

$$g(z)' = \begin{cases} 0 & \text{if z < 0} \\ 1 & \text{if z > 0} \\ undefined & \text{if z = 0} \end{cases} \tag{8}$$

- Leaky ReLu 激活函数：$a = max(0.01z, z)$

$$g(z)' = \begin{cases} 0.01 & \text{if z < 0} \\ 1 & \text{if z > 0} \\ undefined & \text{if z = 0} \end{cases} \tag{9}$$

指导意见：

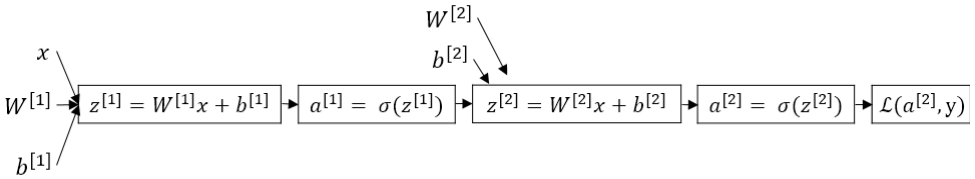
- sigmoid 激活函数：除了输出层是一个二分类问题基本不会用它
- tanh 激活函数：tanh 是非常优秀的，几乎适合所有场合
- ReLu 激活函数：最常用的默认函数；
- 如果不确定用哪个激活函数，就使用 ReLu 或者 Leaky ReLu，虽然很少使用 Leaky ReLU
- 没有固定答案，要依据实际问题在交叉验证集合中进行验证分析
- 我们可以在不同层选用不同的激活函数

非线性激活函数

使用线性的激活函数时，输出结果将是输入的线性组合，这样的话使用神经网络与直接使用线性模型的效果相当，此时神经网络就类似于一个简单的逻辑回归模型，输出都是输入的线性组合，失去了其本身的优势和价值。

1.1.4 神经网络的梯度下降

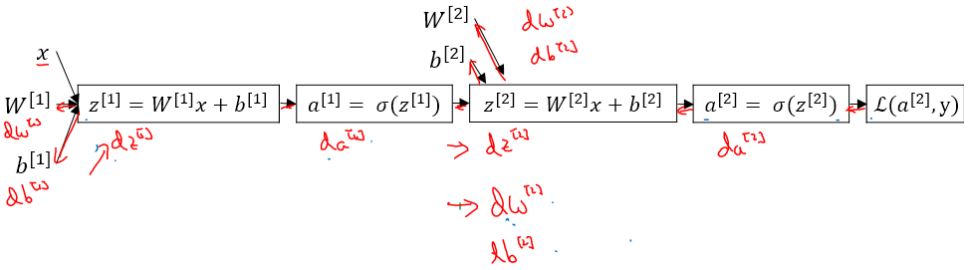
正向传播：



如图，通过输入样本x及参数 $w^{[1]}$、$b^{[1]}$ 到隐藏层，求得 $z^{[1]}$，进而求得 $a^{[1]}$；再将参数 $w^{[2]}$、$b^{[2]}$ 和 $a^{[1]}$ 一起输入输出层求得 $z^{[2]}$，进而求得 $a^{[2]}$，最后得到损失函数 $L(a^{[2]}, y)$，这样一个从前往后递进传播的过程，就称为前向传播（Forward Propagation）。

$$\begin{aligned} z^{[1]} &= w^{[1]T}X + b^{[1]} \\ a^{[1]} &= g(z^{[1]}) \\ z^{[2]} &= w^{[2]T}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) = sigmoid(z^{[2]}) \\ \mathcal{L}(a^{[2]}, y) &= -(y log\ a^{[2]} + (1-y)log(1-a^{[2]})) \end{aligned} \tag{10}$$

反向传播：



在训练过程中，经过前向传播后得到的最终结果跟训练样本的真实值总是存在一定误差，这个误差便是损失函数。想要减小这个误差，当前应用最广的一个算法便是梯度下降，于是用损失函数，从后往前，依次求各个参数的偏导，这就是所谓的反向传播（Back Propagation），一般简称这种算法为BP算法。下图展示了反向传播的整个推导过程：

$$
\begin{aligned}
da^{[2]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \\
dz^{[2]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} = a^{[2]} - y \\
dw^{[2]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}} = dz^{[2]} \cdot a^{[1]T} \\
db^{[2]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}} = dz^{[2]} \\
da^{[1]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} = dz^{[2]} \cdot w^{[2]} \\
dz^{[1]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} = dz^{[2]} \cdot w^{[2]} \times g^{[1]\prime}(z^{[1]}) \\
dw^{[1]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}} = dz^{[1]} \cdot X^T \\
db^{[1]} &= \frac{\partial \mathcal{L}(a^{[2]}, y)}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}} = dz^{[1]}
\end{aligned}
\tag{11}
$$

在具体的算法实现过程中，还是需要采用逻辑回归中用到梯度下降的方法，将各个参数进行向量化、取平均值，不断进行更新。

下面是反向梯度下降公式以及代码比较：

Summary of gradient descent

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

1.1.5 随机初始化

权重随机初始化是很重要的。如果在初始时将两个隐藏神经元的参数设置为相同的大小，那么两个隐藏神经元对输出单元的影响也是相同的，通过反向梯度下降去进行计算的时候，会得到同样的梯度大小，所以在经过多次迭代后，两个隐藏层单位仍然是对称的。无论设置多少个隐藏单元，其最终的影响都是相同的，那么多个隐藏神经元就没有了意义。

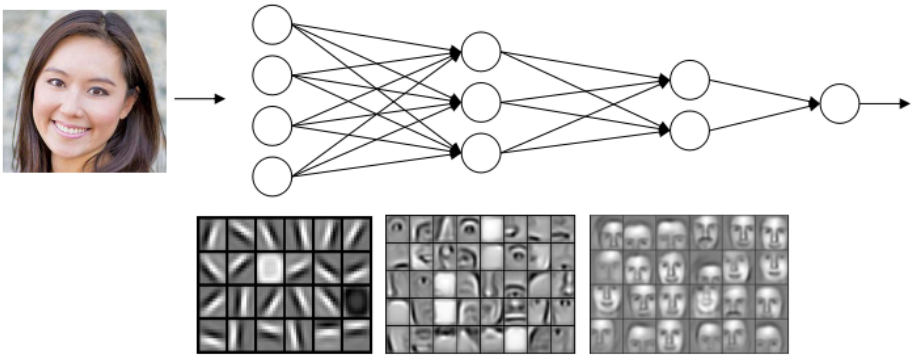
在初始化的时候，W 参数要进行随机初始化，不可以设置为 0。而 b 因为不存在对称性的问题，可以设置为 0。
以 2 个输入，2 个隐藏神经元为例：
$W^{[1]} = np.random.randn(2,2) * 0.01$, $b^{[1]} = np.zeros((2,1))$
$W^{[2]} = np.random.randn(2,2) * 0.01$, $b^{[2]} = 0$

这里将 W 的值乘以 0.01（或者其他的常数值）的原因是为了使得权重 W 初始化为较小的值，这是因为使用 sigmoid 函数或者 tanh 函数作为激活函数时，W 比较小，则 Z=WX+b 所得的值趋近于 0，梯度较大，能够提高算法的更新速度。而如果 W 设置的太大的话，得到的梯度较小，训练过程因此会变得很慢。
ReLU 和 Leaky ReLU 作为激活函数时不存在这种问题，因为在大于 0 的时候，梯度均为 1。

1.2 深层神经网络

深层神经网络含有多个隐藏层，构建方法如前面所述，训练时根据实际情况选择激活函数，进行前向传播获得成本函数进而采用 BP 算法，进行反向传播，梯度下降缩小损失值。

拥有多个隐藏层的深层神经网络能更好得解决一些问题。如图，例如利用神经网络建立一个人脸识别系统，输入一张人脸照片，深度神经网络的第一层可以是一个特征探测器，它负责寻找照片里的边缘方向，卷积神经网络（Convolutional Neural Networks，CNN）专门用来做这种识别。



深层神经网络的第二层可以去探测照片中组成面部的各个特征部分，之后一层可以根据前面获得的特征识别不同的脸型的等等。这样就可以将这个深层神经网络的前几层当做几个简单的探测函数，之后将这几层结合在一起，组成更为复杂的学习函数。从小的细节入手，一步步建立更大更复杂的模型，就需要建立深层神经网络来实现。

同样的，对于语音识别，第一层神经网络可以学习到语言发音的一些音调，后面更深层次的网络可以检测到基本的音素，再到单词信息，逐渐加深可以学到短语、句子。

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。

1.2.1 前向传播

输入：$a^{[l-1]}$
输出：$a^{[l]}$、缓存 $z^{[l]}$
公式：

$$z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]} \tag{12}$$
$$a^{[l]} = g^{[l]}\left(z^{[l]}\right) \tag{13}$$

向量化公式：

$$z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]} \tag{14}$$
$$A^{[l]} = g^{[l]}(Z^{[l]}) \tag{15}$$

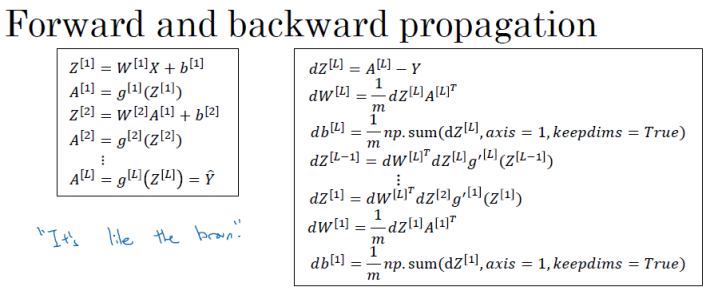
1.2.2 反向传播

输入： $da^{[l]}$
输出： $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$
公式：

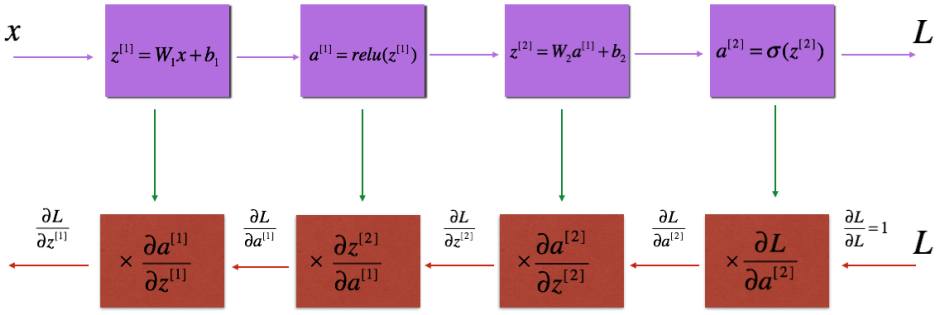
$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]}) \tag{16}$$
$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]} \tag{17}$$
$$db^{[l]} = dz^{[l]} \tag{18}$$
$$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]} \tag{19}$$
$$dz^{[l]} = w^{[l+1]T} dz^{[l+1]} \cdot g^{[l]\prime}(z^{[l]}) \tag{20}$$

向量化公式：

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}\left(Z^{[l]}\right) \tag{21}$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T} \tag{22}$$
$$db^{[l]} = \frac{1}{m}\, np.sum(dz^{[l]}, axis = 1, keepdims = True) \tag{23}$$
$$dA^{[l-1]} = W^{[l]T} . dZ^{[l]} \tag{24}$$



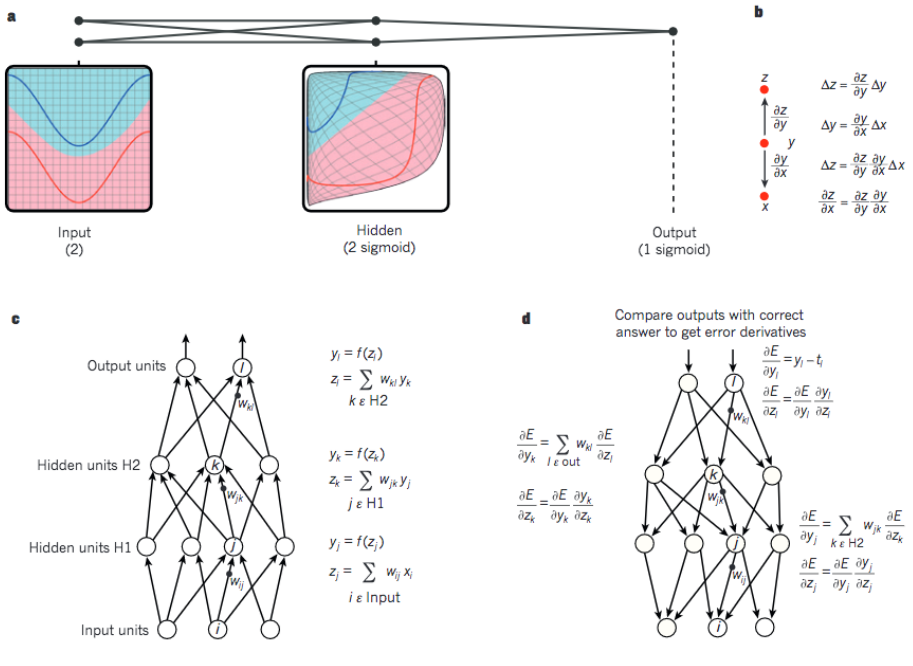
1.2.3 搭建深度神经网络块



神经网络的一步训练（一个梯度下降循环），包含了从 $a^{[0]}$（即 x）经过一系列正向传播计算得到 $\hat{y}$（即 $a^{[L]}$）。然后再计算 $da^{[L]}$，开始实现反向传播，用链式法则得到所有的导数项，W 和 b 也会在每一层被更新。

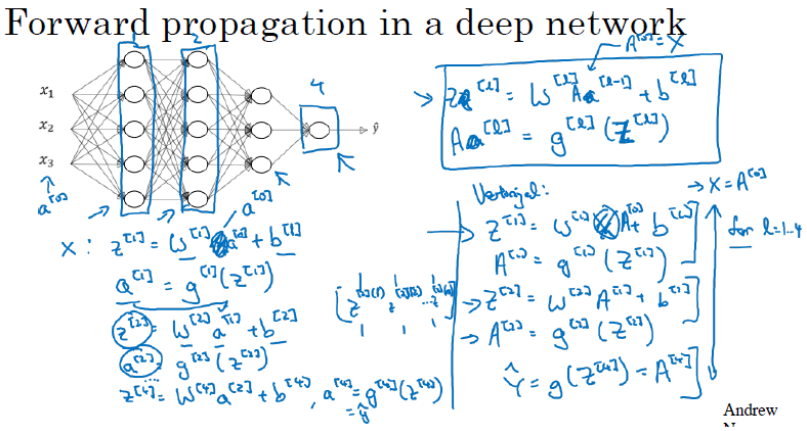
在代码实现时，可以将正向传播过程中计算出来的 z 值缓存下来，待到反向传播计算时使用。

补充一张从 Hinton、LeCun 和 Bengio 写的深度学习综述中摘下来的图，有助于理解整个过程：



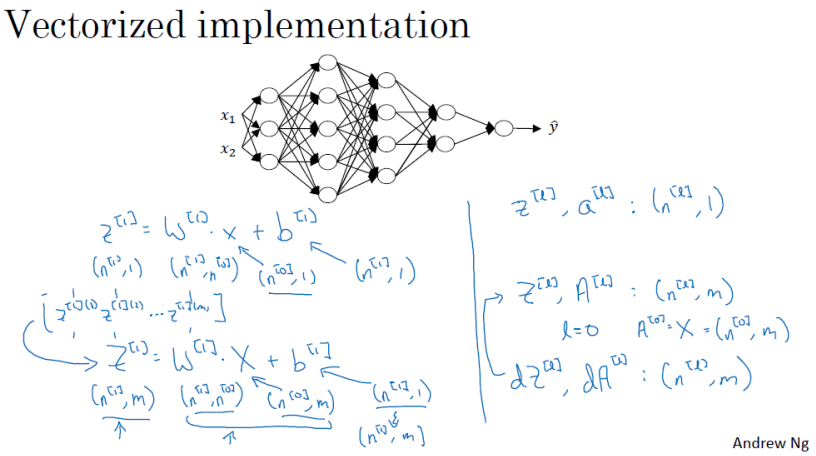
1.2.4 核对矩阵的维度

- 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，$Z^{[l]} = (z^{[l][1]}, z^{[l][2]}, z^{[l][3]}, \dots, z^{[l][m]})$,
- m为训练集大小，所以$Z^{[l]}$的维度不再是$(n^{[l]}, 1)$，而是$(n^{[l]}, m)$。
- $A^{[l]}: (n^{[l]}, m)$，$A^{[0]} = X = (n^{[l]}, m)$



1.2.5 参数和超参数

参数即是我们在过程中想要模型学习到的信息（模型自己能计算出来的），例如 $w^{[l]}, b^{[l]}$。而**超参数（hyper parameters）**即为控制参数的输出值的一些网络信息（需要人经验判断）。超参数的改变会导致最终得到的参数 $w^{[l]}, b^{[l]}$ 的改变。

典型的超参数有：

- 学习速率：α
- 迭代次数：N
- 隐藏层的层数：L
- 每一层的神经元个数：$n^{[1]}, n^{[2]}, \ldots$
- 激活函数 g(z) 的选择

当开发新应用时，预先很难准确知道超参数的最优值应该是什么。因此，通常需要尝试很多不同的值。应用深度学习领域是一个很大程度基于经验的过程。