神经网络与深度学习

第一周 深度学习概论

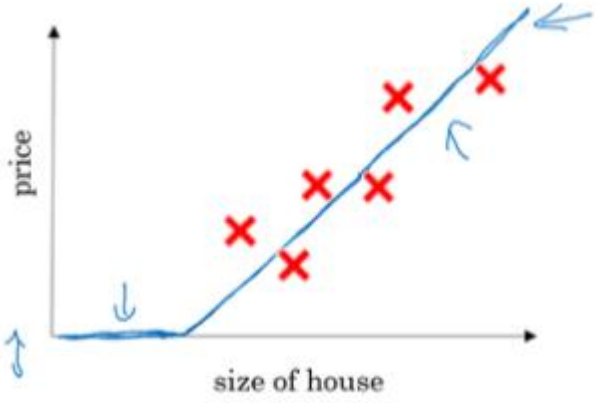
本周的课程主要介绍了神经网络的概念、分类、应用以及深度学习兴起的原因。

1.1 绪论

1.2 什么是神经网络

引例：已知几个房子的房价与面积，通过面积预测未知房价（最简单的神经网络）

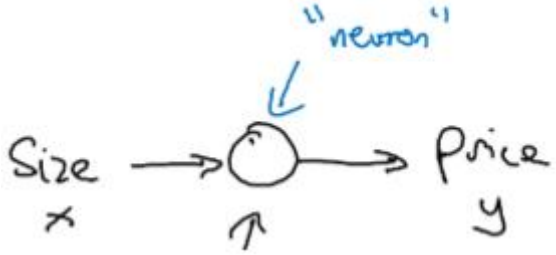
类似高中通过最小二乘法求回归曲线，本处也可通过方差最小的方式将房价与面积进行线性拟合，注意到，房价不能为负，故当面积小到一定值时，房价应该为 0，最终结果如右图（ReLU 函数）。



在上述过程中，我们只需要按照已知的房价-面积关系，对其进行线性拟合，即可求出面积到房价较为精确的映射，利用该映射，当给定面积时，就可以估算出其房价。

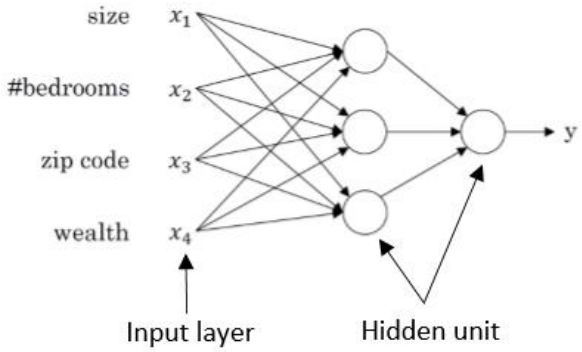
上述的逻辑关系可以归结为下图所示关系

神经网络中所谓的“神经元”可以视为一个线性或非线性的映射，给予其输入 x，即可输出结果 y，而神经网络的训练过程，就是以训练集（输入-输出对）对神经元中的参数进行调整，最终得到一个由 x 到 y 的精确映射。



实际上，很多因素诸如卧室数、邮政编码、步行化程度等均会对房价产生较大的影响。而当对房价的影响因素增多时，单单使用一个神经元对输出和输出进行拟合就远远不够了，这时，我们需要将多个神经元堆叠起来，形成一定规模的神经网络。

如右图所示，神经网络中每个神经元均接收输入的数据，将输出的结果与训练集中的结果比对，调整网络中的参数，进而提高对训练集的拟合程度。注意到：上述神经网络中，第一层每个神经元均接收上一层所有神经元（输入）的输出结果。



小结：本节课首先以房地产依靠面积预测房价的实际问题出发，为我们介绍

了最简单的神经网络-单神经元网络，然后将该问题进行拓展，对房价的影响因素由一个变为多个，这样我们就需要多神经元网络来解决问题。

总结知识点：

神经元是神经网络最基础的结构，可以视为一个最简单的映射，按照一定的规则根据输出得到输出。

神经网络是由多个神经元堆叠形成的网状结构，类似人脑，其可以通过各个神经元输出的加权与调参来进行“学习”“记忆”等活动。其核心思想是利用多个简单映射（神经元）的合成、加权来拟合一个较为复杂、抽象的映射。

拓展：人脑的学习与记忆方式

在人脑中存在着数以亿计的神经元，每个神经元之间通过突触进行连接，在神经信号发出时，一个神经元会对接收的信号进行汇总（对各个输入进行加权），然后根据已有的权重得到一个输出信号。而所谓的“知识”、“记忆”就储存在神经元间的权重中，可以说，神经元间的权重就是知识的存储形式。

1.3 用神经网络进行监督学习

监督学习的定义：监督学习，又叫有导师学习，顾名思义，就是训练集是由输入和输出共同构成的，每一个输入都对应着一个“标准答案”，神经网络将自身的输出结果与正确结果进行对比，然后根据差异进行调参，进而得出从输入到输出的精确映射，这个过程就是神经网络的训练过程；而无导师学习则只是要求相似的输入要有相似输出。

监督学习分类：监督学习大致分为两类：回归问题与分类问题。从输出结果来说，前者的结果是连续的而后者的结果是离散的。换句话说，前者是一个到某一连续区间的映射，后者是一个到某一离散集合的映射，也就是前者是定量的，后者是定性的。举个例子，上文提到的房地产预测问题就是一个回归问题，得到的结果是某一连续区间上的点；而像广告推荐问题则是分类问题，对于一个广告，用户的反馈要么是看（1），要么是不看（0），这样输出结果就是一个离散的集合{0,1}。

监督学习的应用：

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

上文提到的网络是标准神经网络（StdNN）主要用于房地产预测以及广告推荐；卷积神经网络（CNN）主要用于图像识别；循环神经网络（RNN）擅于处理序列数据（Sequence Data），主要用于语音识别（时间的一维序列）、机器翻译；而类似自动驾驶等输入较为多样、复杂的情形，则需要混合神经网络等较为复杂的网

络。

结构化数据与非结构化数据：

结构化数据主要指数据库中的数据，非结构化数据则是图片、音频等数据；计算机较为擅于处理结构化数据，在神经网络出现后，计算机处理非结构化数据的能力也有了较大的发展。

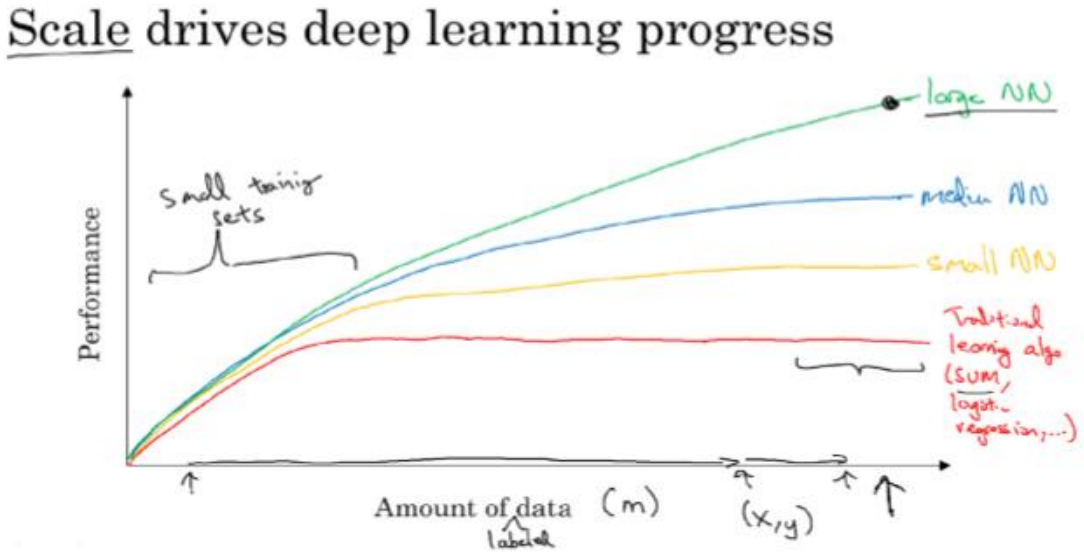
小结：

本节课主要介绍了监督学习的定义、分类及其应用，同时介绍了结构化数据与非结构化数据在监督学习领域的应用。

1.4 为什么深度学习会兴起

Scale drives deep learning progress.

规模既指数据规模大，又指神经网络规模大。



上图中，横轴代表数据规模，纵轴代表效果。从数据规模角度来看，在较小规模数据范围内，各个神经网络算法与早期的学习算法（如支持向量机）差距并不明显，甚至还会出现不如早期学习算法的情况。而到了大数据范围内，神经网络算法明显优于早期学习算法，并且神经网络规模越大，效果越好。

深度学习兴起的原因可以总结为两个方面：一方面是海量数据的产生。信息化时代产生的海量数据为神经网络的训练提供了良好的训练集，由上图可知，随着数据量的增大，大规模神经网络的优势愈发突出。另一方面则是技术的进步。神经网络的训练需要消耗巨大的算力，目前的解决方案主要是将计算移植到 GPU 上。同时，算法的改善，例如将 sigmoid 函数换为 ReLU 函数，也同样大大提高了神经网络的运行速度。

1.5 关于这门课

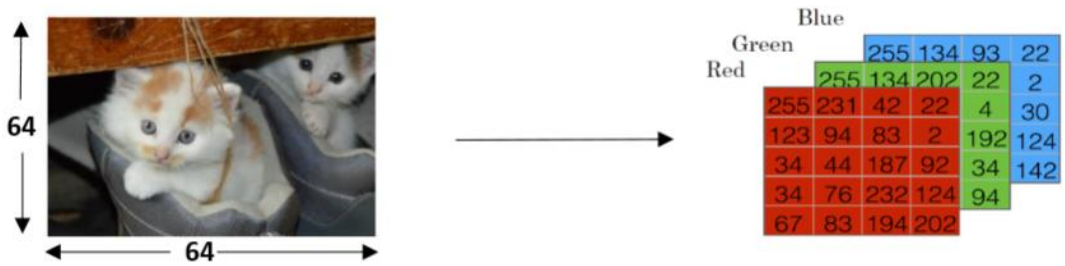
1.6 课程资源

第二周 神经网络基础

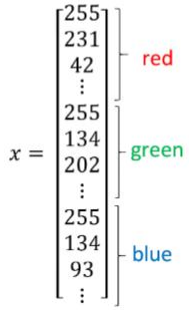
2.1 二分分类

引例：判断图片中是否有猫

首先，分析问题，寻找输入。由于神经网络本质上来讲是一种映射，所以说寻找一个合适的输入形式就显得十分重要。图片属于非结构化数据，要用计算机处理，首先要将其转化为计算机可以识别的模式。如下图所示，图片的存储方法主要有 RGB 模型，即一张图片在计算机中以三个矩阵的形式存储，每个矩阵分别存放该图片每个像素点的 RGB 值。



对于计算机而言，处理矩阵的步骤较为复杂，会耗费较大的算力，故可以将矩阵的信息提取出来，形成一个特征向量 X，其中 X 是一个 64×64×3 的列向量，包含上述三个矩阵的全部信息，且处理较为方便。



有了输入，判断图片中是否有猫的问题就转化为，设法构建一个从列向量 X 到{0,1}的映射 f，使得当 X 对应图片中有猫时，f(X)=1，反之 f(X)=0，这样，就将所有的输入图片分为两类。

由上述例子可归纳，二分分类本质上是构建一个映射，该映射的值域是{0,1}。即按照一定的规律或法则，将所有输入分为两类，这属于监督学习中的分类问题，也应该是最简单的分类问题。实现方法则可以是训练一个神经网络来寻找一个较为精确的映射。

课程中常用的符号（Notation）

(x,y) ： 表示一个单独的样本
x ： 表示 n_x 维的特征向量
$x \in R^{n_x}$ ： 表示样本 x 包含 n_x 个特征
y ： 标签，值为 0 或 1
$(x^{(n)}, y^{(n)})$ ： 表示样本 n 的输入和输出
$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \cdots\cdots (x^{(m)}, y^{(m)})\}$ ： 训练集
m_test ： 测试集的测试样本个数
m / m_train ： 训练集的训练样本个数

2.2 logistic 回归

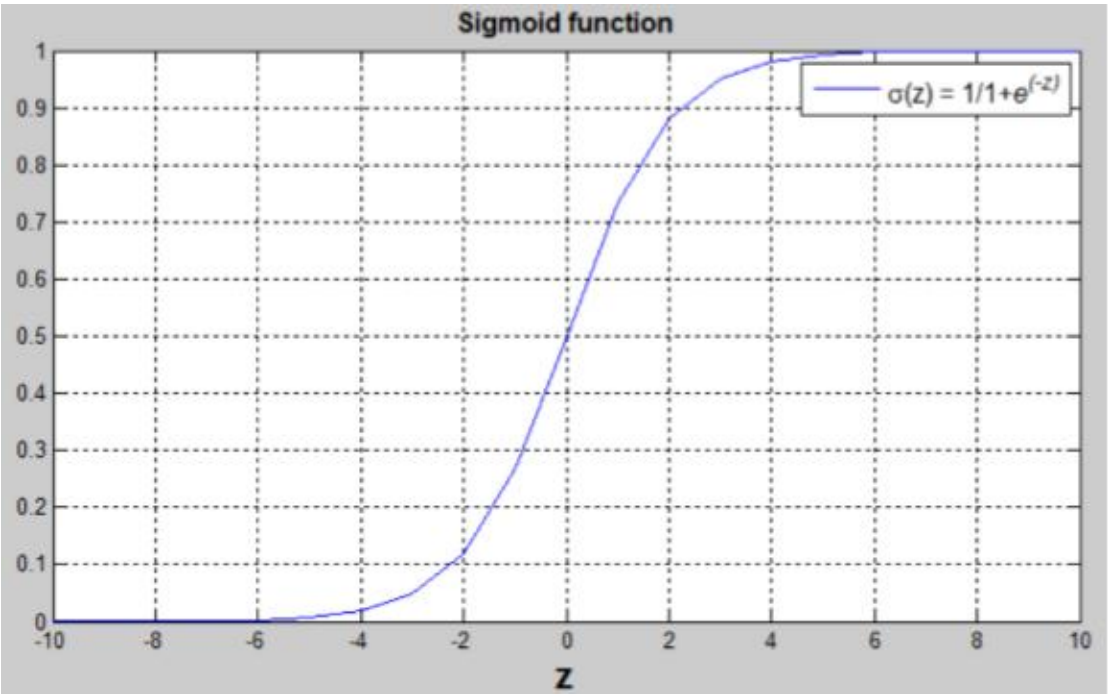
接上文引例

分析：

已知训练集中很多(x,y)序对，要求一个 x 到 y 的精确映射，最简单易行的方式应该是求线性回归方程 $\hat{y} = w^T x + b$（1），其中，权重：$w \in R^{n_x}$ ，n_x 是特征值数量；阈值：$b \in R$

但（1）式存在一个问题，即 $\hat{y}$ 的范围不可控，可能是负值，也可能很大，而我们想要得到的 $\hat{y}$ 是图片中含有猫的概率，应该位于区间(0,1)内。故需要对结果进行压缩得到 $\hat{y} = \sigma(w^T x + b)$ 使得 $\hat{y}$ 位于区间(0,1)内。

Sigmoid 函数



Sigmoid 函数解析式为 $\sigma(z) = \frac{1}{1+e^{-z}}$ ，由解析式和图像可知，sigmoid 函数的定义域为 R，值域为(0,1)，可以将 $\hat{y}$ 的范围很好地限制的(0,1)区间内。

小结：logistic 回归是线性回归方程的类推，首先利用线性回归方程对训练集中的离散点进行拟合，随后利用 sigmoid 函数对结果进行压缩，得到一个合理的结果。

2.3 logistic 回归损失函数

对于上一小节中，logistic 回归模型的预测结果 $\hat{y}^{(i)} = \sigma\left(w^T x^{(i)} + b\right)$ ，需要建立一个函数来对其“精确度”进行评估，这个函数就是损失函数(loss function)。

自然地，我们会想到将损失函数定义为距离型，即 $L\left(\hat{y}^{(i)}, y(i)\right) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$ （1），（1）式十分直观易懂，其几何意义就是两点间的距离，但其在优化操作时会导致优化问题变为非凸的，难以求解，故对重新定义如下的损失函数：

$$L\left(\hat{y}^{(i)}, y^{(i)}\right) = -y^{(i)} log\left(\hat{y}^{(i)}\right) - \left(1 - y^{(i)}\right) log\left(1 - \hat{y}^{(i)}\right)$$

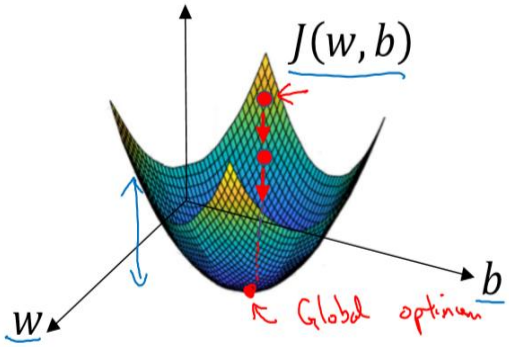
对于训练集中的元素 $(x^{(i)}, y^{(i)})$ 均有一个损失函数值 $L\left(\hat{y}^{(i)}, y^{(i)}\right)$ 与之相对应，这样可以很好的评估单个元素的拟合程度。而为了评估模型对整个训练集的

拟合程度，我们则需要引入一个新的函数，成本函数(cost function)：$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})$，该函数是训练集所有元素损失函数值的平均值，可以很好的衡量模型中权重 w 和阈值 b 对训练集的拟合程度。

小结：损失函数与成本函数的目的就是为了衡量模型的拟合程度如何。其大致思路如下：为了达到衡量的目的，我们需要一个函数，自然地，我们会想到距离，但使用距离衡量会带来优化问题非凸的难题，进而使用一种较为奇怪的方式进行衡量。有了一个元素拟合程度的度量后，对所有元素的拟合程度取平均值，即可得到整个训练集的拟合程度。

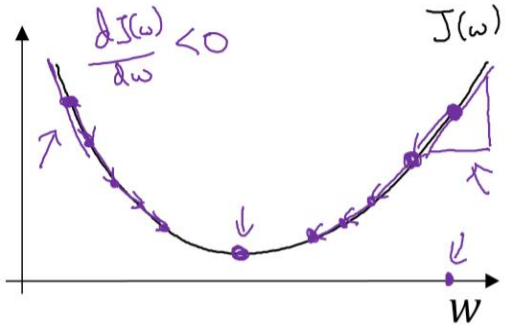
2.4 梯度下降法

承接上一小节内容，在上一小节中，我们得到了评估模型中权重 w 与阈值 b 对于训练集拟合程度的函数，成本函数 $J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})$。为了增大拟合程度，我们需要找到 w 和 b 使得成本函数 $J(w,b)$ 最小，梯度下降法就是寻找合适的 w 和 b 的一种方法。



梯度下降法：

由右图可以看出，成本函数 $J(w,b)$ 是一个凸函数，即 $J(w,b)$ 只有一个极值点，这样就可以通过梯度下降法求得全局最优解。



以一元函数为例，梯度下降法就是将 w 按照一定的规则不断更新，即 $w=w-\alpha\frac{\mathrm{d}J(w)}{\mathrm{d}w}$，其中，$\alpha$为学习率。分析该过程不难得知，随着迭代次数的增加，w 将会沿着梯度方向不断趋近于全局最优解，在迭代一定次数后，将会得到全局最优解或全局最优解的近似值。

在实际操作时，由于成本函数为 w 和 b 的二元函数，故迭代过程中也要对 b 进行更新，即重复 $w=w-\alpha\frac{\mathrm{d}J(w,b)}{\mathrm{d}w}$ 和 $b=b-\alpha\frac{\mathrm{d}J(w,b)}{\mathrm{d}b}$。

小结：2.2 节中给出了拟合的数学模型，2.3 节中给出了评估拟合结果的两个函数，本节中则给出了提高拟合度的方法。Logistic 回归先将复杂的分类问题转化为了线性回归拟合的问题，进而给出了评估拟合度和提高拟合度的方法，最终使得模型能够得以应用。

2.5 导数

I am among that maybe smaller group of people that are expert in calculus and I am very familiar with calculus. So it is definitely okay for me to skip this video.

2.6 更多导数的例子

同上

2.7 计算图

计算图的概念比较像《集合论与图论》中表示计算的二元树（如下图）

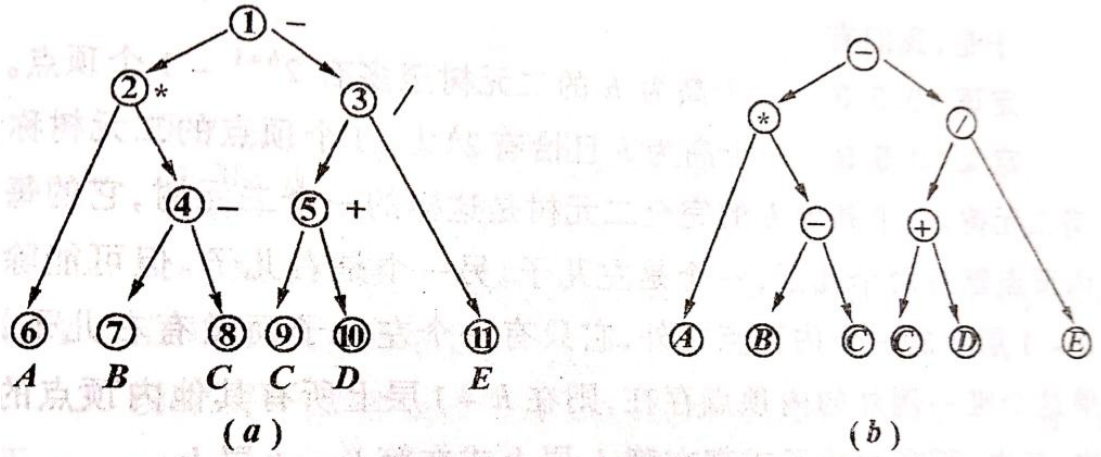
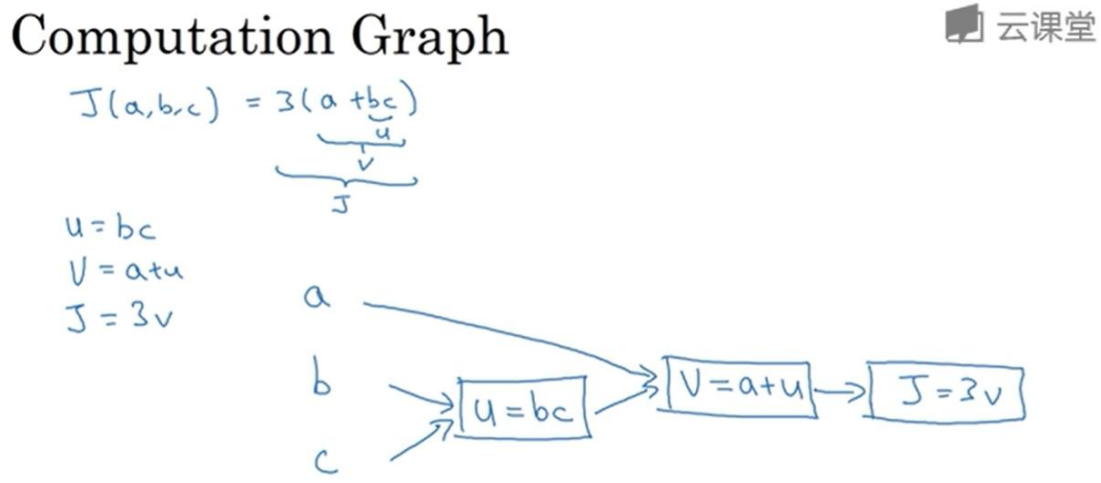


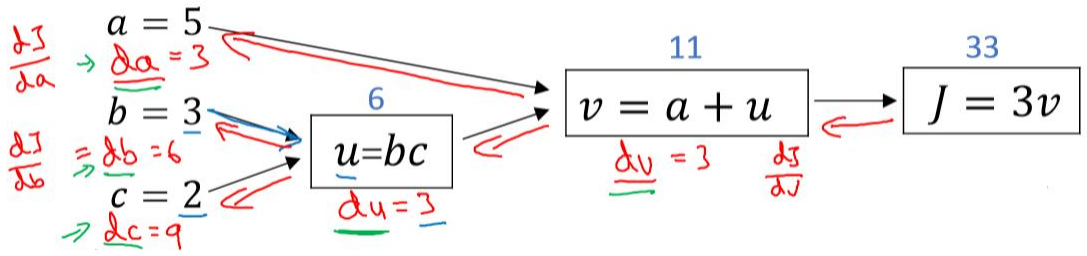
图 10.5.4　表达式 $A*(B-C)-(C+D)/E$ 的树表示

其主要思想就是将函数计算过程中的每一步都显式地表达出来，按照计算先后法则分别进行计算，最终得到函数值。这样，求函数值的过程可以看作是在计算图上的正向传播。



2.8 计算图的导数计算

观察计算图的结构不难发现，其层次结构与数学分析求导中的链式法则十分相像，故利用这种结构对函数进行求导会十分便捷。



承接 2.7 节中的函数，以求 $\frac{dJ}{da}$ 为例，根据计算图，首先要求出 $\frac{dJ}{dv}$ ，然后再求 $\frac{dv}{da}$ ，最终 $\frac{dJ}{da}=\frac{dJ}{dv}\times\frac{dv}{da}$ 得到 $\frac{dJ}{da}$ 的数值。类似地，$\frac{dJ}{db}$ 、$\frac{dJ}{dc}$ 也可按照上述步骤求出。这样，求导的过程可以看作是在计算图上的反向传播。

约定：在 Python 编程实现时，由于对哪个函数求导较为清楚，故诸如 $\frac{dJ}{da}$ 等微商的形式，可以简写为 da。

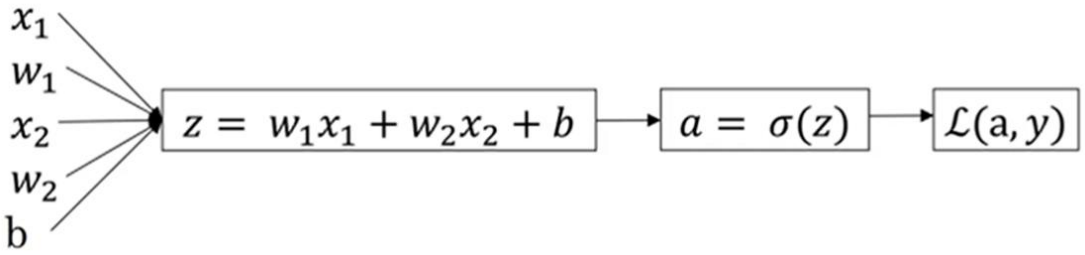
2.9 logistic 回归中的梯度下降法

回顾 2.2 节中的内容，我们有预测函数 $z=w^Tx+b$ ，$\hat{y}=\sigma(z)=a$

在 2.3 节中，我们知道了如何评估权重 w、阈值 b 对单个样本的拟合度，即损失函数 $L(a,y)=-y\,log\,(a)-(1-y)\,log\,(1-a)$

在本节中，不妨设训练集只有一个元素（多个元素的最优化将在后续章节中介绍），则成本函数与损失函数相等。

将成本函数以计算图的形式表达如下



按照计算图反向传播的方式分别求出 L 关于 w_1、w_2、b 的偏导数，然后按照 2.4 节中的步骤对 w_1、w_2、b 进行更新，在迭代一定次数后得到最优解。

2.10 m 个样本的梯度下降

2.9 节中，我们利用一个特例（训练集只有一个元素）演示了 logistic 回归中梯度下降法的应用，而现实应用中往往训练集有很多个元素，并且函数本身也有多个特征值，在本节中，将会利用 m 个元素的训练集，假定函数有 n 个特征值，应用梯度下降法。

Python 伪代码如下图所示

```
#初始化成本函数及其导数
J = 0
for i in range(1,n):
    dw[i] = 0
db = 0

#计算成本函数对各参数值的偏导
#外层循环遍历训练集
for i in range(1,m):
    z[i] = w1*x1 + w2*x2 + b
    a[i] = sigmoid(z[i])
    J += -(y[i]*log(a[i]) + (1-y[i])*log(1-a[i]))
    dz[i] = a[i]*(1-a[i])
    #内层循环遍历所有特征值
    for j in range(1,n):
        dw[j] += x[i][j] * dz[i]
    db += dz[i]
J /= m
for i in range(1,n):
    dw[i] /= m
db /= m

#更新参数值
for i in range(1,n):
    w[i] = w[i] - α*dw[i]
b = b - α*db
```

存在的问题：for 循环影响代码性能

由上图可知，为了遍历训练集和所有特征值，代码实现的过程中需要嵌套使用两个 for 循环，并且在更新参数值的过程中也会用到 for 循环，这样过多层嵌套会导致代码效率低下，为了提高代码效率，下一节中会介绍向量化方法。

2.11 向量化

可以看到，上一节中梯度下降法的实现是利用两层嵌套的 for 循环，这样会导致代码效率低下，在本节将会介绍向量化方法来提高代码效率。

向量化方法顾名思义，就是在参数计算的过程中不利用 for 循环将向量 W、X 的各维度一一计算，而是利用 Python 的 numpy 库，将 W、X 直接以向量的形式进行计算。具体代码对比演示如下。

```
.vscode > vectorization.py > ...
1   import numpy as np
2   import time
3
4   #随机产生一百万个随机数
5   a = np.random.rand(1000000)
6   b = np.random.rand(1000000)
7
8
9   #计算向量化方法相乘时间
10  tic = time.time()
11  c = np.dot(a,b)
12  toc = time.time()
13  print("Vectorization:"+str(1000*(toc-tic))+"\nResult:"+str(c))
14
15
16  #计算for循环方法相乘时间
17  c = 0
18  tic = time.time()
19  for i in range(1,1000000):
20      c += a[i]*b[i]
21  toc = time.time()
22  print("For loop:"+str(1000*(toc-tic))+"\nResult:"+str(c))
23
```

```
Vectorization:1.00398063659667797
Result:249750.17975935753
For loop:560.5001449584961
Result:249750.02335297843
PS E:\Python\编程文件>
```

原因：向量化方法充分利用并行化处理方法来进行计算，人工智能算法多利用 GPU 的原因就是因为 GPU 更擅长进行并行计算。SIMD，单指令流多数据流

2.12 向量化的更多例子

在编写神经网络程序时，经验表明程序中 for 循环越少，程序效率越高，这样就需要使用向量化方法来避免显式的 for 循环。下面是向量化的几个例子。

```
.vscode > vectorization.py > ...
1  import numpy as np
2  import time
3
4  #产生随机数矩阵
5  a = np.random.rand(10000,10000)
6  b = np.random.rand(10000)
7
8
9  #计算向量化方法相乘时间
10 tic = time.time()
11 c = np.dot(a,b)
12 toc = time.time()
13 print("Vectorization:"+str(1000*(toc-tic)))
14
15
16 #计算for循环方法相乘时间
17 c = np.zeros((10000,1))
18 tic = time.time()
19 for i in range(1,10000):
20     for j in range(1,10000):
21         c[i] += a[i][j] * b[j]
22 toc = time.time()
23 print("For loop:"+str(1000*(toc-tic)))
24
25
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

  File "e:/Python/编程文件/.vscode/vectorization.py", line 17, in <module>
    c = np.zeros(10000,1)
TypeError: Cannot interpret '1' as a data type
PS E:\Python\编程文件> & E:/Python/python.exe e:/Python/编程文件/.vscode/vectorization.py
Vectorization:46.87643051147461
For loop:277822.52836227417
PS E:\Python\编程文件>
```

下面是利用向量化方法去掉一个 for 循环的伪代码

```
.vscode > logistic.py > ...
1  #初始化成本函数及其导数
2  J = 0
3  #向量化版本
4  dw = np.zeros((n,1))
5  db = 0
6
7  #计算成本函数对各参数值的偏导
8  #外层循环遍历训练集
9  for i in range(1,m):
10     z[i] = w1*x1 + w2*x2 + b
11     a[i] = sigmoid(z[i])
12     J += -(y[i]*log(a[i]) + (1-y[i])*log(1-a[i]))
13     dz[i] = a[i]*(1-a[i])
14     #向量化版本
15     dw = x[i] * dz[i]
16     db += dz[i]
17 J /= m
18 #向量化版本
19 dw /= m
20 db /= m
21
22 #更新参数值
23 #向量化版本
24 w = w - α*dw
25 b = b - α*db
```

2.13 向量化 logistic 回归

上节中讲述了利用向量化将双层 for 循环变为单层 for 循环，本节中将介绍在 logistic 回归的正向传播过程中利用向量化取缔程序中所有 for 循环。

我们可以将输入信息写成一个矩阵的形式，即训练输入矩阵 $X = [x_1, x_2, \cdots, x_m]$，其中$x_i$为列向量；然后利用矩阵乘法就可以计算出激活函数值，具体步骤如下：

$$X = [x_1, x_2, \cdots, x_m]$$

$$Z = [z_1, z_2, \cdots, z_m] = w^T X + [b, b, \cdots, b] = [w^T x_1 + b, w^T x_2 + b, \cdots, w^T x_m + b]$$

$$A = [a_1, a_2, \cdots, a_m] = \sigma(Z)$$

在代码实现的过程中，只需要两行代码即可

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

注意到：该段代码第一行中 np.dot(w.T,X)是一个行向量，而 b 是一个标量，但仍可进行相加操作，原因是在执行该段代码过程中，b 先被拓展为一个 m 维行向量再参与运算，这种机制叫做广播，在 2.15 节中将进行详细解读。

2.14 向量化 logistic 回归的梯度输出

上节介绍了通过向量化进行正向传播求 Z 的方法，本节将介绍通过向量化进行反向传播求导数的方法。具体步骤如下：

$$dZ = A - Y = [a_1 - y_1, a_2 - y_2, \cdots, a_m - y_m]$$

$$dW = X\, dZ^T / m$$

$$db = (z_1 + z_2 + \cdots + z_m)/m$$

伪代码实现向量化 logistic 回归如下：

```
import numpy as np
#初始化成本函数及其导数
J = 0
dW = np.zeros((n,1))
db = 0
times = 10000
for i in range(1,times):
    #计算成本函数对各参数值的偏导
    Z = np.dot(w.T,X) + b
    A = sigmoid(Z)
    dZ = A - Y
    dW = np.dot(X,dZ.T) / m
    db = np.sum(dZ) / m
    #更新参数值
    w = w - α*dW
    b = b - α*db
```

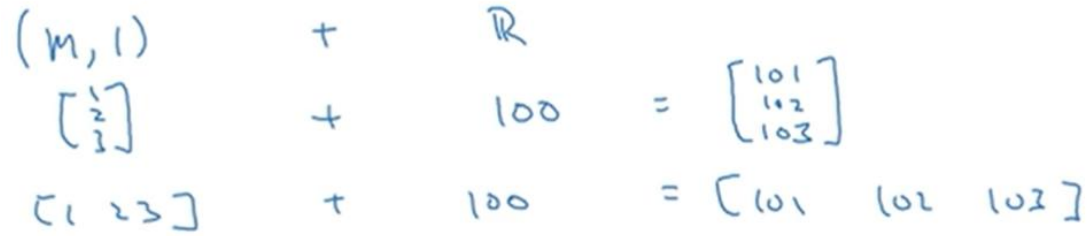
2.15 Python 中的广播

由于矩阵的加减法、乘除法对矩阵的形式有着特殊的要求，如果两个矩阵不满足一定的形式，那么它们将无法进行运算。Python 中的广播就是在两个矩阵无法进行某一运算时，通过一定的规则将矩阵进行拓展，进而使得两个矩阵可以进行运算。下面将会列举编写神经网络时常用的几种广播的形式。

1、向量与标量相加减

向量与标量相加减时，首先会将标量拓展为与目标向量相同维数的向量，再按照向量加减法进行加减。2.13 中的 np.dot(w.T,X) + b 就是这种类型的广播。

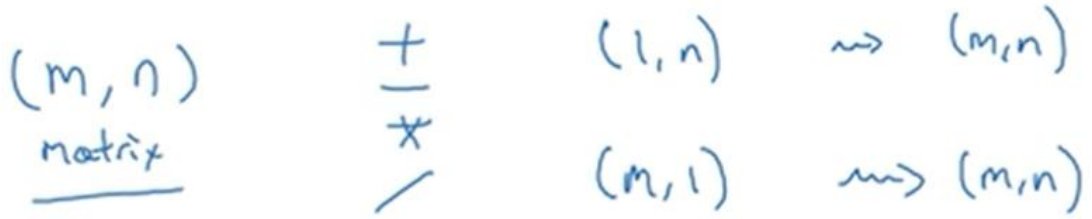
例：



2、矩阵与向量的加减乘除

矩阵与向量进行加减乘除时，首先会把向量按行/列拓展为与目标矩阵相同的形式，然后再与目标矩阵进行运算

例：



应用实例：

```python
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]])

n = np.sum(A,axis=0)
#axis=0纵向求和 axis=1横向求和

print(A/n*100)
```

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

2.16 关于 python/numpy 向量的说明

秩为 1 的数组：

代码 a = np.random.randn(5)执行后得到的 a 不是行向量或列向量，而是一个秩为 1 的数组，即执行 a.shape()后得到的结果是(5,)。

这个数据结构很神奇，不要使用，会造成很多 bug。

防范措施：

不使用 a = np.random.randn(5)而是使用 a = np.random.randn(5,1)来获取行向量

若得到了一个秩为 1 数组，利用 a.reshape(5,1)将其转化为向量

利用 assert()函数验证：Python 中很多时候有很奇怪的方法将不能计算的东西计算出结果而不报错，利用 assert()函数可以验证断言是否正确，进而人工进行报错。例：assert(a.shape == (5,1))

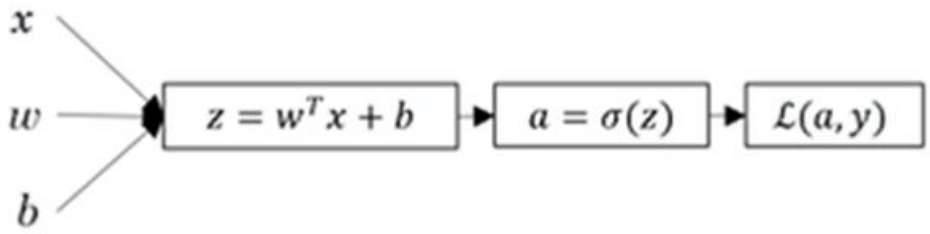
2.17 Jupyter/Ipython 笔记本的快速指南

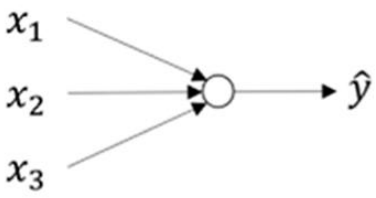
具体操作类似 C 语言作业提交系统

第三周 浅层神经网络

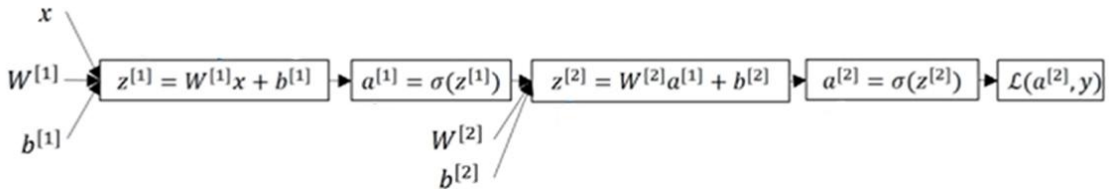
3.1 神经网络概览

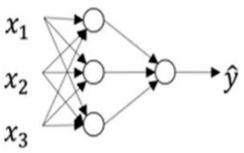
回顾：上周的课程主要介绍了 logistic 回归及其实现方法





如上图、右图所示，logistic 回归也可视为一种及其简单的神经网络，简单到只有一个神经元，训练输入 X 直接输入到神经元中，计算得到损失函数 L，然后通过梯度下降法迭代改变权重 W 和阈值 b，最终得到一个从 X 到 y 的精确映射。而在实践过程中，一个神经元得到的映射往往是不够精确的，这样，我们将多个神经元进行堆叠，形成网状结构，这样就构建出了神经网络。



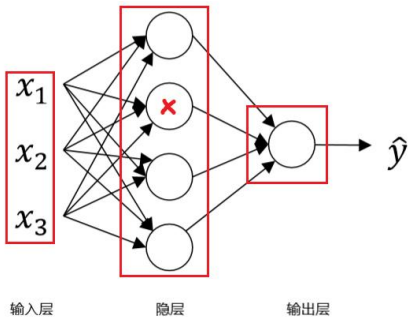


如上图、右图所示，通过神经元的堆叠，我们得到了一个相对复杂的神经网络，训练输入 X 直接作用于输入层，输入层按照 logistic 回归的法则计算得到结果 $\sigma(Z^{[1]})$，并将其作为第二层的输入参与第二层的计算，最终得到输出结果 $\hat{y}$ 。

相应地，类似 logistic 回归，神经网络也会按照计算图进行反向传播，即求导，进而通过梯度下降法调参，这些内容会在本周的后续课程中进行深入讲解。

3.2 神经网络表示

如右图所示的神经网络，它是一个双层神经网络（输入层不计算层数），输入层是训练输入 X 的 n 个特征（X 是一个 n 维向量），负责向隐层传递数据；隐层被称为隐层是因为在训练集中，其真正数值是无法观测的，其主要对输入层的激活值按照类似 logistic 回归的算法进行计算得到隐层的激活值并向输出层传递；输出层主要负责输出 $\hat{y}$。



神经网络中涉及的符号：

激活值：激活值是不同层传递给第一层的值；用上标[i]表示第 i 层的激活值，下标 i 表示该层中第 i 个神经元产生的激活值。例如：$a_2^{[1]}$ 表示上图中标有“×”的神经元的激活值。一般地，输入层输入的向量 X 也可表示为 $a^{[0]}$；在图中，$a^{[1]}$ 是一个四维向量，即 $a^{[1]}=\left[a_1^{[1]},a_2^{[1]},a_3^{[1]},a_4^{[1]}\right]^T$，分别对应隐层四个神经元的激活值；$a^{[2]}$是一个实数，即 $a^{[2]}=\hat{y}$。

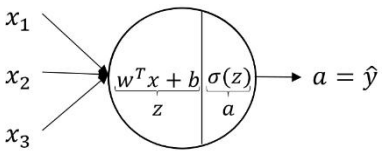
权重：权重的定义与 logistic 回归中的定义基本相同，差别之处在于在神经网络中，权重 W 可能是一个矩阵。其表示方式类似激活值，用上下标的形式表示第 i 层第 j 个神经元。在图中，$w^{[1]}$表示隐层的权重，是一个 4×3 矩阵，4 对应着隐层中的 4 个神经元，3 对应着输入向量的三个维。

阈值：阈值的定义与 logistic 回归中的定义基本相同，表示方式具体可参照权重的表示方式类推。

3.3 计算神经网络的输出

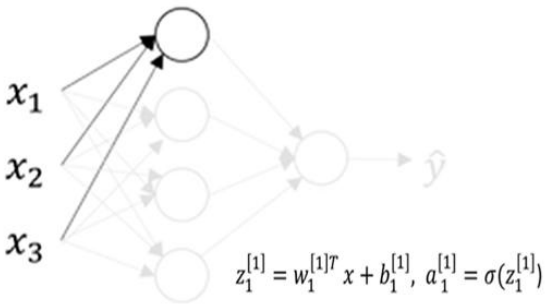
回顾：logistic 回归的计算（正向传播）

如右图所示，在 logistic 回归的正向传播过程中，我们首先要利用输入向量 X、权重 W 和阈值 b，计算 $z=w^Tx+b$ ，然后根据$\hat{y}=\sigma(z)=a$ 得出最终的预测值$\hat{y}$



逐层计算：

如右图所示，神经网络可以看作是很多个 logistic 回归单元的堆叠，那么在计算过程中，不妨将每层各个神经元分开看待，分别按照 logistic 回归进行计算。



向量化：

以隐层为例，对隐层的四个神经元分别按照 logistic 回归进行计算后，我

们得到下面四个式子

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]}) \quad z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]}) \quad z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

为了提高代码效率，我们要对上述式子进行向量化

$$z^{[1]} = w^{[1]}x + b^{[1]} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}, a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_1^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = sigmoid\left(z^{[1]}\right) \quad ①$$

这样，我们就得到了隐层的激活值$a^{[1]}$。输出层的计算则是标准的 logistic 回归，在此不再赘述。

3.4 多个例子中的向量化

上一节中主要介绍了对于单个输入，神经网络如何进行向量化，本节中将介绍如何将整个训练集进行向量化的方法。

对于 3.3 节中的①式，我们可以做一个简单的类推，①式中的单个输入 x 是一个 4 维列向量，训练集中有 m 个这样的 4 维列向量，把这些向量堆叠起来，那么整个训练集 X 就可以看作是一个 4×m 矩阵。相应地，向量化过程如下所示

$$Z^{[1]} = w^{[1]}X + b^{[1]} = \begin{bmatrix} w_1^{[1]T}x^{(1)} + b_1^{[1]} & w_1^{[1]T}x^{(2)} + b_1^{[1]}, \ldots, w_1^{[1]T}x^{(m)} + b_1^{[1]} \\ w_2^{[1]T}x^{(1)} + b_2^{[1]} & w_2^{[1]T}x^{(2)} + b_2^{[1]}, \ldots, w_2^{[1]T}x^{(m)} + b_2^{[1]} \\ w_3^{[1]T}x^{(1)} + b_3^{[1]} & w_3^{[1]T}x^{(2)} + b_3^{[1]}, \ldots, w_3^{[1]T}x^{(m)} + b_3^{[1]} \\ w_4^{[1]T}x^{(1)} + b_4^{[1]} & w_4^{[1]T}x^{(2)} + b_4^{[1]}, \ldots, w_4^{[1]T}x^{(m)} + b_4^{[1]} \end{bmatrix}$$

$$= \begin{bmatrix} z_1^{[1](1)}, z_1^{[1](2)}, \ldots, z_1^{[1](m)} \\ z_2^{[1](1)}, z_2^{[1](2)}, \ldots, z_2^{[1](m)} \\ z_3^{[1](1)}, z_3^{[1](2)}, \ldots, z_3^{[1](m)} \\ z_4^{[1](1)}, z_4^{[1](2)}, \ldots, z_4^{[1](m)} \end{bmatrix} = \left[z^{[1](1)}, z^{[1](2)}, \ldots, z^{[1](m)}\right]$$

$$A^{[1]} = sigmoid\left(Z^{[1]}\right) = \left[a^{[1](1)}, a^{[1](2)}, \ldots, a^{[1](m)}\right]$$

注：在矩阵$Z^{[1]}$、$A^{[1]}$、X 中，横向表示不同的训练样本，纵向表示不同神经元的计算结果

3.5 向量化实现的解释

本节主要证明上一节中通过将列向量堆叠的方式可以得到正确的结果

证明：

考察单个样本的向量化过程

$$z^{[1]} = w^{[1]}x + b^{[1]} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

可得 $z_i^{[1]} = w_i^{[1]T}x + b_i^{[1]}$ ①

多样本向量化过程中，$X = \begin{bmatrix} x_1^{(1)}, & x_1^{(2)}, \ldots, & x_1^{(m)} \\ x_2^{(1)}, & x_2^{(2)}, \ldots, & x_2^{(m)} \\ x_3^{(1)}, & x_3^{(2)}, \ldots, & x_3^{(m)} \\ x_4^{(1)}, & x_4^{(2)}, \ldots, & x_4^{(m)} \end{bmatrix} = \left[x^{(1)}, x^{(2)}, \ldots, x^{(m)}\right]$

$$Z^{[1]} = w^{[1]}X + b^{[1]} = \begin{bmatrix} w_1^{[1]T}x^{(1)} + b_1^{[1]}, & w_1^{[1]T}x^{(2)} + b_1^{[1]}, \ldots, & w_1^{[1]T}x^{(m)} + b_1^{[1]} \\ w_2^{[1]T}x^{(1)} + b_2^{[1]}, & w_2^{[1]T}x^{(2)} + b_2^{[1]}, \ldots, & w_2^{[1]T}x^{(m)} + b_2^{[1]} \\ w_3^{[1]T}x^{(1)} + b_3^{[1]}, & w_3^{[1]T}x^{(2)} + b_3^{[1]}, \ldots, & w_3^{[1]T}x^{(m)} + b_3^{[1]} \\ w_4^{[1]T}x^{(1)} + b_4^{[1]}, & w_4^{[1]T}x^{(2)} + b_4^{[1]}, \ldots, & w_4^{[1]T}x^{(m)} + b_4^{[1]} \end{bmatrix}$$

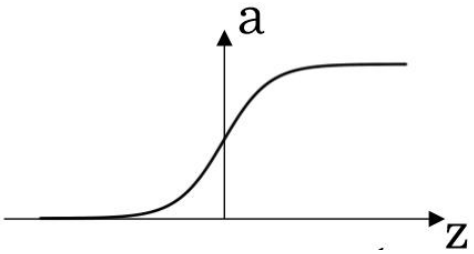
由①式可知，$Z^{[1]} = \begin{bmatrix} z_1^{[1](1)}, & z_1^{[1](2)}, \ldots, & z_1^{[1](m)} \\ z_2^{[1](1)}, & z_2^{[1](2)}, \ldots, & z_2^{[1](m)} \\ z_3^{[1](1)}, & z_3^{[1](2)}, \ldots, & z_3^{[1](m)} \\ z_4^{[1](1)}, & z_4^{[1](2)}, \ldots, & z_4^{[1](m)} \end{bmatrix} = \left[z^{[1](1)}, z^{[1](2)}, \ldots, z^{[1](m)}\right]$

【证毕】

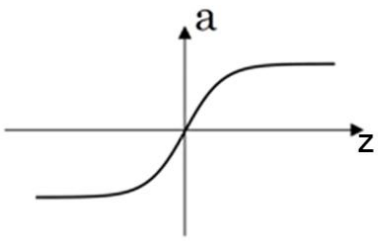
3.6 激活函数

本节中将会介绍几种常见的激活函数及其优缺点

1、sigmoid 函数和 tanh 函数



sigmoid：$a = \frac{1}{1+e^{-z}}$



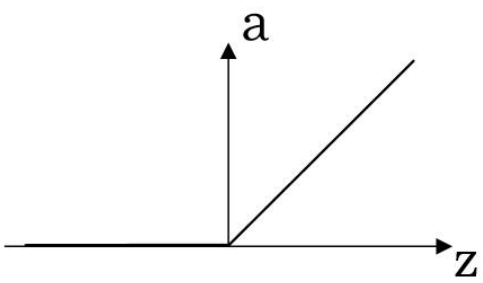
tanh：$a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

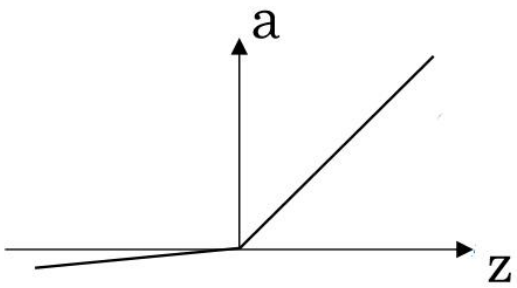
sigmoid 函数的输出区间为(0,1)，故其在二分分类时具有很大的优势，处理二分分类的神经网络的输出层激活函数一般可以使用 sigmoid 函数，其他时候不要用！

tanh 函数是 sigmoid 函数的平移和伸缩，其函数平均值接近 0，具有较好的数据中心化效果，神经网络的隐层中使用 tanh 函数的效果一般好于使用 sigmoid 函数，可以考虑在隐层中使用 tanh 函数。

二者具有一个共同的缺点，在 z 值很大或很小时，二者的梯度接近 0，这样会增加算法的迭代次数，降低效率。

2、ReLU 函数和 leaky ReLU 函数





ReLU：$a = max(0, z)$　　leaky ReLU：$a = max(0.01z, z)$

由于二者的斜率均为定值，ReLU 函数和 leaky ReLU 函数不存在斜率趋于 0 的部分，故二者的学习效率要远高于 sigmoid 函数和 tanh 函数。绝大多数神经网络的隐层激活函数都会采用 ReLU 函数（不确定用哪个就用 ReLU），也可以试试 leaky ReLU。

二者存在一个共同问题，就是在 z = 0 处导数没有定义。这个问题在实际应用中很少见，可以忽略；或者可以将 z = 0 处的导数定义为 0 或 1。

3.7 为什么需要非线性激活函数？

从神经网络最初的功能来看，神经网络主要的功能就是对生产中某些复杂的函数进行拟合，而这些函数往往都是非线性的。

假如神经网络中使用的激活函数均为线性激活函数，为了方便不妨设激活函数为恒等激活函数 $g(z) = z$，那么整个神经网络做的仅仅是对(x,y)进行线性拟合，一方面这种拟合的精确度不够，另一方面这种线性拟合与单纯的 logistic 回归区别不大，神经网络的使用没有意义。下面将要证明神经网络做的仅仅是对(x,y)进行线性拟合。

证明：

已知：$z^{[1]} = W^{[1]}x + b^{[1]}$ ，$a^{[1]} = z^{[1]}$

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ ，$a^{[2]} = z^{[2]}$

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

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

$\therefore\ a^{[2]} = W^{[2]}a^{[1]} + b^{[2]} = W^{[2]}\left(W^{[1]}x + b^{[1]}\right) + b^{[2]}$

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

$= W^{'}x + b^{'}$

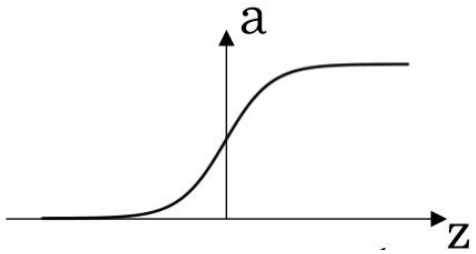
故 $a^{[2]}$ 仍为 x 的线性组合

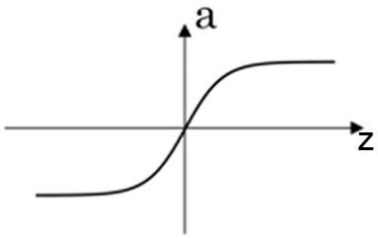
【证毕】

由上述证明可知，如果使用线性激活函数，那么神经网络的输出就是 x 的线性组合，这样就失去了神经网络的意义。但有时在一些回归问题的输出层（例如 1.2 节中提到的房地产预测问题），也可以使用恒等激活函数。

3.8激活函数的导数

1、sigmoid 函数和 tanh 函数





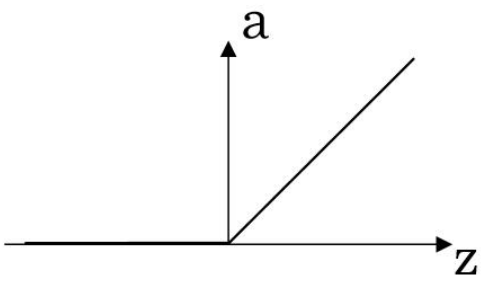
sigmoid：$a = g(z) = \frac{1}{1+e^{-z}}$

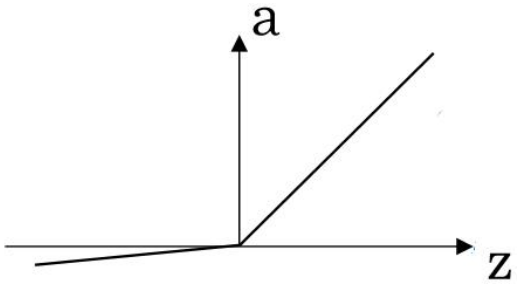
$$g'(z) = a(1-a)$$

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

$$g'(z) = 1 - a^2$$

2、ReLU 函数和 leaky ReLU 函数





ReLU：$a = g(z) = max(0, z)$

$$g'(z) = \begin{cases} 0, z < 0 \\ 1, z \geq 0 \end{cases}$$

leaky ReLU：$a = g(z) = max(0.01z, z)$

$$g'(z) = \begin{cases} 0.01, z < 0 \\ 1 \quad , z \geq 0 \end{cases}$$

3.9 神经网络的梯度下降法

本节将介绍实现双层神经网络梯度下降法的正向传播步骤、反向传播步骤和程序总体的结构。本节中假设神经网络的输出层激活函数为 sigmoid 函数，隐层激活函数未知。

1、正向传播

正向传播过程在 3.3 节中已有详细介绍，本节中仅写出相应公式，证明在此不再赘述。公式如下所示

$$Z^{[1]} = W^{[1]}x + b^{[1]} \quad , \quad A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \quad , \quad A^{[2]} = g^{[2]}(Z^{[2]})$$

2、反向传播

反向传播的计算过程可以视为按照计算图反向进行，即先计算输出层再计算隐层。输出层的计算与 logistic 回归类似，只不过“输入”要改换为隐层的输出；而隐层由于激活函数未知，并且涉及复合函数求导，其形式与 logistic 回归有较大的差异。

输出层计算：

$$dZ^{[2]} = A^{[2]} - Y \ , \ dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \qquad db^{[2]} =$$

$\frac{1}{m}np.sum(\mathrm{d}Z^{[2]}, axis=1, keepdims=True)$

注：keepdims=True 可以避免出现 shape 为(n,)的一维数组的情况，也可以用 reshape(n,1)显式地实现

隐层计算：

$\mathrm{d}Z^{[1]} = W^{[2]T}\mathrm{d}Z^{[2]} * g^{[1]'}(Z^{[1]})$（*表示矩阵元素相乘，故 $\mathrm{d}Z^{[1]}$是$n^{[1]}\times$m 矩阵）

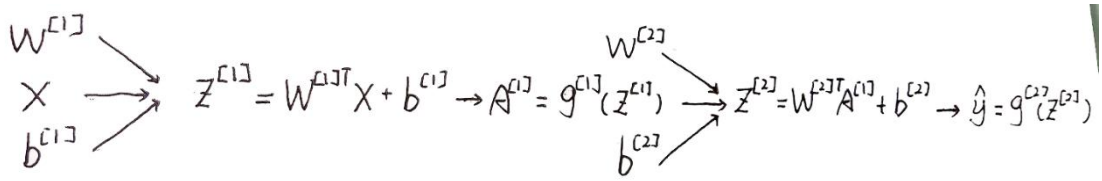
$$\mathrm{d}W^{[1]} = \frac{1}{m}dZ^{[1]}X^{T}$$

$$\mathrm{d}b^{[1]} = \frac{1}{m}np.sum(\mathrm{d}Z^{[1]}, axis=1, keepdims=True)$$

3、程序总体结构

与 logistic 回归类似，神经网络的大体思路也是预测-调参，主循环体的写法也类似，首先进行正向传播，然后进行反向传播（上文已经介绍），最终利用梯度下降法调参。但与 logistic 回归不同的是，神经网络涉及到隐层的参与，故调参时既要更新输出层参数也要更新隐层参数。注意到：与 logistic 回归不同，在神经网络的初始化时，不要将参数都初始化为 0，而是应该以随机数的方式对参数进行初始化。

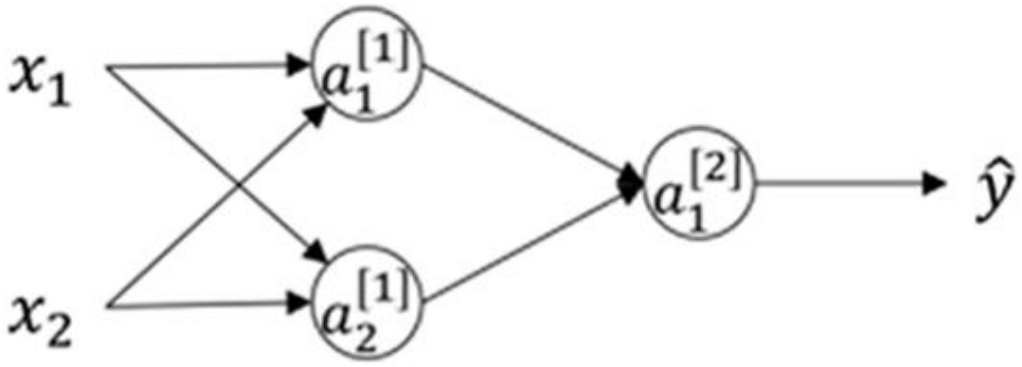
3.10 直观理解反向传播



根据计算图，很容易就可以算出 $\mathrm{d}Z^{[1]}$、$\mathrm{d}W^{[1]}$、$\mathrm{d}b^{[1]}$

3.11 随机初始化

问题：为什么不能将权重矩阵初始化为零阵？



以上图所示的神经网络为例

假设隐层的权重$W^{[1]}$和输入层的权重$W^{[2]}$均为零阵

那么将会出现 $a_1^{[1]} = a_2^{[1]}$的情况，由于神经网络具有高度的对称性通过归纳法可

证明，无论神经网络迭代多少次，$a_1^{[1]} = a_2^{[1]}$始终成立，这样隐层多个神经元的结构就失去了意义，但阈值 b 初始值是否为 0 影响不大。
解决方案：随机初始化

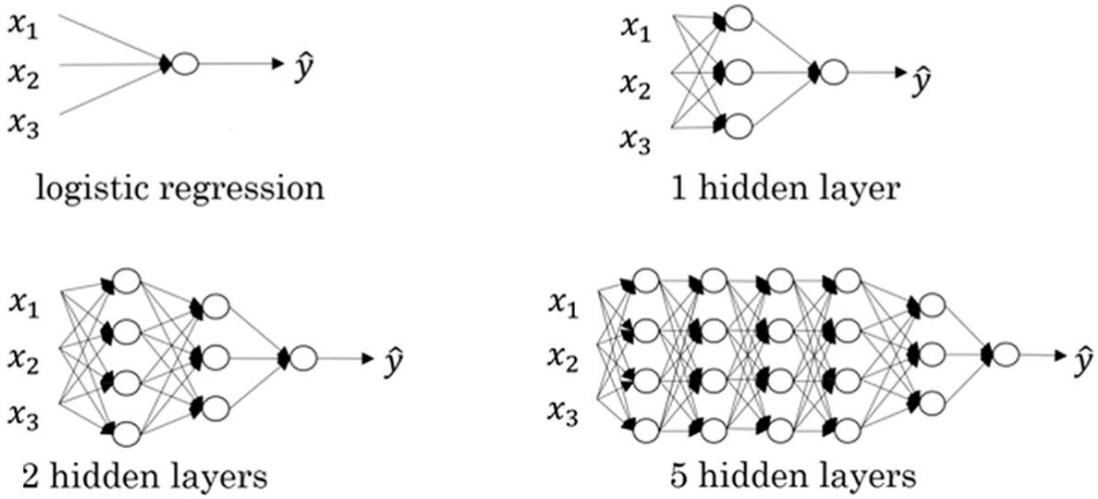
$W^{[1]} = np.random.randn((2,2)) * 0.01$ ， $b^{[1]} = np.zero((2,1))$

$W^{[2]} = np.random.randn((2,1)) * 0.01$ ， $b^{[2]} = 0$

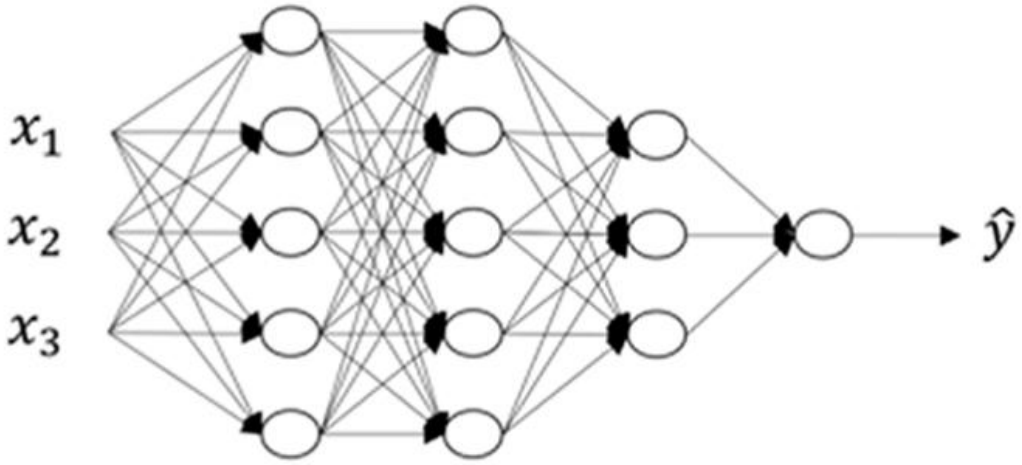
注：权重矩阵乘以一个系数 0.01 是为了防止输出结果过大，使得 sigmoid 函数的斜率趋于 0，增加迭代次数，降低算法效率。

第四周 深层神经网络

4.1 深层神经网络



如上图所示，第二周所学习的 logistic 回归可以视为最“浅”的神经网络，即单层神经网络；第三周则学习了具有单个隐层的双层神经网络。由于一些较为复杂的函数浅层神经网络是无法学习的，所以我们需要更“深”的神经网络，所谓“深层神经网络”指的就是层数较多的神经网络。



接下来我们将以上图为例介绍深层神经网络中的一些术语和符号
神经网络的层数（layers）：L，上图中，L = 4

l 层的神经元个数：$n^{[l]}$,上图中，$n^{[1]}=5,\ldots,n^{[4]}=n^{[L]}=1,n^{[0]}=n_x=3$

4.2 深层网络中的前向传播

神经网络的前向传播实现的思想与浅层神经网络类似，都是通过计算图由前向后进行计算，只是深层网络的深度较高，计算步骤较多，在逐层计算过程中需要用 for 循环遍历各个层，但其大体思路与浅层神经网络还是相同的。以 4.1 节中的神经网络为例，向量化前向传播过程如下：

$Z^{[1]}=W^{[1]}x\quad+b^{[1]}$ ， $A^{[1]}=g^{[1]}(Z^{[1]})$

$Z^{[2]}=W^{[2]}A^{[1]}+b^{[2]}$ ， $A^{[2]}=g^{[2]}(Z^{[2]})$

$Z^{[3]}=W^{[3]}A^{[2]}+b^{[3]}$ ， $A^{[3]}=g^{[3]}(Z^{[3]})$

$Z^{[4]}=W^{[4]}A^{[3]}+b^{[4]}$ ， $A^{[4]}=g^{[4]}(Z^{[4]})$

其基本格式为 $Z^{[l]}=W^{[l]}A^{[l-1]}+b^{[l]}$ ， $A^{[l]}=g^{[l]}(Z^{[l]})$，故可以使用 for 循环对神经网络逐层进行计算，目前而言，这个对神经网络逐层遍历的 for 循环很难去掉。

4.3 核对矩阵的维数

本节中将主要介绍深层神经网络中涉及矩阵的维度与其相应的理解，这样便于对矩阵维度进行检测，避免很多不必要的 bug。

1、权重矩阵 W：

$shape(W^{[l]})=(n^{[l]},n^{[l-1]})$

权重矩阵 W 的维度可以这样理解：W 的每行代表着一个神经元中不同特征的权重，W 的每列代表着不同的神经元。

相应地：$shape(dW^{[l]})=(n^{[l]},n^{[l-1]})$

2、阈值矩阵 b：

$shape(b^{[l]})=(n^{[l]},1)$

阈值矩阵的维度很好理解，每一个神经元对应着一个阈值，所以说阈值矩阵的维度应该是$(n^{[l]},1)$。注意到：输入向量化后，$Z^{[l]}$的维度会发生变化，但利用 python 中的广播，$b^{[l]}$仍可不变。

相应地：$shape(db^{[l]})=(n^{[l]},1)$

3、输出矩阵 A

$shape(Z^{[l]})=(n^{[l]},m)$

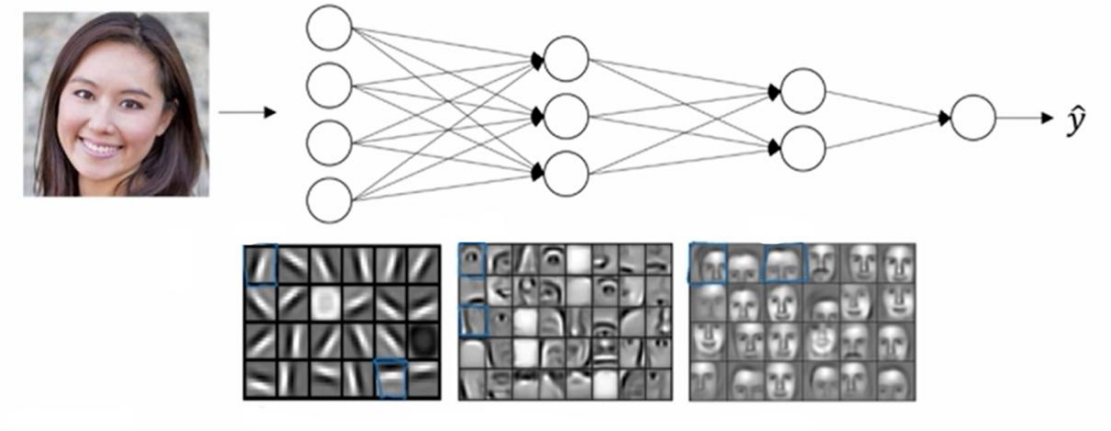
$shape(A^{[l]})=(n^{[l]},m)$

相应地：$shape(dZ^{[l]})=(n^{[l]},m)$　　$shape(dA^{[l]})=(n^{[l]},m)$

了解了各个矩阵的维度，在实现过程中，我们可以将 assert()函数配合

shape()函数使用，进而判断所得的矩阵是否正确。

4.4 为什么使用深层表示

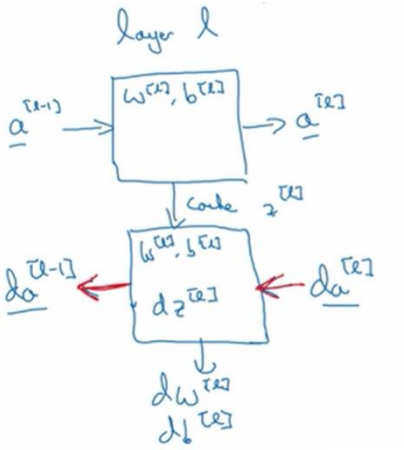


以人脸识别为例，深层神经网络可以直观理解为不同功能由浅入深的叠加，比如第一个隐层可以视为对图片进行边缘提取，第二个隐层可以视为在边缘提取的基础上，将特征分类（人的五官），第三个隐层可以视为将已分类的特征进行组合，进而得到了人脸识别的输出结果。

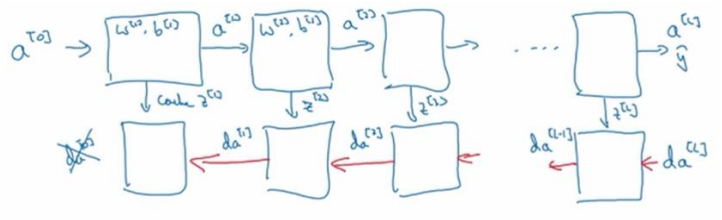
使用深层神经网络的原因在于很多复杂的函数很难利用浅层的神经网络进行拟合，神经网络的每个“层”可以看作是函数的复合，这样经过多次复合后，就比较容易表达较为复杂的函数。其次，神经网络的深度对于提高算法效率也有显著作用，因为在层数不变只能增加神经元个数的前提下，要先成倍地提高算法的效率，神经元个数需要呈指数增长，所以说增加神经网络的层数是提高算法效率比较“经济”的方式。

4.5 搭建深层网络块

对于神经网络的第l层，如右图所示，在正向传播过程中，其输入为上一层的结果$a^{[l-1]}$根据自身的权重$w^{[l]}$和阈值$b^{[l]}$，输出结果$a^{[l]}$并将$z^{[l]}$缓存。在反向传播过程中，其输入为下一层求得关于$a^{[l]}$的导数$da^{[l]}$，根据自身的权重$w^{[l]}$和阈值$b^{[l]}$在计算图中反向计算出$dw^{[l]}$、$db^{[l]}$、$da^{[l-1]}$，并将$da^{[l-1]}$传递给上一层。



将上述基本模块堆叠在一起则得到如下图所示的深层神经网络结构。



4.6 前向和反向传播

本节的主要思路就是按照计算图来正向计算和反向计算，主要的注意事项就是正向传播和反向传播时的初始化问题。

1、第 l 层的前向传播

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \quad , \quad A^{[l]} = g^{[l]}(Z^{[l]})$$

2、第 l 层的反向传播

$$\mathrm{d}Z^{[l]} = \mathrm{d}A^{[l]} * g^{[l]'}\left(z^{[l]}\right) \quad , \quad \mathrm{d}W^{[l]} = \frac{1}{m}\mathrm{d}Z^{[l]} \cdot A^{[l-1]T}$$

$$\mathrm{d}b^{[l]} = \frac{1}{m} np.sum(\mathrm{d}Z^{[l]}, axis = 1, keepdims = True)$$

$$\mathrm{d}A^{[l-1]} = W^{[l]T}\mathrm{d}Z^{[l]}$$

3、初始化

正向传播初始化：$A^{[0]} = X$

反向传播初始化：$dA^{[L]} = \frac{\mathrm{d}L(\hat{y},y)}{\mathrm{d}a}$

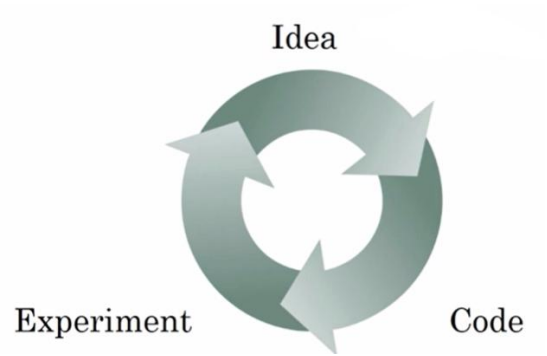
4.7 参数 VS 超参数

参数主要指的是那些会随着算法迭代而改变的量，而超参数主要指的是那些在设计算法之处就被设定的参量，这些参量从一定意义上决定着参数的最终值和算法的效率，下面是一些常见的参数和超参数。

参数：神经网络各层的 W 和 b，即$W^{[l]}$、$b^{[l]}$

超参数：①学习率 α ②迭代次数 iterations ③隐层层数（深度） ④隐层神经元个数 ⑤激活函数

超参数调试：超参数调试是一个高度经验化的过程。很多时候超参数的最优值会随着时间的推移而变化，产生这种变化的原因是多样的，可能是 CPU、GPU、测试集等的变化造成的，这样就需要不断地对超参数进行修改和测试。



4.8 这和大脑有什么关系？

总结一句，可以这样类比，但是实际关系不大，人脑的学习机制至今未知。