摘要

本文对前馈神经网络做了总结：

主要包括以下几个方面：

1、神经网络的介绍

2、代价函数与输出单元

3、激活函数

4、架构设计

5、梯度下降，链式法则，反向传播算法 和计算图

6、向量化

7、使用python搭建深度神经网络

8、前馈神经网络在NLP中的应用

前馈神经网络介绍

深度前馈网络 (deep feedforward network)，也被称作**前馈神经网络** (feedforward neural network)，或者**多层感知机** (multiplayer perceptron **MLP**),是典型的深度学习模型，其目标近似某个函数f^*,将输入映射到输出。之所以被成为**前馈(feedforward)**，是因为模型的输出和模型本身之间没有反馈(feedback)连接，当包含反馈连接时，他们被称为**循环神经网络** (recurrent neural network)。

前馈神经网络被称为网络是因为他们通常由许多不同函数复合在一起来表示，该模型与一个有向无环图关联，该图描述了函数是如何复合在一起的，例如有三个函数f^1, f^2, f^3连接在一个链上，则形成$f(x) = f^3(f^2(f^1(x)))$,链的长度称为**深度** 。每个样本x 对于一个标签y，对于每个样本（x,y）,输出层必须产生一个接近y的值，但是训练样本并没有指明其他层应该怎么做，因此，学习算法必须决定如何使用这些层来最好的实现f^*的近似，训练数据没有给出中间层需要给出的输出，因此被称为**隐藏层**(hidden layer)。

这些网络只所以被称作神经网络，是因为它们受到了神经科学的启发。

理解前馈神经网络的方式是从线性模型开始，并考虑如何克服它的局限性，线性模型，例如逻辑回归和线性回归，都可以高效且可靠的拟合，但线性模型只能模拟线性函数，无法表发两个输入变量之间的相互作用。为了扩展线性模型来表示x的非线性函数，可以将模型使用在一个x经过变换后输入项($\phi(x)$上，

如何选择ϕ

1、使用一个通用的ϕ，例如RBF核，如果ϕ具有足够高的维数，那么我们总是有足够的能力来拟合训练集，但对于测试集上的泛化比较差。通用特征映射只能基于局部光滑的原则，没有足够信息进行编码来进行高级问题。

2、手动设计ϕ,深度学习出现以前，这一直是主流的方法，但这种方法对于每个单独的任务都需要人们数十年的努力，从业者各自擅长特定的领域，并且不同领域之间很难迁移(transfer)。

3、深度学习的策略是学习ϕ，模型：$y = f(x;\theta,w) = \phi(x;\theta)^T w$,有两组参数：1、从一大类函数中学习$\phi$的参数$\theta$,用于将$\phi(x)$映射到y的输出参数w。$\phi$定义了隐藏层，放弃了凸性的方法，但利大于弊，人类专家可以将他们的知识编码进网络来帮助泛化，他们只需要设计那些他们期望能够表现良好的函数族即可，不需要去寻找精确的函数。

训练神经网络与训练其他机器学习方法的差别

神经网络与其他机器模型的最大区别就是神经网络的非线性导致我们感兴趣的代价函数是非凸的，因此几乎都是基于梯度来使代价函数下降。

事实上我们也可以用梯度下降来训练诸如线性回归和SVM，当训练集相当大时，这也是常用的。

代价函数

指导原则： **最大似然原理 + 正则化**，在输出单元中详细说明。

输出单元

用于高斯输出分布的线性单元(回归问题)

$\hat{y} = W^T h + b$,h为隐藏层输出的特征，最大化对数似然等价于最小化均方误差

高斯输出分布模型：$y = W^T x + \epsilon$，$\epsilon \in N(0, \sigma^2)$

$P(\epsilon) = \frac{1}{\sqrt{2\pi}\sigma} exp^{-\frac{\epsilon^2}{2\sigma^2}}$

$P(y|x;W) = \frac{1}{\sqrt{2\pi}\sigma} exp^{-\frac{(y-W^T x)^2}{2\sigma^2}}$

似然函数为：$L(\theta) = \prod_{i=1}^{m} P(y^i|x^i;W) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} exp^{-\frac{(y^i-W^T x^2)^2}{2\sigma^2}}$

对数似然函数为$l(\theta) = \ln L(\theta) = m \ln \frac{1}{\sqrt{2\pi}\sigma} + \sum_{i=1}^{m} -\frac{(y^i-W^T x^i)^2}{2\sigma^2}$

从上式可以看到，最大化对数似然函数$l(\theta)$等价于最小化最小二乘$\sum_{i=1}^{m}(y^i - W^T x^i)^2$

用伯努利输出分布的sigmoid单元（二分类问题）

最大似然等价于交叉熵。下面详细推导logistic 回归，并说明为什么代价函数是交叉熵而不是最小二乘。

logistic 回归数学推导： logistic 回归虽然名为回归，但其实是一种分类算法，这种算法有很多优点，可以直接对分类的可能性进行建模，无需事先假设数据分布，因此，相比于其他分类算法，例如GDA（高斯判别分析，高斯假设），Naive Bayes（朴素贝叶斯，条件独立假设）等算法，避免了假设分布不准确所带来的问题，有更好的鲁棒性。

而且，它不仅预测出类别，还可以近似得到概率预测，这对许多辅助决策的任务很有用。也可以推广为多元分类模型，即softmax模型。

logisitc 模型如下：

$y \in 0, 1$

$x \in R^n$

$h_W = \frac{1}{1+e^{W^T x}}$

$P(y=1|x;W) = h_W$

$P(y=0|x;W) = 1 - h_W$

W为需要学习的参数

总结上式得到：$P(y|x;W) = h_W^y (1-h_W)^{1-y}$

最大似然有： $L(W) = P(y|x;W) = \prod_{i=1}^{m} h_W(x^i)^{y^i} (1 - h_W(x^i))^{1-y^i}$

取对数有：$l(\theta)=\sum_{i=1} m(y^i \log h_W(x^i)+(1-y^i)\log^{1-h_W(x^i)})$

最大化似然函数，等价于最小化负的似然函数，因此，cost函数为：
$-l(W)=-\sum_{i=1} m(y^i \log h_W(x^i)+(1-y^i)\log^{1-h_W(x^i)})$

利用梯度下降法可知$W=W-\alpha\nabla_W(-l(W))$

具体的关于cost函数的推导过程比较长，限于时间，就不在这里详细论述，具体可参见吴恩达斯坦福机器学习课程附材料，这里只说明大概思路，sigmoid函数的导数 a(1-a),cost 函数中存在log(a)与log(1-a)，求导后，a和1-a会到分母上，根据链式法则，两者相乘，经过化简，可以得到如下公式：
$\frac{\alpha}{\alpha W_j}(-l(W))=-\sum i=1m(y^i-h_W(x^i))x_j^i$

从而可实现对参数的更新：$W_j=W_j+\alpha\sum_{i=1} m(y^i-h_W(x^i))x_j^i$

为什么cost函数是交叉熵而不是最小二乘：

上面的方法，从理论上说明了，为什么logistic使用交叉熵，而不是最小二乘，但是还不够直观，接下来给出一种更加易于理解的思路： 假设损失函数为最小二乘，即，$L(W)=\frac{1}{2}\sum_{i=1}^{m}(y^i-h_W x^i)$

求导可得:$\frac{\alpha L}{\alpha W}=2\sum_{i=1}^{m}(h_W x^i-y^i)h_W x^i(1-h_W x^i)x$（注：（ sigmoid函数的导数是a(1-a) ））

分析上式，假设y^i 等于1，而$h_W x^i$ 等于0，此时目标与预测值之间有很大误差，但是导数为0，无法实现优化，同理当y^i 等于0，$h_\theta x^i$等于1，1-$h_W x^i$ 等于0，那么loss函数也等于0,因此，当标记与预测差别很大时，也无法实现优化。

用于Multinoulli输出分布的softmax单元(多分类问题)

softmax可以看作是sigmoid的扩展

为了推广具有n个值的离散型变量的情况，创造一个向量$\hat{y}$,它的每个元素是$\hat{y}_i=P(y=i|x)$,要保证每个$\hat{y}_i$介于0到1之间，还要保证他们之和为1

从广义线性模型可以推导出softmax 函数的形式为：$softmax(z_i)=\frac{exp(z_i)}{\sum_j exp(z_j)}$

最大化对数似然softmax的输出值：$logsoftmax(z_i)=z_i-log\sum_j exp(z_j)$,第一项鼓励$z_i$增大，第二项鼓励其它项减小。并且$z_j$越大，则有更大的惩罚。

从神经科学角度看，softmax是一种在参与其中的单元之间形成竞争的方式，**赢者通吃**

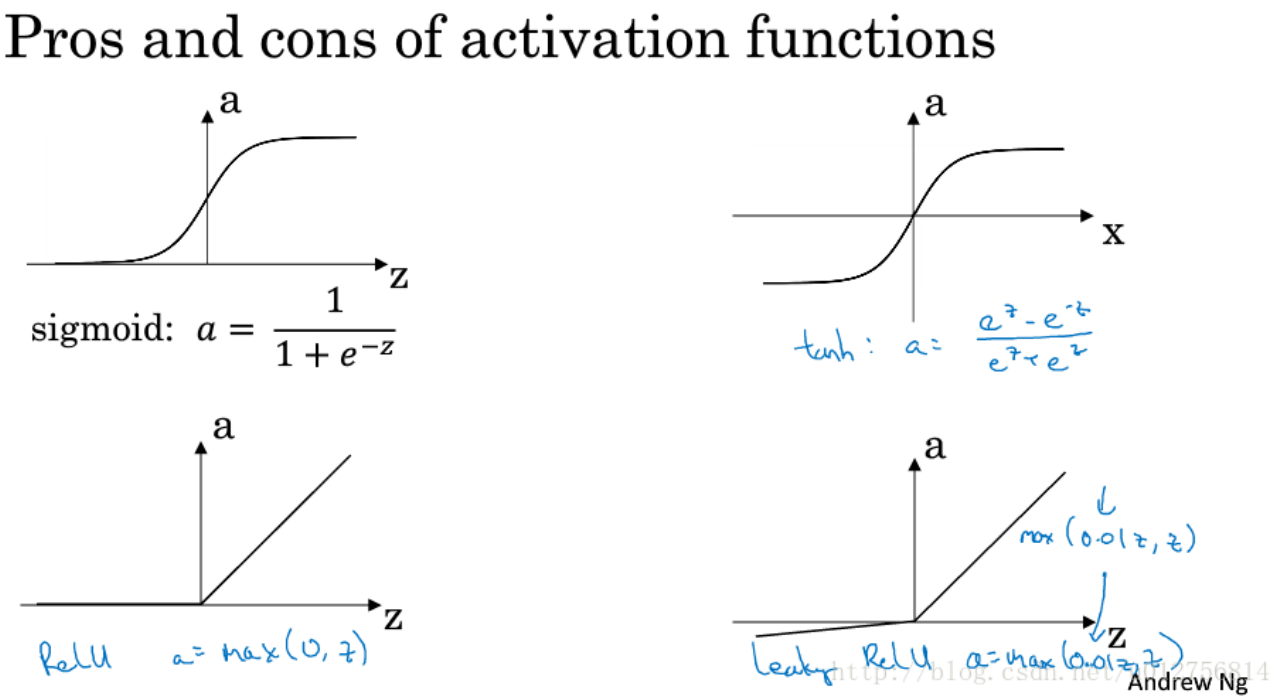
其他输出类型

线性，sigmoid，softmax输出单元是最常见的，神经网络可以推广到我们希望的任何种类的输出层，最大似然原则给如何设计代价函数提供了指导。

事实上每一种cost函数都可以添加正则化项，（在正则化章节中会有详细说明）这里就不再介绍。

激活函数

常见的激活函数有：sigmoid，tanh，relu，leaky Relu，maxout，方程和图像如下图所示。



sigmoid，tanh函数分别将数据压缩到[0,1]和[-1,1],构建深层网络时，进行反向传播时，梯度中会乘上每一层的值，比如：

$$dW^l = \frac{\partial err}{\partial W^l} = \frac{1}{m} dZ^l A^{(l-1)T}$$

这样在构建深层网络时，进行多次压缩，传播到前边的层时，梯度会非常小，出现梯度消失的问题，即gradient vanish，因此，现在一般不常用。sigmoid函数只有在进行二分类问题时（需要将结果输出为概率，压缩到[0,1]）在输出层使用。tanh函数虽然比sigmoid函数在大多数情况下比较好，但是也存在梯度消失的问题。

Relu函数是目前使用最广泛的激活函数，在Z小于0时，输出0，在Z大于0时，输出Z，这样即实现了非线性，也不会出现梯度消失的问题，事实上，在吴恩达老师对Hittion的采访中，也提到了Relu，Relu虽然很简单，但是有严格的数学证明Relu相比sigmoid和tanh的优势（这个没有研究过，就不在这里说明）。

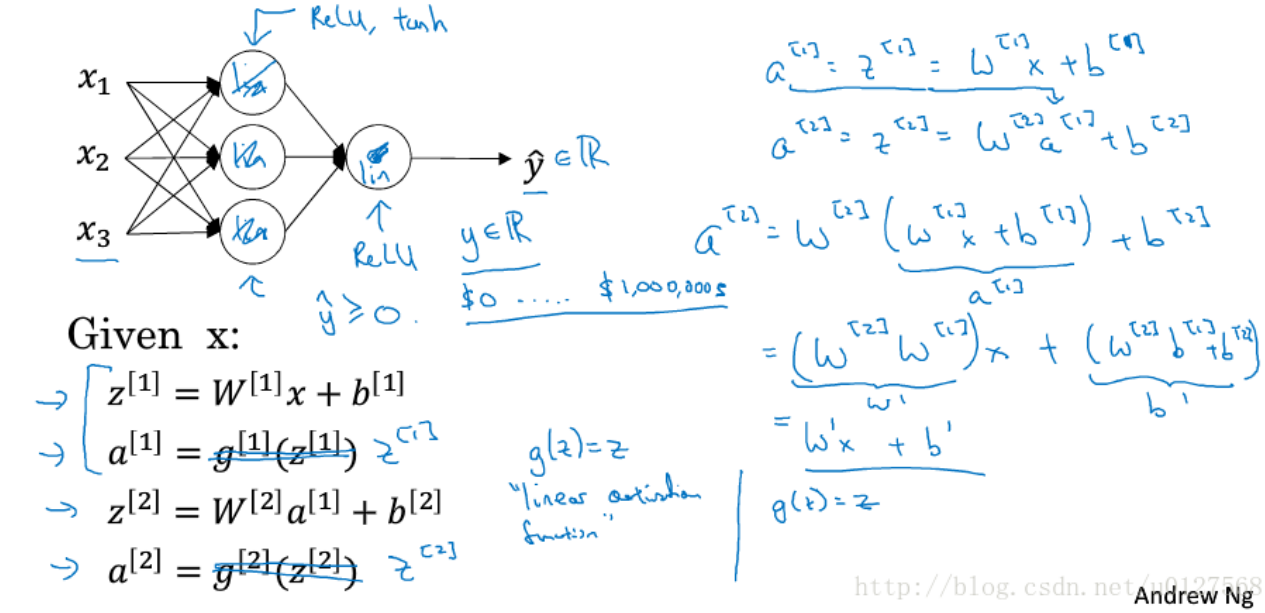
leaky Relu是Relu函数的进阶版，它允许在Z小于0出现一些梯度，事实上，还存在其他各种类型的Relu的变形，这里就不详述了。

maxout 单元可以学习多段的分段线性的凸函数。

softplus：$log(1+e^a)$，ReLU的平滑版本，不鼓励使用。

事实上，关于激活函数的选择依然是一个开放的和依赖情况的问题，需要根据具体业务需求进行选择。

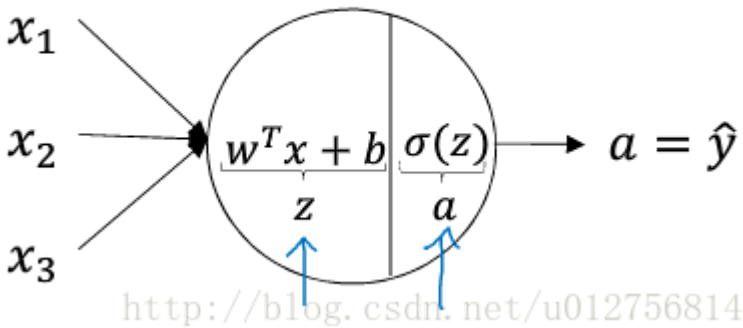
为什么要是有非线性函数作为激活函数 ：

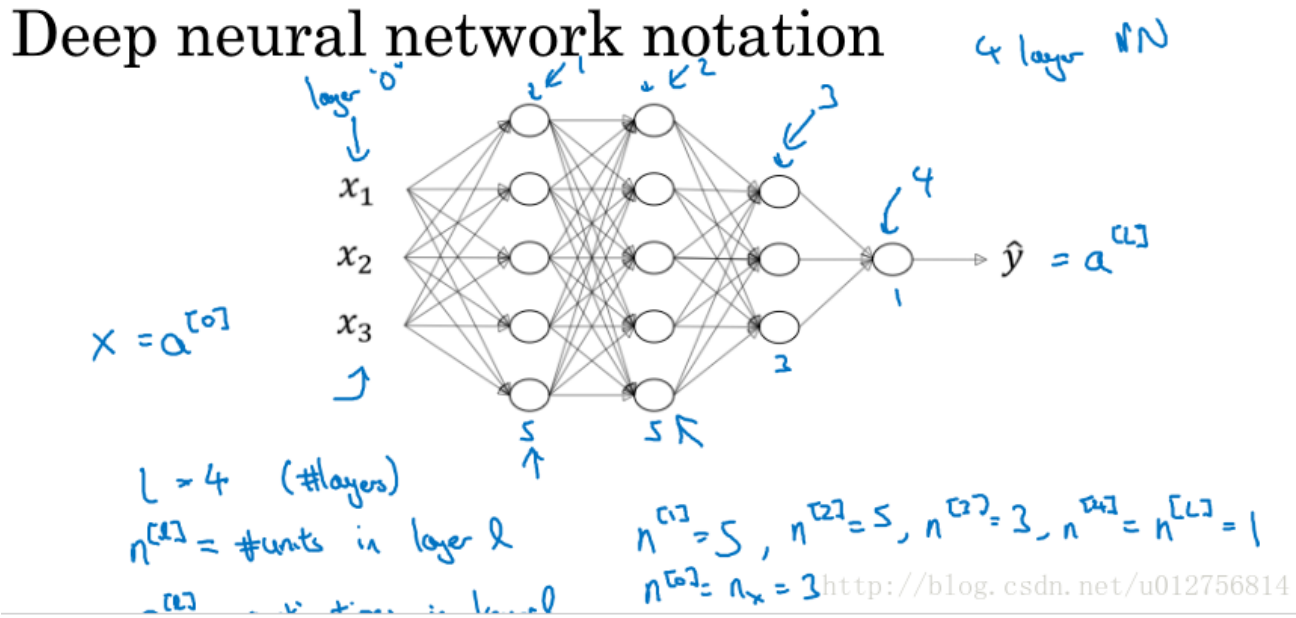


吴恩达老师在课程中做了证明（见上图），如果使用线性函数作为激活函数，即使使用多层神经网络，构建的也只能得到输入的线性函数，只能得到线性的假设空间，与单层线性输出的结构，定义的是相同的假设空间，模型复杂度与单层线性网络一致。无法描述复杂的函数。

架构设计

神经元模型 ：

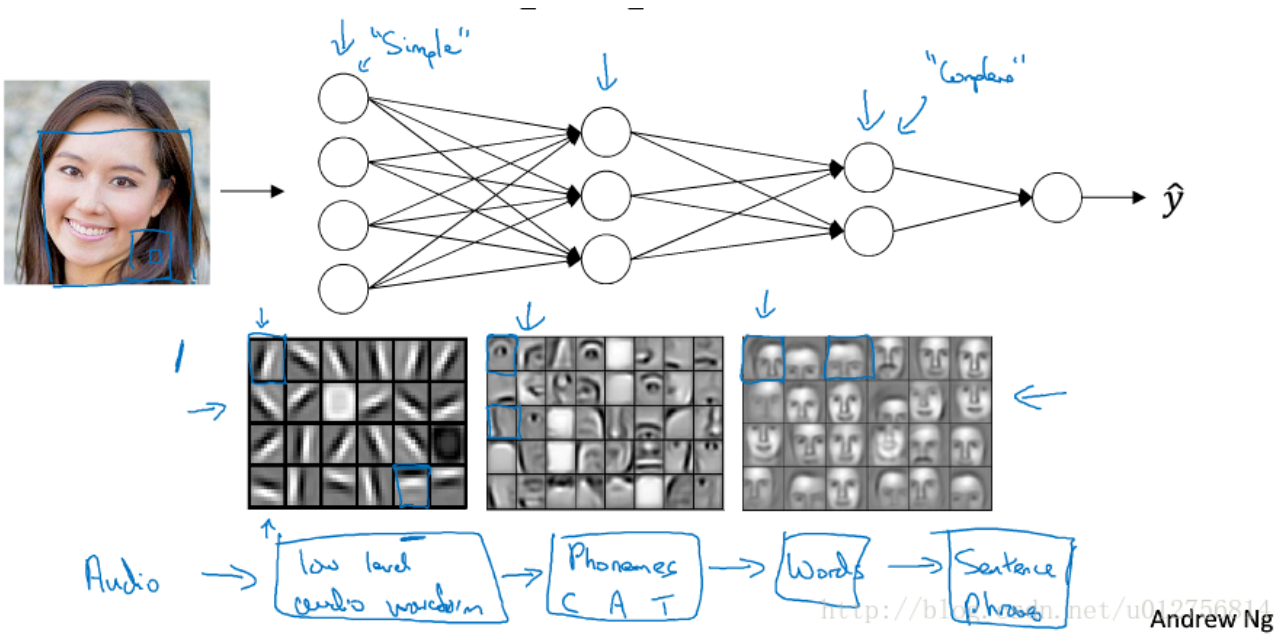




上边两张图说明了前馈神经网络的基本结构，图1为单个神经元模型，输入层加权叠加，加上偏置值，然后经过激活函数，进行输出，可以作为下一层的输入或最终的输出，图2是一个4层深度神经网络（输入为第0层，不计入层数），图中每个圆圈代表一个神经元（每一个连接代表一个需要学习的权值，偏置项没有显示在图中。可以想象在每一层的输入都有一个值为1的项指向下一层的每个神经元。偏置项也需要学习得到），给定一个网络结构，实际上就是给定了一个假设空间。

前馈神经网络的架构考虑是网络的深度（多少层）和每一层的宽度（每一层神经元的个数）

神经网络到底实现了什么？



当我们将一个训练好的神经网络训练好之后，输入一个data，然后将每一层的结果，进行输出，可以明显看到，每一个隐藏层都像一个特征抽取器，一层一层的由简及繁抽取资料的模式，并将这些模式作为更高级的特征，然后将这些更高级的特征交给输出层，使用相对简单的函数，即可实现分类或者其他任务。常规机器算法都需要人为设计特征，但特征工程非常重要但又难设计，也是制约人工智能发展的瓶颈（例如语音识别和图像识别特征难以设计，因此深度学习优势非常明显，但NLP领域特征，相对容易抽取，因此优势并不是十分明显）。

神经网络将问题从如何抽取特征转变为如何设计网络结构的问题。但是网络结构的设计也是一个非常开放的问题，需要设计多少隐藏层，每一层需要多少个神经元，没有定论，只能通过通过不断的试验和直觉，经验，硬件条件等。另外，神经元之间的连接方式也是一个开放的问题，比较经典的模型包括卷积神经网络，循环神经网络，递归神经网络等等。

万能近似理论

一个前馈神经网络如果具有线性输出层和至少一种'挤压'性质的激活函数(例如sigmoid)的隐藏层，那么给予网络足够多数量的隐藏单元，它可以以任意精度来近似从一个有限维空间到另一个有限维空间的可测函数，也可以模拟任何函数的导数。

为什么一定要深？

上一节内容说明了，神经网络将特征抽取问题转变为网络架构的问题，万能近似理论给出只要有足够多数量的隐藏单元，就可以模拟任何可测函数，那么为什么一定要深呢？

关于这个问题。我认为可以总结为如下两个原因：1、参数耦合 2、问题解耦，下面就这两方面进行一些论述：

虽然我们处于一个大数据时代，但仍然面临着数据不充分的问题，尤其是带有标记的数据，因此，如何有效的避免overfitting 仍然是一个非常重要的问题，事实上上文关于为什么一定要深的总结，都是为了减弱overfitting影响。许多人认为，深度神经网络，参数量巨大，因此容易overfitting，事实上，参数量巨大，是因为我们面对的问题太过复杂，必须用大量的参数进行描述，相比于常规的机器学习算法（例如决策树，HMM，矩阵分解等），深度神经网络所需要的参数要少的多，事实上更不容易overfitting（会在以后的文章中进行论述）。

假设采用相同的神经元数目（相同的参数量），宽胖的模型，参数之间的相关性差，而瘦高（deep）模型，参数之间相互耦合，因此有更丰富的表达能力，因此瘦高模型使用比较少的参数，即可实现复杂的模型，有效减少overfitting的影响。

deep 神经网络将一个大问题逐层分解为一个个小问题，使问题逐渐解耦，而分解后的每个小问题，只需要少量的数据，即可训练好，因此需要更少的数据，减少overfitting的影响。

梯度下降，链式法则，反向传播算法 和计算图

梯度下降法 ：

梯度下降法（gradient descent）是求解无约束最优化问题的一种最常用方法，是实现神经网络训练（反向传播）的基础。

假设f(x)是具有一阶连续偏导数的函数(实际应用中并不十分严格，比如relu函数)，最优化的问题可以表示为：

$$x^* = min_{x \subseteq R^n} f(x)$$

梯度下降法是一种迭代算法，选取适当的初值，不断迭代，更新参数值，进行目标函数的最小化，直到收敛，负梯度方向是使函数值下降最快的方向，在迭代的每一步，以负梯度方向更新参数的值，从而达到最小化参数值的目的。

f(x)具有一阶连续导数，若第k次迭代值为xk ,根据一阶泰勒展开将f(x)在xk 处展开，有如下公式：
f(x)=f(xk)+gk(x−xk),这里gk为f(x)在xk处的梯度，通过迭代可以得到：

$$x^{k+1} \leftarrow x^k - \alpha * g_k$$

当目标函数为凸函数时，梯度下降法的解释全局最优解(比如logistic regression)，但一般情况下，其解不保证是全局最优解，而且梯度下降法的收敛速度也未必很快，但是深度神经网络训练过程，反向传播采用梯度下降算法，因此本文只简要说明梯度下降。

链式法则 ：

在论述链式法则之前，先做一些符号上的说明， 假设网络有L层， 第l层的神经元个数为dl 第l层,第i个神经元的线性加权值为zli,经过激活函数后的输出值为ali， 第l-1层第i个元素，连接到第l层第j个元素的权值为

$$W_{j,i}^l$$

第l-1层到第l层第j个元素的偏置值为blj 假设a0i=xi

正向传播过程：

正向传播公式如下公式：

$$z_j^l = \sum_{i=1}^{d^{l-1}} W_{j,i}^l * a_i^{l-1} + b_j^l$$

$$a_j^l = actf(z_j^l)$$

（其中actf为激活函数）

根据输出al与标签y，计算cost函数err

反向传播过程：

计算初始

$$\frac{\partial err}{\partial a_j^L}$$

假设cost函数为err(X,y,W,b),简记为err，我们的需要求取err关于每一个

$$W_{j,i}^l$$

和blj的梯度，从而最小化err。

根据链式法则有：

$$\frac{\partial err}{\partial W_{j,i}^l} = \frac{\partial err}{\partial {a_j}^l} * \frac{\partial {a_j}^l}{\partial z_j^l} * \frac{\partial z_j^l}{\partial W_{j,i}^l}$$

$$\frac{\partial err}{\partial b_j^l} = \frac{\partial err}{\partial {a_j}^l} * \frac{\partial {a_j}^l}{\partial z_j^l} * \frac{\partial z_j^l}{\partial b_j^l}$$

其中：

$$\frac{\partial z_j^l}{\partial W_{j,i}^l} = a_i^{l-1}$$

$$\frac{\partial {z_j}^l}{\partial {b}_j^l} = 1$$

$$\frac{\partial {a_j}^l}{\partial z_j^l}$$

为激活函数导数， 如果激活函数为relu：ifzlj<0,梯度为0，否则，梯度为1， 如果激活函数为leaky relu：ifzlj<0,梯度为0.01，否则，梯度为1， 如果激活函数为sigmoid函数，梯度为alj∗(1−alj) 如果激活函数为tanh函数，梯度为1−alj∗alj

$$\frac{\partial err}{\partial {a_j}^l} = \sum_{k=1}^{d^{l+1}} \frac{\partial err}{\partial z_k^{l+1}} * \frac{\partial z_k^{l+1}}{\partial {a_j}^l} = \sum_{k=1}^{d^{l+1}} \frac{\partial err}{\partial z_k^{l+1}} * W_{k,j}^{l+1}$$

为了与课程中的符号和后续实现保持一致，做如下约定：

dalj =

$$\frac{\partial err}{\partial a_j^L}$$

$$dz_j^l = \frac{\partial err}{\partial z_j^l} = da_j^l * \frac{\partial {a_j}^l}{\partial {z_j}^l}$$

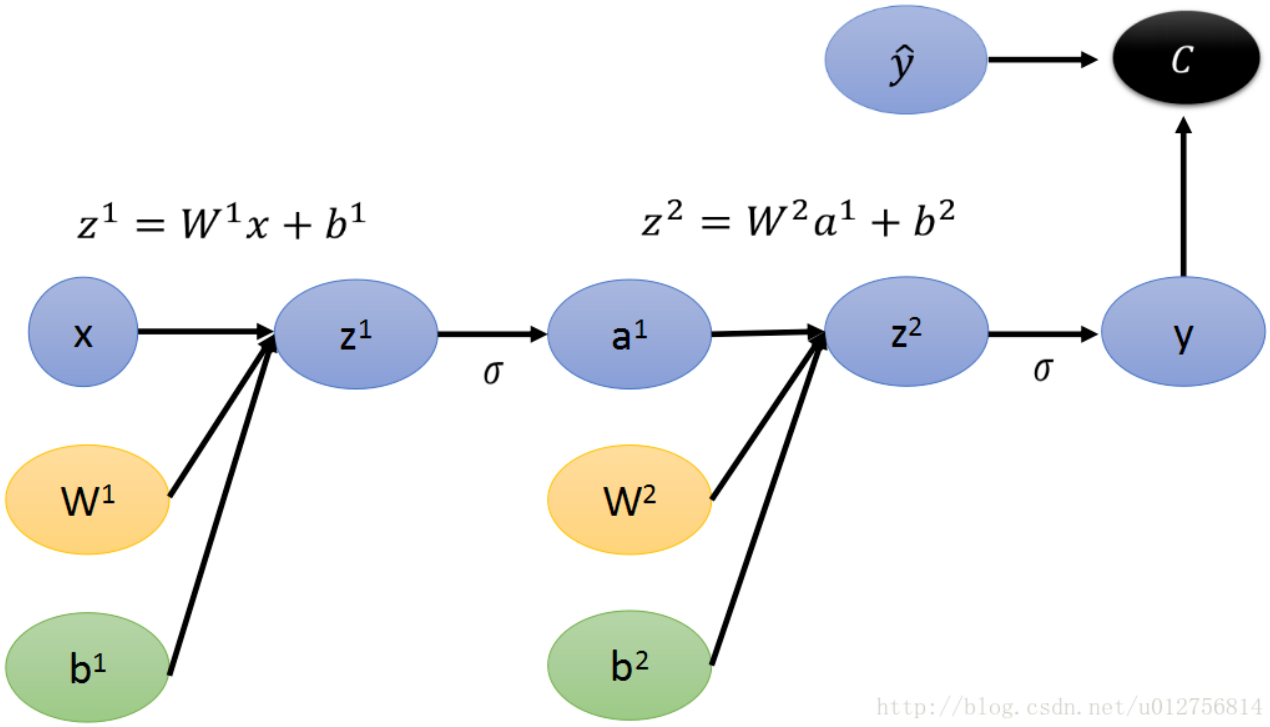
$$d{W_{j,i}}^l = \frac{\partial err}{\partial W_{j,i}^l} = d{z_j}^l * a_i^{l-1}$$

$$d{b_j}^l = \frac{\partial err}{\partial b_j^l} = dz_j^l$$

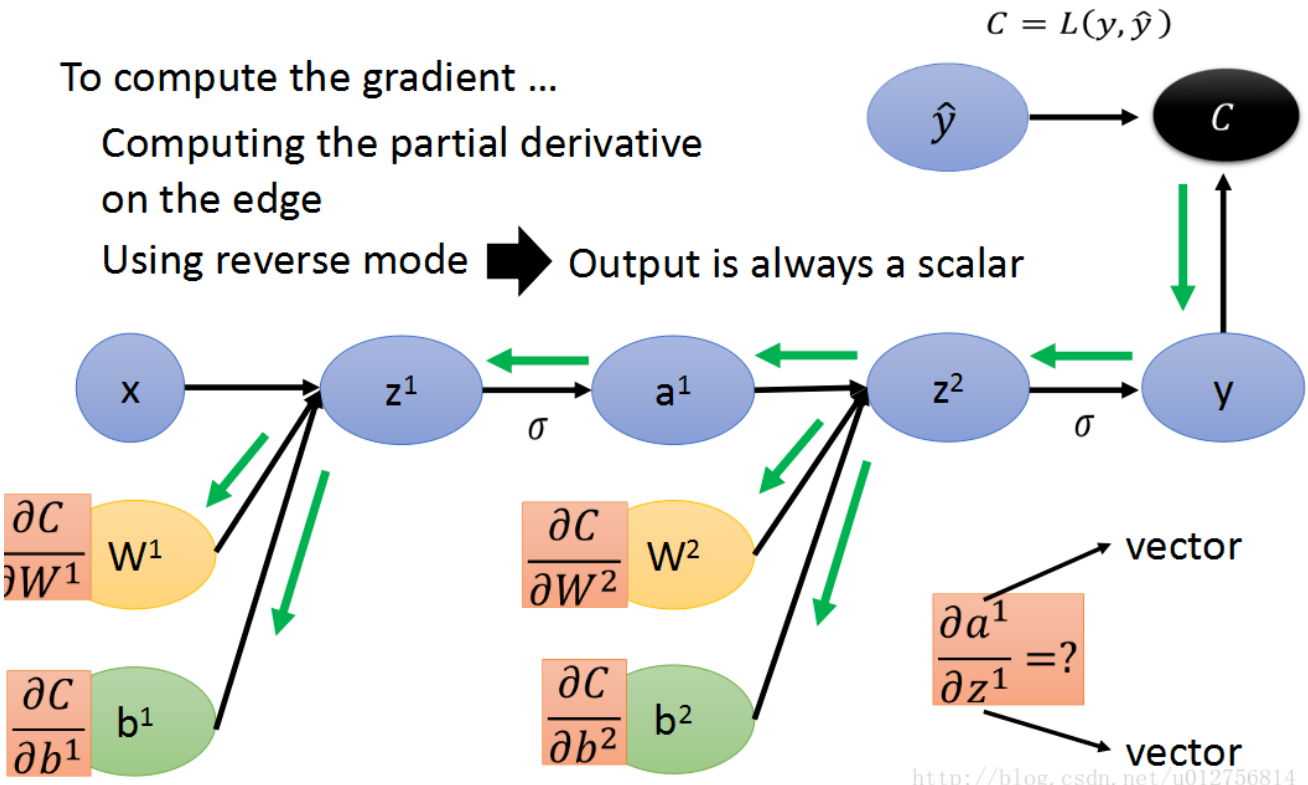
反向传播算法 ：

1、初始化所有权重：$W_{j,i}^l$和blj 2、for i in range(T): (1):前向传播(forward),计算损失函数err (2):反向传播(backword)，计算err关于$W_{j,i}^l$和blj 的导数$d{W_{j,i}}^l$ 和dblj (3)：更新权值 $W_{j,i}^l$ =$W_{j,i}^l$ - learning_rate *$d{W_{j,i}}^l$ blj = blj - learning_rate*dblj

计算图 ： 上文对反向传播算法做了详细的描述，但还是不够直观，而计算图(computational Graph)是用图来描述变量之间相关操作的一种语言，能够用图形化的方式，清晰的描述变量之间的相关关系，计算图中，每个节点代表一个变量，变量可以为标量，向量，矩阵，张量。同时也定义了变量之间的操作，通过这些变量和操作来描述整个复杂的模型。



正向传播



反向传播

上边两张图分别是正向传播和反向传播的计算图表示

正向传播时，从输入节点开始（包括输入数据和参数），按照计算图中定义的操作，从前往后进行计算，每个节点的值只与其直接相连的输入节点有关，每次只需执行简单的操作，即可实现正向传播。

反向传播时，根据链式法则，从后往前，记录cost函数对每个节点的偏微分，计算每个节点的偏微分时，只需用cost函数关于之相连的输出的节点变量的偏导数乘以该与之相连的输出节点变量关于本节点变量的导数即可，如果有一个节点有多个输出，分别先乘，然后相加即可，而不用关心与之没有相连的节点变量，只需执行一些非常简单的函数，即可实现反向传播。

事实上，大部分的深度学习框架tensorflow，CNTK等实现反向传播算法时，都使用计算图进行计算。

向量化（Vectorization）：

上述反向传播算法的描述，都是对单个参数进行分析，如果全部采用for循环，计算效率就会非常低下，CPU和GPU中都是实现了SIMD(single instructions multiple data)指令，尤其GPU。采用矩阵计算的方式，可以非常快速的实现计算，大大提高效率，因此，如果可以不用循环，一定不用循环，采用矩阵的方式进行操作。

编程技巧：采用向量化的方式，一定要注意数据的维度，通过分析数据的shape，可以有效避免编程过程中可能出现的bug。

(1)通过np.shape获取数据维度信息 (2)采用np.reshape进行维度转换 (3)进行求和，求均值等操作时，要指定在哪个维度上进行操作，同时指明keep_dims=True,保持原来数据的维度 (4)多使用assert等对数据维度进行断言 (5)使用np.squeeze去掉多余的空维度 (6)避免出现(n,)这样的向量(即不是行向量，也不是列向量，操作容易出现混乱) (7)注意python中的广播

在对神经网络的实现中，对数据和参数维度做如下约定：

神经网络中每一层中的数据(包括输入和输出层)，每一列代表一个数据，将每个数据在列方向堆叠并排构成矩阵，其中nx为每个数据的维度，m为数据量。对W,b的维度约定如下：

Al : [nl ,m]

$$W^l : [n^l, n^{l-1}]$$

bl :[nl,1]

下面我们用向量化的方式实现，上述前向和反向传播过程。

正向传播： linear-forward:

$$Z^l = W^l A^{l-1} + b^l$$

numpy中使用np.dot() 和np.add()实现

linear_activation_forward : Al=actf(Zl) L层 Model：从前往后，执行L次

反向传播： Linear backward：

$$dW^l = \frac{\partial err}{\partial W^l} = \frac{1}{m} dZ^l A^{(l-1)T}$$

$$db^l = \frac{\partial err}{\partial b^l} = \frac{1}{m} \sum_{i=1}^{m} dZ_i^l dA^{l-1} = \frac{\partial err}{\partial A^{l-1}} = W^{lT} dZ^l$$

Linear-Activation backward:

$$dZ^l = dA^l * g^{'}(Z^l)$$

L层 Model：从后往前，执行L次

更新权值：

Wl=Wl - learning_rate *dWl bl=bl - learning_rate * bl

反向传播过程出现的矩阵转置做一个简单的阐述

采用向量化的方式进行反向传播时，出现了两个转置，课程中讲主要考虑维度上的一致的问题，接下来，我将尝试进一步阐释为什么会出现转置。

先来分析：

$$dA^{l-1} = \frac{\partial err}{\partial A^{l-1}} = W^{lT} dZ^{l}$$

单个值的梯度如下：

$$\frac{\partial err}{\partial a_j{}^{l}} = \sum_{k=1}^{d^{l+1}} \frac{\partial err}{\partial z_k{}^{l+1}} * \frac{\partial z_k{}^{l+1}}{\partial a_j{}^{l}} = \sum_{k=1}^{d^{l+1}} \frac{\partial err}{\partial z_k{}^{l+1}} * W_{k,j}^{l+1}$$

为了符号一致，做如下替换：用l-1替换l，用l替换l+1，用i替换j，用j替换k，并将右边标量相乘的顺序交换，从而有$\frac{\partial err}{\partial a_i{}^{l-1}} = \sum_{j=1}^{d^{l}} W_{j,i}{}^{l} \frac{\partial err}{\partial z_j{}^{l}}$可以看到加权相加时，列固定，在不同行方向上移动，因此为了使用矩阵计算，需要将矩阵进行装置。 从而得到： $dA^{l-1} = \frac{\partial err}{\partial A^{l-1}} = W^{lT} dZ^{l}$

同理对于：

$$dW^{l} = \frac{\partial err}{\partial W^{l}} = \frac{1}{m} dZ^{l} A^{(l-1)T}$$

单个值的梯度如下：

$$dW_{j,i}{}^{l} = \frac{\partial err}{\partial W_{j,i}^{l}} = dz_j{}^{l} * a_i{}^{l-1}$$

为了符号一致，做一下替换，用i表示j，用j表示i

从而有$dW_{i,j}{}^{l} = dz_i{}^{l} * a_j{}^{l-1}$这里只是一个数据的对W的影响，全部数据公式可以改写为：

$dW_{i,j}{}^{l} = \frac{1}{m} * dz_i{}^{l} * \sum_{k=1} a_{k,j}{}^{l-1}$可以看到，式子中是对后一项中每一行数据进行乘法，因此，需要将矩阵进行转置， 从而有$dW^{l} = \frac{\partial err}{\partial W^{l}} = \frac{1}{m} dZ^{l} A^{(l-1)T}$

使用python搭建深度神经网络

接下来我们使用python来搭建深层的神经网络（这里以分类任务为例）： 整体框架： **定义网络架构**

初始化所有权值和偏置

实现正向传播

实现前向传播的线性部分，即求Zl : (

$$Z^{l} = W^{l} A^{l-1} + b^{l}$$

)

定义激活函数

定义linear-activation输出：Al=actf(Zl)

执行L次循环，从前向后依次传播，得到AL 根据AL 和 Y,计算cost函数 **实现反向传播**

实现反向传播线性部分： Linear backward：$dW^l = \frac{\partial err}{\partial W^l} = \frac{1}{m} dZ^l A^{(l-1)T}$

$db^l = \frac{\partial err}{\partial b^l} = \frac{1}{m}\sum_{i=1}^{m} dZ_i^l dA^{l-1} = \frac{\partial err}{\partial A^{l-1}} = W^{lT} dZ^l$ 定义激活函数的梯度函数： 定义linear-activation 反向传播部分： Linear-Activation backward: $dZ^l = dA^l * g'(Z^l)$ 从后往前，执行L次反向传播过程： 整体过程入下图所示 **更新权值和偏置值** Wl=Wl - learning_rate *dWl bl=bl - learning_rate * bl

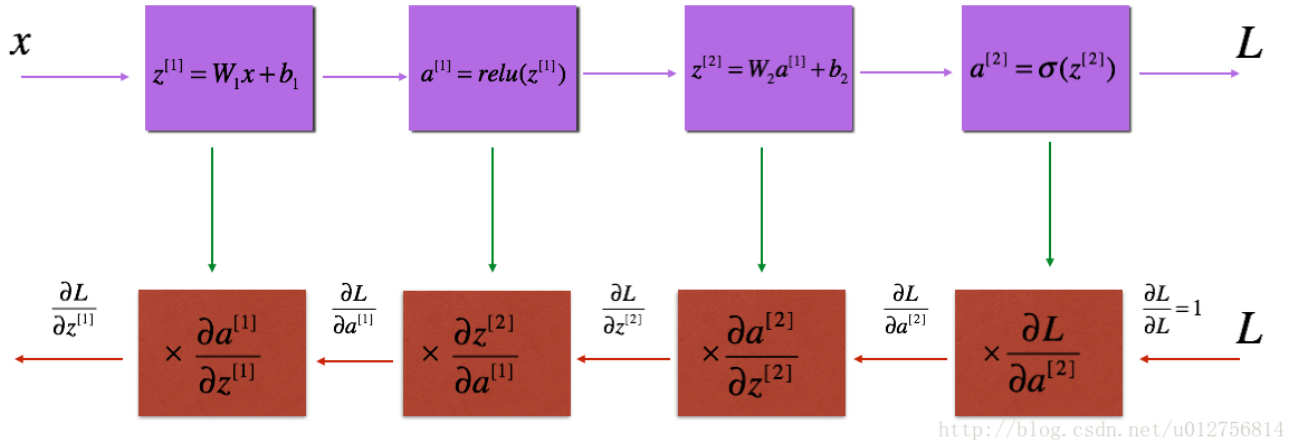


Figure 3 : Forward and Backward propagation for LINEAR->RELU->LINEAR->SIGMOID The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

下面对上面提到的每一步进行具体实现：

1、定义网络共有L层，输入数据维度为12288，数据量为209，输出为{0,1}，1到L-1的激活函数为relu，最后一层为了实现二分类，使用sigmoid函数。

2、初始化网络参数，参数维度如下：

3.2 - L-layer Neural Network

The initialization for a deeper L-layer neural network is more complicated because there are many more weight matrices and bias vectors. When completing the `initialize_parameters_deep`, you should make sure that your dimensions match between each layer. Recall that $n^{[l]}$ is the number of units in layer l. Thus for example if the size of our input X is $(12288, 209)$ (with $m = 209$ examples) then:

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]} X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
⋮	⋮	⋮	⋮	⋮
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]} A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

使用循环逐层初始化参数，定义initialize_parameters_deep(layer_dims)：其中layer_dims为定义的网络的维度列表。比如，定义一个3层神经网络，第一层到第三层节点数分别为5,4,3,则layer_dims输入[5,4,3]。关于函数初始化的方法，下一篇文章中会详述，这里就不再详述。 函数体如下：

```
# GRADED FUNCTION: initialize_parameters_deep

def initialize_parameters_deep(layer_dims):
    """
    Arguments:

    layer_dims -- python array (list) containing the dimensions of each layer in our network
```

```
    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])*0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l],1))
        ### END CODE HERE ###

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters12345678910111213141516171819202122232425262728
```

实现前向传播的线性部分，即求Zl : (

$$Z^l = W^l A^{l-1} + b^l$$

)

定义linear_forward(A,W , b)

```
# GRADED FUNCTION: linear_forward

def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of previous layer, number of
examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation parameter
    cache -- a python dictionary containing "A", "W" and "b" ; stored for computing the backward
pass efficiently
    """

    ### START CODE HERE ### (≈ 1 line of code)
    Z = np.dot(W,A) + b
    ### END CODE HERE ###

    assert(Z.shape == (W.shape[0], A.shape[1]))

    cache = (A, W, b)
```

```
    return Z, cache12345678910111213141516171819202122232425
```

定义激活函数（这里实现sigmoid和Relu函数）：

```
def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the backward pass
efficiently
    """

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache12345678910111213141516171819202122232425262728293031323334435
```

定义linear-activation 前向传播： linear_activation_forward(A_prev, W, b, activation)

```
# GRADED FUNCTION: linear_activation_forward

def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:

    A_prev -- activations from previous layer (or input data): (size of previous layer, number
```

```
of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or
"relu"

    Returns:
    A -- the output of the activation function, also called the post-activation value
    cache -- a python dictionary containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### (≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev,W,b)
        A, activation_cache = sigmoid(Z)
        ### END CODE HERE ###

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### (≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev,W,b)
        A, activation_cache = relu(Z)
        ### END CODE HERE ###

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache12345678910111213141516171819202122232425262728293031323334353 6
```

定义L次前向传播循环(1-L-1层使用Relu激活函数，第L层使用sigmoid激活函数)： L_model_forward:

```
# GRADED FUNCTION: L_model_forward

def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of them, indexed from 0 to
L-2)
                the cache of linear_sigmoid_forward() (there is one, indexed L-1)
    """

    caches = []
    A = X
```

```
    L = len(parameters) // 2                  # number of layers in the neural network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
        A_prev = A
        ### START CODE HERE ### (≈ 2 lines of code)
        A, cache =
linear_activation_forward(A_prev,parameters['W'+str(l)],parameters['b'+str(l)],'relu')
        caches.append(cache)
        ### END CODE HERE ###

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    ### START CODE HERE ### (≈ 2 lines of code)
    AL, cache =
linear_activation_forward(A,parameters['W'+str(L)],parameters['b'+str(L)],'sigmoid')
    caches.append(cache)
    ### END CODE HERE ###

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches12345678910111213141516171819202122232425262728293031323334353637 38
```

计算cost函数： 对于二分类问题，cost函数为交叉熵，这个在logistic部分做了比较详细的论述，这里只列出公式：

$$err = -\frac{1}{m}\sum_{i=1}^{m}(y^i log^{a_i{}^L} + (1-y^i) log^{1-a_i{}^L})$$

只是使用np.multiply实现元素相乘，使用np.sum 实现求和，cost为标量，使用np。squeeze去掉多余的维度。

```
# GRADED FUNCTION: compute_cost

def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (1, number of
examples)
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number
of examples)

    Returns:
    cost -- cross-entropy cost
    """

    m = Y.shape[1]

    # Compute loss from aL and y.
    ### START CODE HERE ### (≈ 1 lines of code)
    cost = -1/m*np.sum((np.multiply(Y,np.log(AL))+np.multiply((1-Y),np.log(1-AL))))
    ### END CODE HERE ###

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g. this
```

```
turns [[17]] into 17).
    assert(cost.shape == ())

    return cost12345678910111213141516171819202122232425
```

定义反向传播线性部分：

```
# GRADED FUNCTION: linear_backward

def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation in the current
layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1),
same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    ### START CODE HERE ### (≈ 3 lines of code)
    dW = 1/m * np.dot(dZ,A_prev.T)
    db = 1/m * np.sum(dZ,axis=1,keepdims=True)
    dA_prev = np.dot(W.T,dZ)
    ### END CODE HERE ###

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db1234567891011121314151617181920212223242526272829
```

定义激活函数的梯度函数：

```
def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z

    """
```

```
    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)

    return dZ12345678910111213141516171819202122232425262728293031323334353637383940414142
```

定义反向传播linear-activation 部分：

```
def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store for computing backward
propagation efficiently
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or
"relu"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1),
same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    linear_cache, activation_cache = cache
```

```
    if activation == "relu":
        ### START CODE HERE ### (≈ 2 lines of code)
        dZ = relu_backward(dA,cache[1])
        dA_prev, dW, db = linear_backward(dZ,cache[0])
        ### END CODE HERE ###

    elif activation == "sigmoid":
        ### START CODE HERE ### (≈ 2 lines of code)
        dZ = sigmoid_backward(dA,cache[1])
        dA_prev, dW, db = linear_backward(dZ,cache[0])
        ### END CODE HERE ###

    return dA_prev, dW, db12345678910111213141516171819202122232425262728 29
```

从后往前，执行L次反向传播过程,初始化cost函数关于AL的梯度

$$err = -\frac{1}{m}\sum_{i=1}^{m}(y^i log^{a_i{}^L} + (1-y^i) log^{1-a_i{}^L})$$

与aLi 有关的只有yi,logx的导数为1/x,向量化表示为：

$$dA^L = \frac{\partial err}{\partial A^l} = -\frac{Y}{A^L} - \frac{1-Y}{1-A^L}$$

定义L层模型模型的反向传播： 输入中的caches为正向传播每一层输出的Z和A

```
# GRADED FUNCTION: L_model_backward

def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR -> SIGMOID group

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (it's caches[l], for l in
range(L-1) i.e l = 0...L-2)
                the cache of linear_activation_forward() with "sigmoid" (it's caches[L-1])

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation

    ### START CODE HERE ### (1 line of code)
```

```
    dAL = -np.divide(Y,AL) - np.divide((1-Y),(1-AL))
    ### END CODE HERE ###

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "grads["dAL"],
grads["dWL"], grads["dbL"]
    ### START CODE HERE ### (approx. 2 lines)
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] =
linear_activation_backward(dAL,current_cache,'sigmoid')
    ### END CODE HERE ###

    for l in reversed(range(L - 1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 2)], caches". Outputs: "grads["dA" + str(l + 1)] ,
grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]
        ### START CODE HERE ### (approx. 5 lines)
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads['dA'
+str(l+2)],current_cache,'relu')
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
        ### END CODE HERE ###

    return
grads12345678910111213141516171819202122232425262728293031323334353637383940414243444546 47
```

更新参数：

```
# GRADED FUNCTION: update_parameters

def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of L_model_backward

    Returns:
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # number of layers in the neural network

    # Update rule for each parameter. Use a for loop.
    ### START CODE HERE ### (≈ 3 lines of code)
    for l in range(L):
        parameters["W" + str(l+1)] = parameters['W'+str(l+1)] -learning_rate *
grads['dW'+str(l+1)]
        parameters["b" + str(l+1)] = parameters['b'+str(l+1)] -learning_rate *
```

```
grads['db'+str(l+1)]
    ### END CODE HERE ###

    return parameters12345678910111213141516171819202122232425 26
```

将上述函数组合成为一个完整的函数：

```
# GRADED FUNCTION: L_layer_model

def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000,
print_cost=False):#lr was 0.009
    """
    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.

    Arguments:
    X -- data, numpy array of shape (number of examples, num_px * num_px * 3)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of
examples)
    layers_dims -- list containing the input size and each layer size, of length (number of
layers + 1).
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 100 steps

    Returns:
    parameters -- parameters learnt by the model. They can then be used to predict.
    """

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization.
    ### START CODE HERE ###
    parameters = initialize_parameters_deep(layers_dims)
    ### END CODE HERE ###

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
        ### START CODE HERE ### (≈ 1 line of code)
        AL, caches = L_model_forward(X,parameters)
        ### END CODE HERE ###

        # Compute cost.
        ### START CODE HERE ### (≈ 1 line of code)
        cost = compute_cost(AL, Y)
        ### END CODE HERE ###

        # Backward propagation.
        ### START CODE HERE ### (≈ 1 line of code)
        grads = L_model_backward(AL,Y,caches)
        ### END CODE HERE ###
```

```
        # Update parameters.
        ### START CODE HERE ### (≈ 1 line of code)
        parameters = update_parameters(parameters,grads,learning_rate)
        ### END CODE HERE ###

        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

前馈神经网络在NLP中的应用

基本思路 ：1、使用dence vector 表示单词序列

2、将dence vector 进行拼接，作为神经输入

3、将拼接后向量输入到多层神经网络中，进行各种任务

相关课程

吴恩达 深度学习工程师

台湾大学 李宏毅 深度学习，机器学习