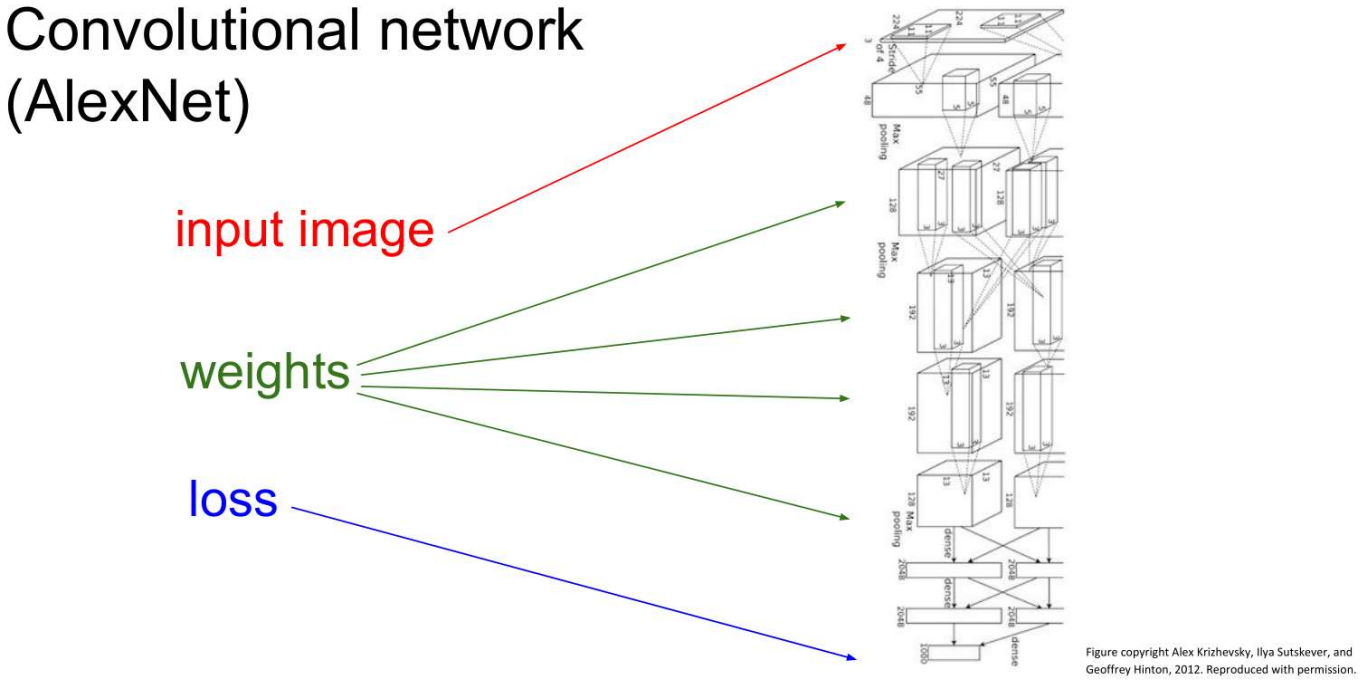
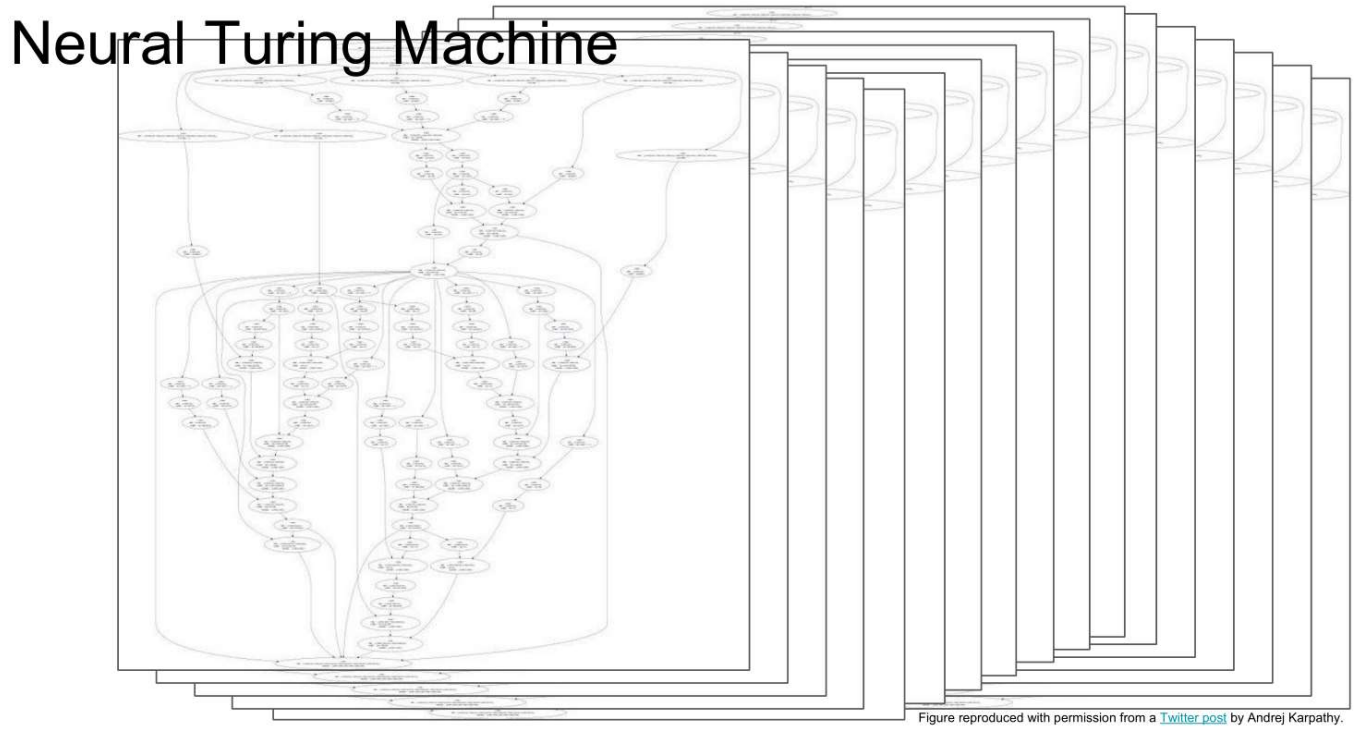
Lecture 4 Backpropagation and Neutral Networks

Part #1: Backpropagation

1、必要性

之前在SVM和softmax中，算个梯度直接可以用解析式求导来计算，方便快捷，而且计算速度也快不少，但是，在碰到像下面这样的情况的时候，想要直接通过对解析式求导求出梯度的公式用于计算梯度显然是不合理的。下面是**AlexNet**和**图灵机**。

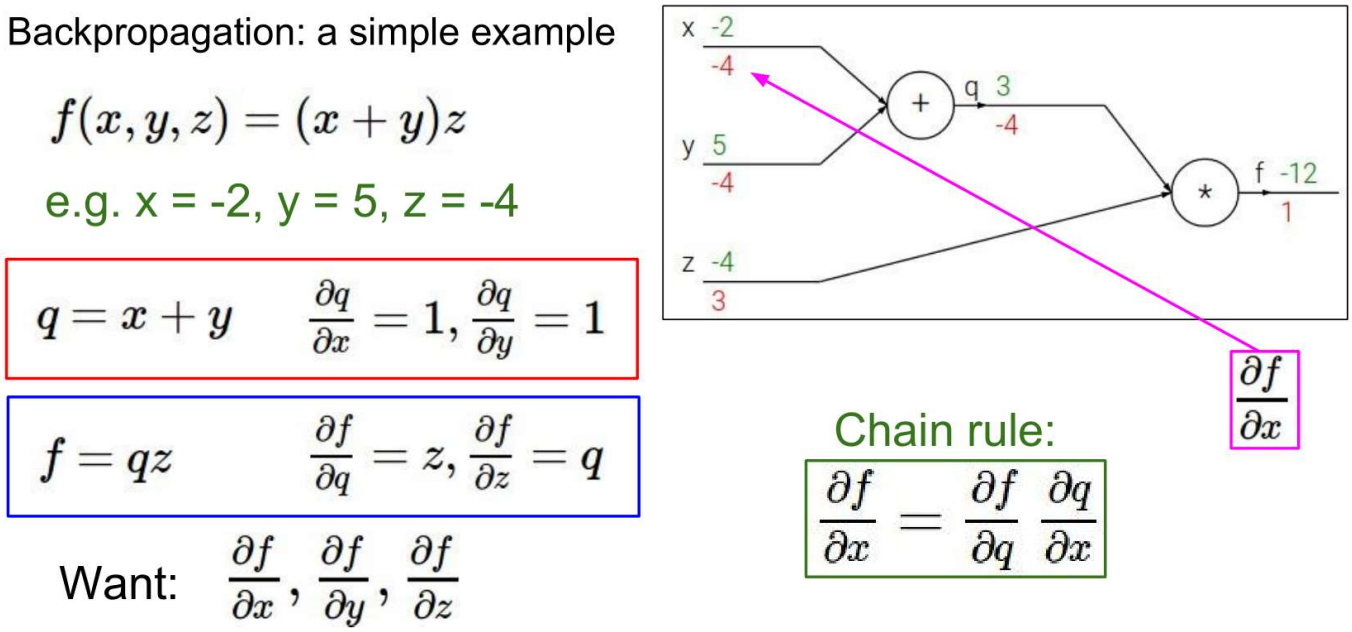




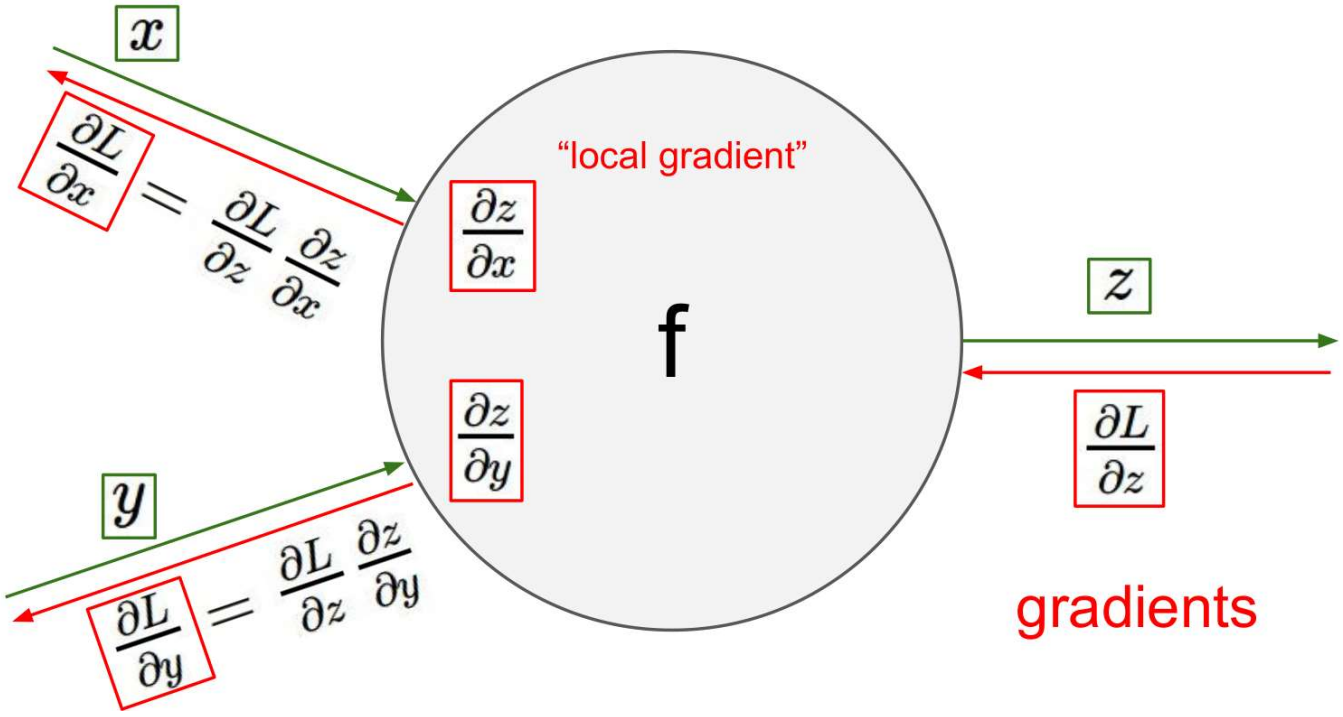
像上面这样的复杂的神经网络或者复杂的计算，梯度的计算是必要的但是很难直接去做，所以这个时候我们就需要开辟出另一种方法来解决这个问题————**反向传播**(Backpropagation)

2、理论基础 —— 微积分求导和链式法则

先看一个简单的例子：

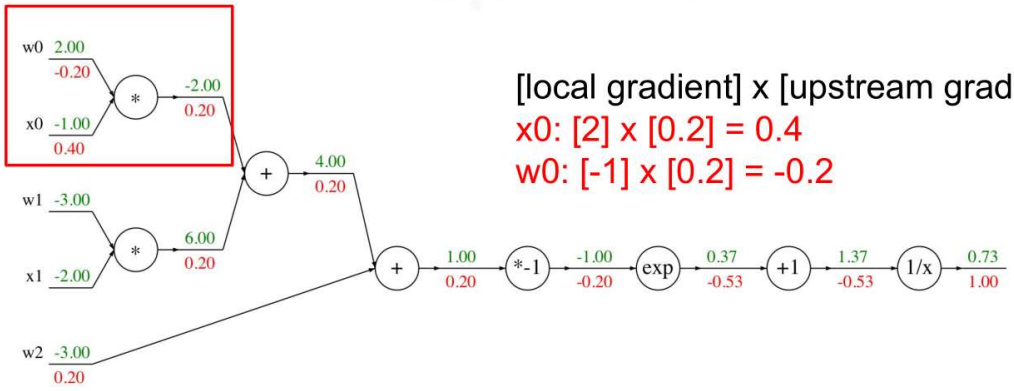


对右上角的图，我们想要获得输出f对每个input的梯度，也就是对f对三个输入x、y和z求偏导，这里我们应用到一个定理——**链式法则**，也就是右下角的公式。链式法则的意思就是求对输入的偏导可以用对中间量的偏导的乘积来表示，这样就大大简化了我们进行复杂计算时的计算量，这对我们求梯度自然也是非常关键的。



基本的计算公式如上图，就是上一级的梯度传递乘以"local gradient"——**本级的梯度**，下面是一个比较复杂的计算实例。

Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[local gradient] x [upstream gradient]
x0: [2] x [0.2] = 0.4
w0: [-1] x [0.2] = -0.2

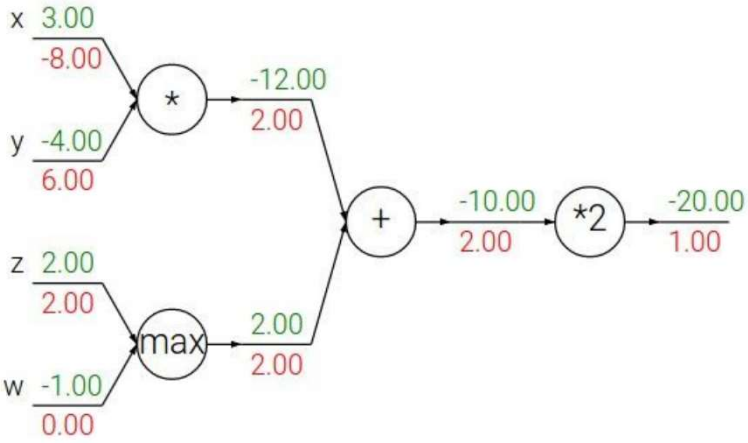
$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

基本运算的梯度类型

add gate: gradient distributor

max gate: gradient router

mul gate: gradient switcher



理解：

- 加法门：*gradient distributor*，意思是说，对于加法门来说，作用仅仅是将后一级的梯度向前传递而已，并不会对梯度有什么样的影响。
- 最大值门：*gradient router*，意思是说，这里只将后一级的梯度传递到最大的输入那里去，其他的较小输入就完全不管了。
- 乘法门：*gradient switcher*，意思是说，乘法门会对梯度做处理，使得前一级获得的梯度是后一级的梯度与另一个输入的乘积。

以上基本就是大概的需要了解的梯度类型。

3、梯度 for vecterized code(向量)

前面讲的基本上都是标量的梯度，现在需要注意的是向量的梯度，因为在神经网络的计算中，其实向量的应用要比标量的应用更多，甚至于可以说，在神经网络的每一个模块，其实都是向量在做运算，标量几乎很少有参与。所以说，我们更需要来说向量梯度的研究。

雅可比矩阵

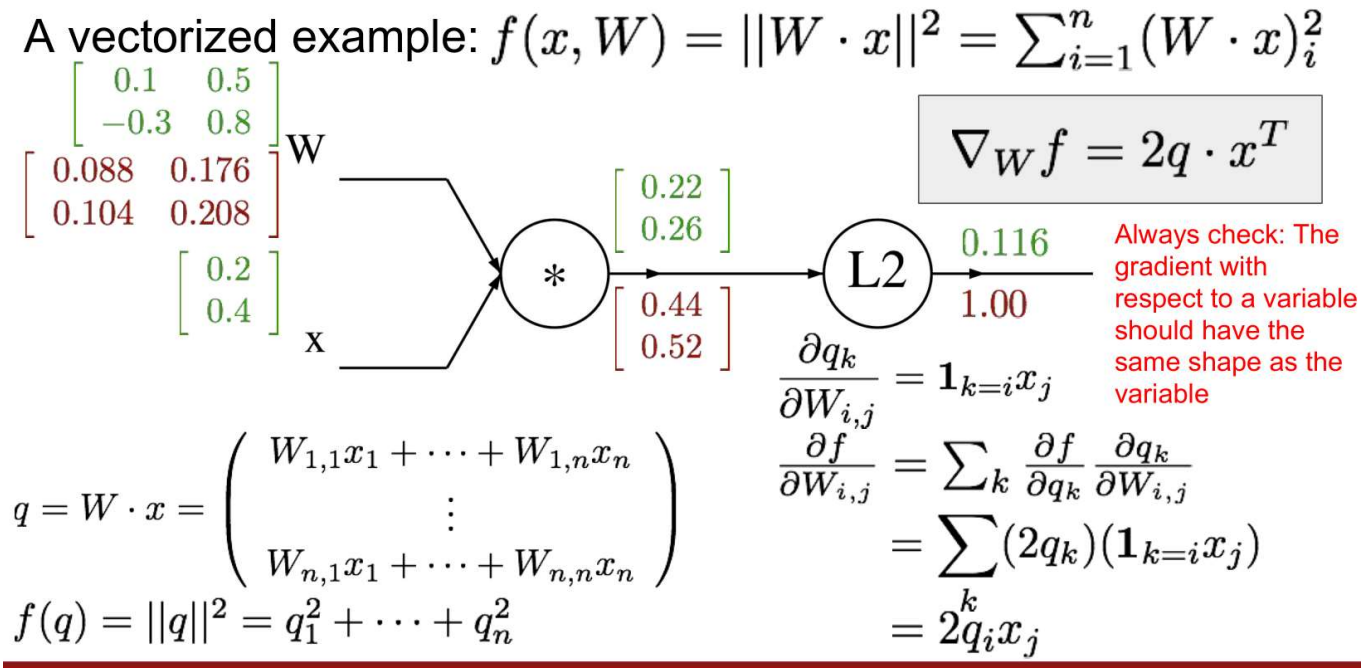
向量对向量求导的基本形式就是雅克比矩阵，具体的可以看下面这个图，没啥好说的，比较关键的是，对向量的求导等于对其每个元素的求导：

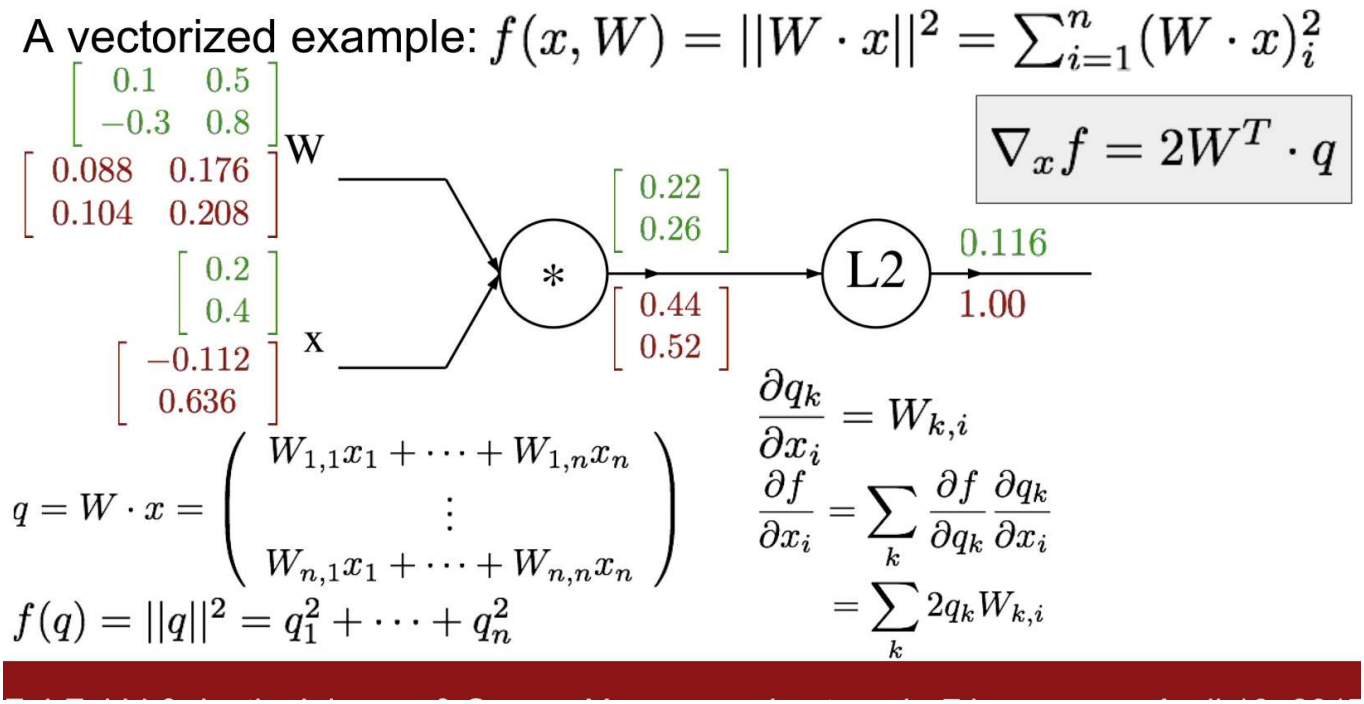
$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

https://blog.csdn.net/guoyunfei20/article/details/78229381

示例：

L2距离

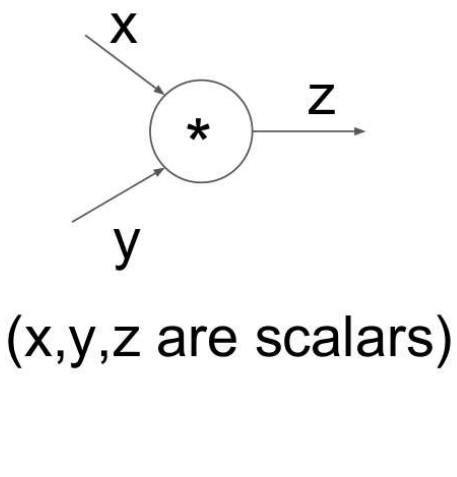




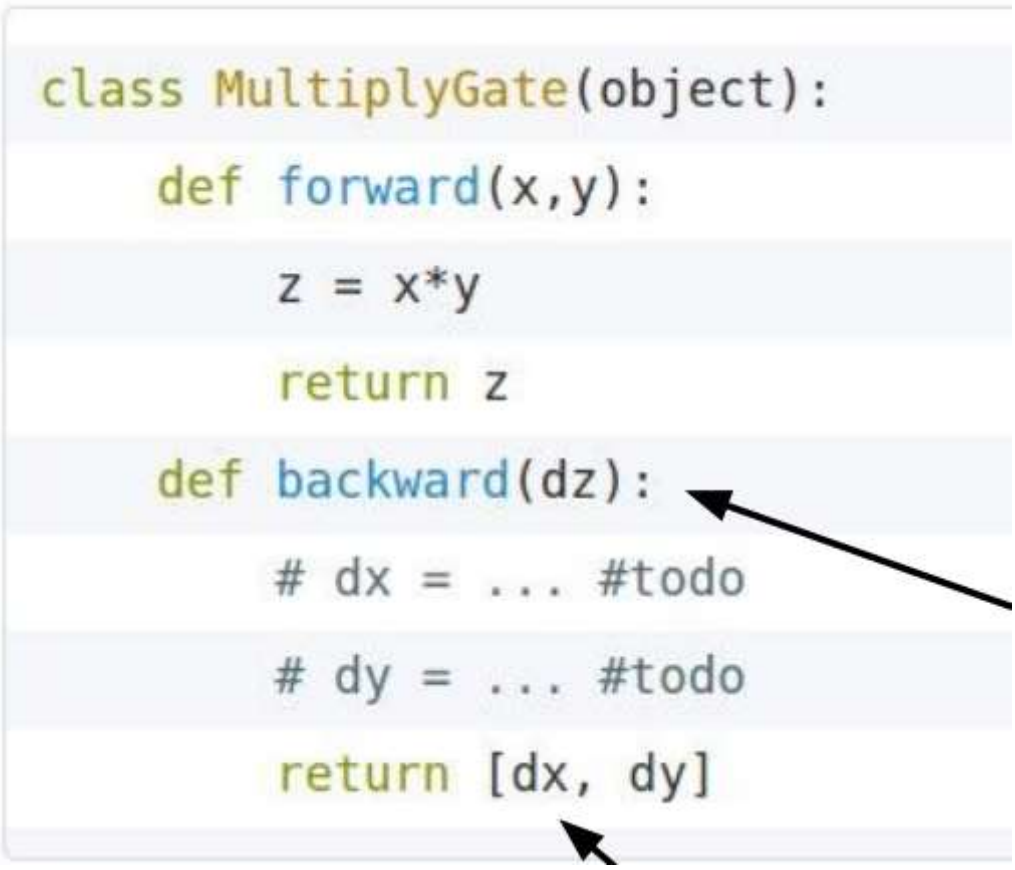
这是计算L2距离的时候的，输出对权重和输入分别求导的结果。

4、作为API调用

这里其实就是前向传播和后向传播的基本方法与思路，基本形式在下面




```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```



其中需要注意的是：

第一张图与第二张图的不同在于：

```
self.x = x
self.y = y
```

这一部分在第二张图中并没有出现，而这是计算乘积梯度必须要有的一部分。原因在于，乘积梯度需要用输入变量的值来计算，因此需要**在前向传播中对变量的值进行缓存，以便在反向传播中进行调用，不然反向传播将难以得到正确的结果，这个是我们需要注意的部分**

Part #2: Neutral Network

1、神经网络的基本形式

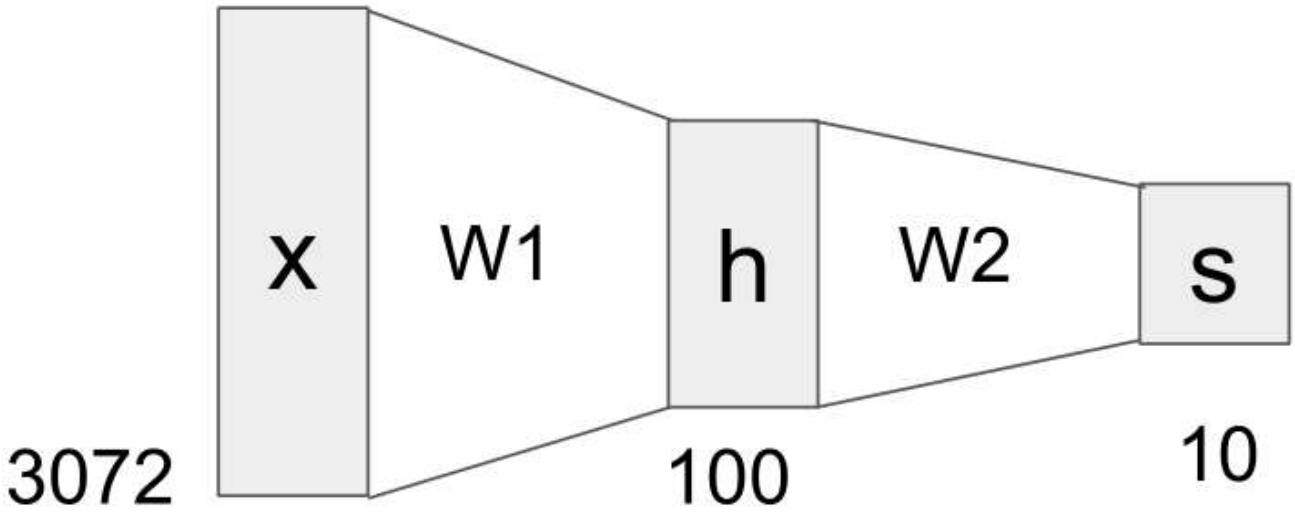
之前的分类器其实都属于线性分类器，带入到神经网络之中，其实需要加一些非线性的东西，可以理解为对网络的改进和信息传递的整合。下面例子的处理的**max**作为非线性因素。

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

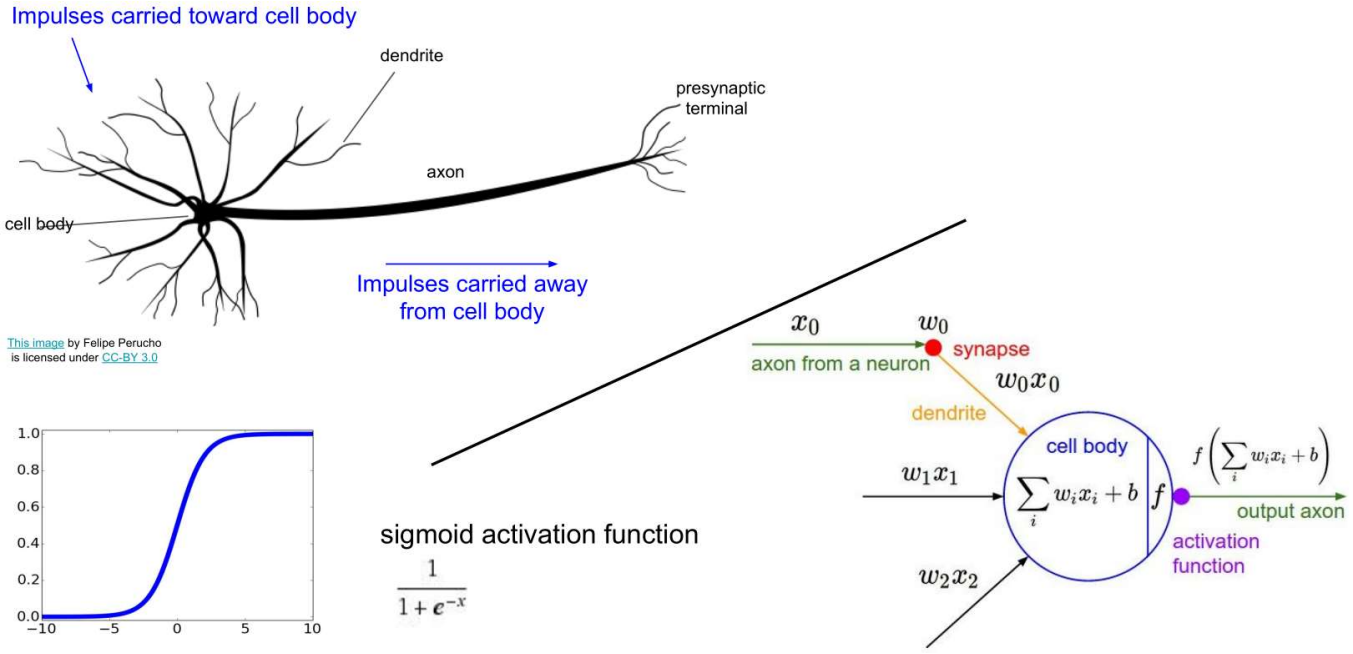
or 3-layer Neural Network

$f = W_3 \max(0, W_2 \max(0, W_1 x))$



2、从神经元看神经网络

下图是对单个神经元对于NN来说是单层的神经网络做了一个解析与比较，可以看到每层的基本计算原则。其中我们需要注意的是每次层的输出其实经过了一个**activation function(激活函数)**，这个是需要注意的。



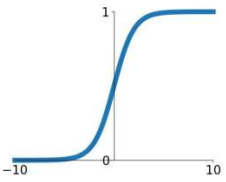
代码如下：

```
class Neuron:
  # ...
  def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

3、常用激活函数

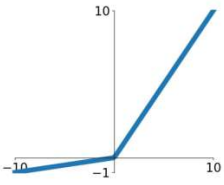
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



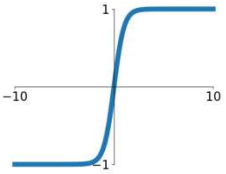
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

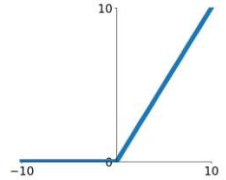


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

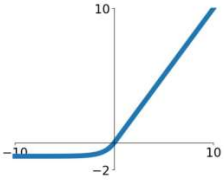
ReLU

$\max(0, x)$

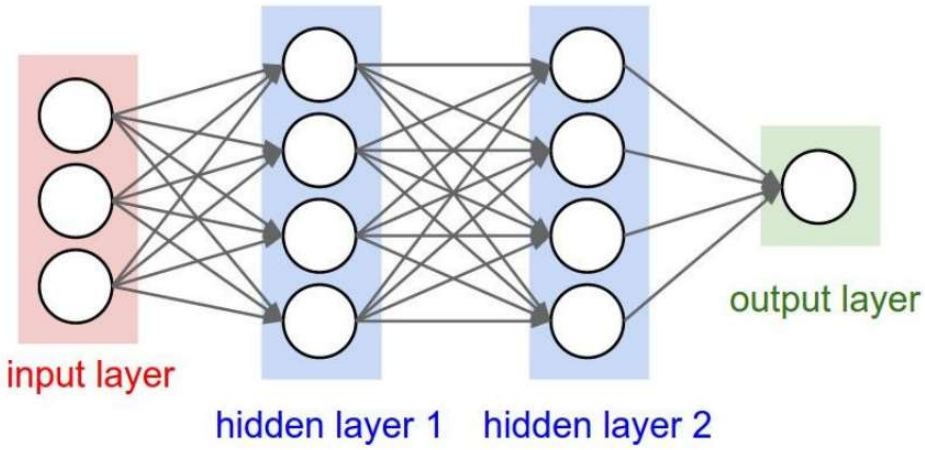


ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



4、简单阐明计算原则




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```