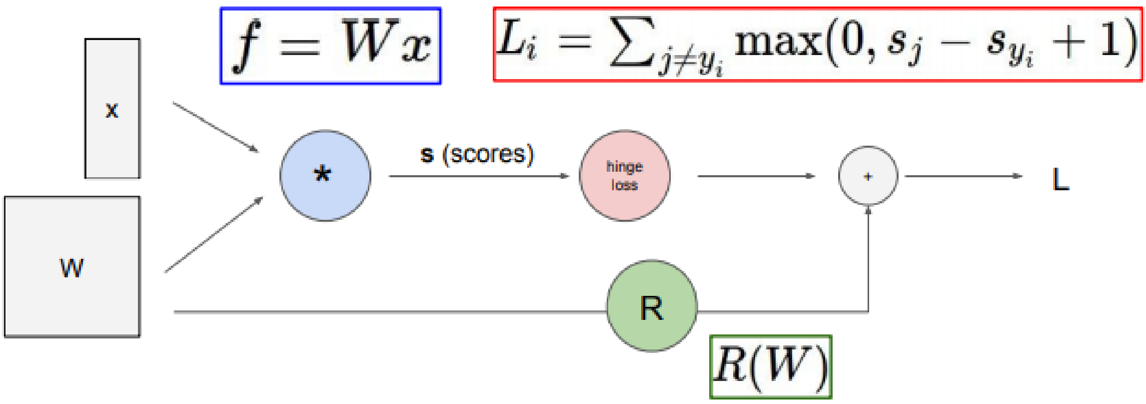
역전파(Back Propagation)

: gradient를 얻기 위해 computational graph 내부의 모든 변수에 대해 chain rule을 재귀적으로 사용. 복잡한 함수를 이용하여 작업할 때 매우 유용.

computational graph는 그래프를 이용해 어떤 함수든지 표현할 수 있게 해줌.

그래프의 각 노드는 연산 단계를 나타낸다. 아래는 input이 x, W인 선형 classifier. hinge loss라는 계산노드를 통해 데이터 항 Li를 계산한다. 노드 R은 regularization 항, 최종 loss인 L은 regularization 항과 데이터 항의 합.



<예제>

첫번째 단계는 항상 함수 f를 computational graph로 나타낸다.

x는 -2, y는 5, z는 -4이고 최종 노드를 통과하면 -12를 얻는다. 이 때 x+y 덧셈 노드를 q라고 부르고 f는 q*z가 된다. q와 z에 대한 f의 gradient는 곱셈규칙에 의해 각각 z와 q이다. backpropagation은 chain rule의 재귀적인 응용으로 chain rule에 의해 뒤에서부터 시작한다.

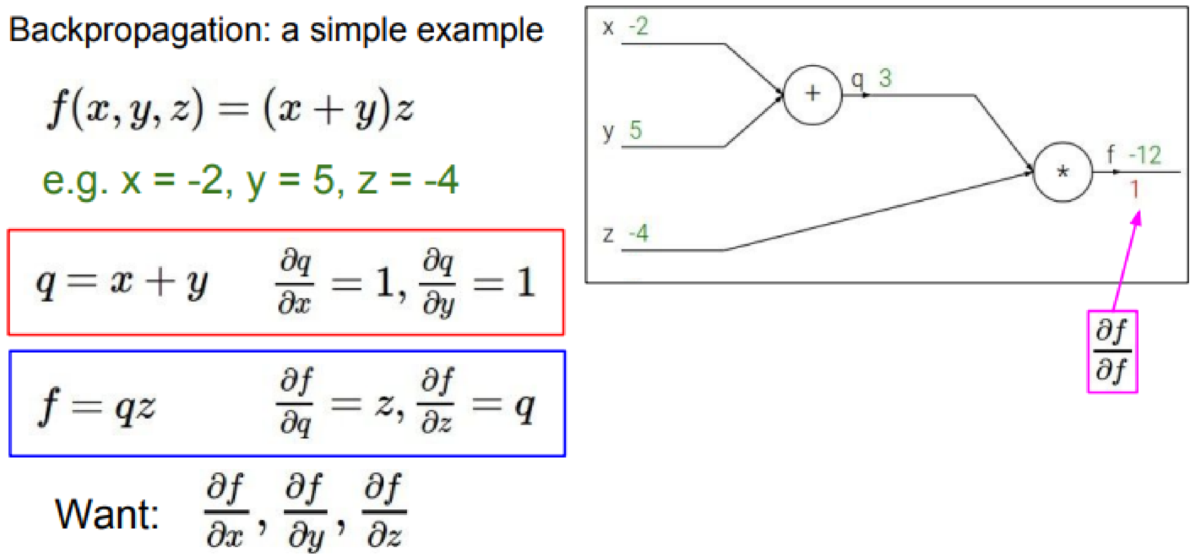
1. 출력의 f에 대한 gradient를 계산하면 1.

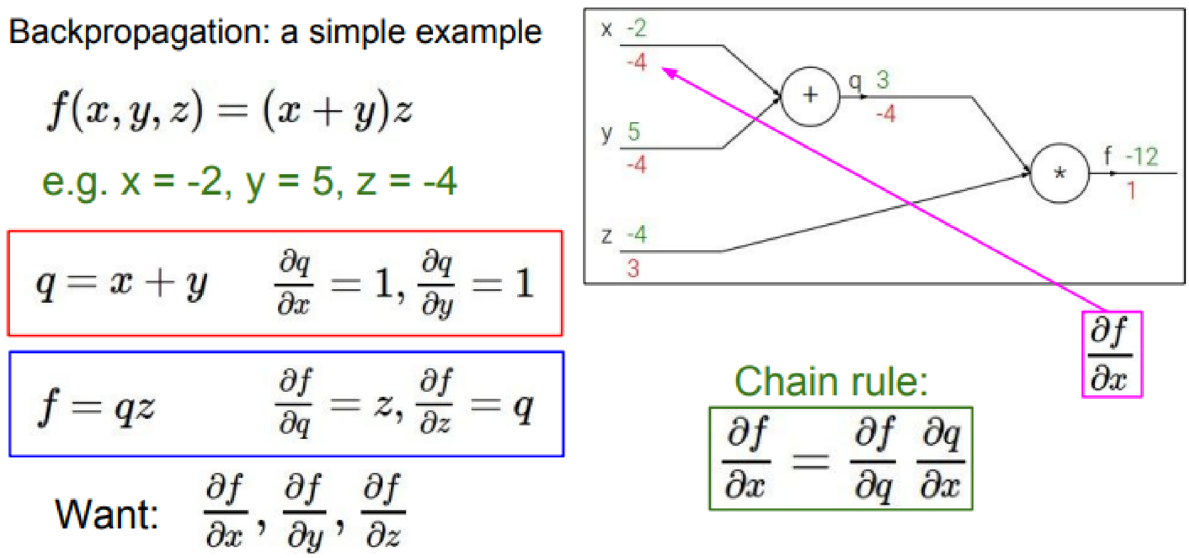
2. z에 대한 f의 미분은 q. q의 값은 3. 따라서 z에 대한 f의 미분값은 3.

3. q에 대한 f의 미분값은 z와 같고 여기에서 z는 -4. 따라서 q에 대한 f의 미분값은 -4.

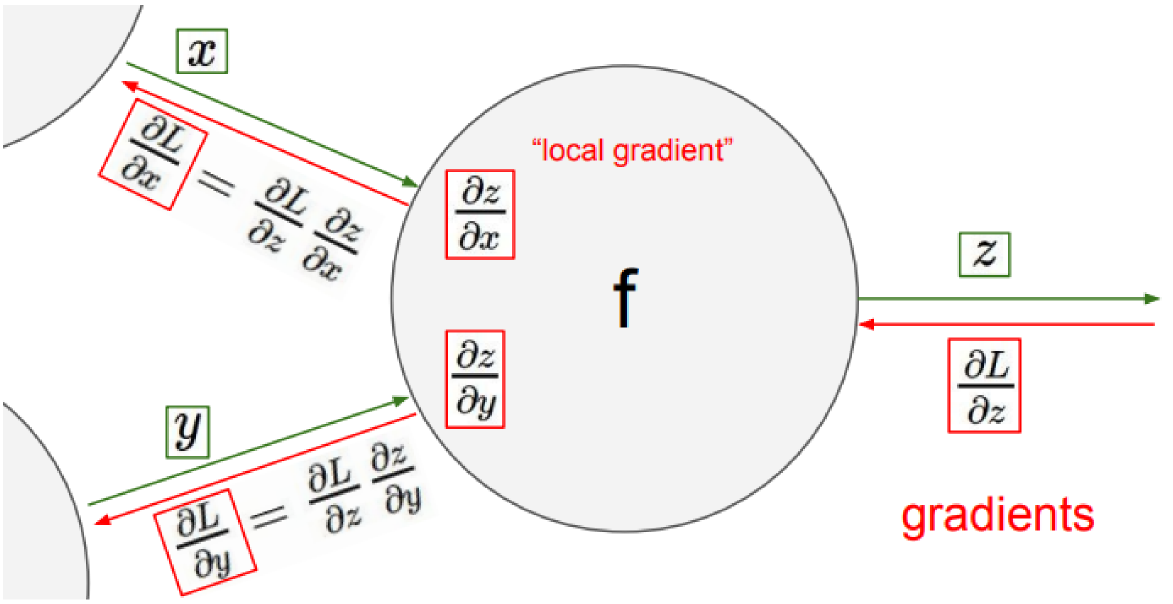
4. y에 대한 f의 미분값 => y와 f는 바로 연결되어있지 않지만 f는 z와 연결되어있다. 이때 chain rule을 이용해 y에 대한 f의 미분은 q에 대한 f의 미분과 y에 대한 q의 미분의 곱으로 나타낼 수 있다. 따라서 -4.

5. f에 대한 x의 gradient는 -4 * 1 이므로 -4.





local gradient를 구할 수도 있다 (들어오는 입력에 대한 출력의 기울기: z에서 x에 대한 gradient를 구할 수 있으며 y에 대한 gradient). 각 노드는 local 입력을 받고 밑에 보이는 것처럼 다음 노드로 출력값을 보낸다.



<예제2>

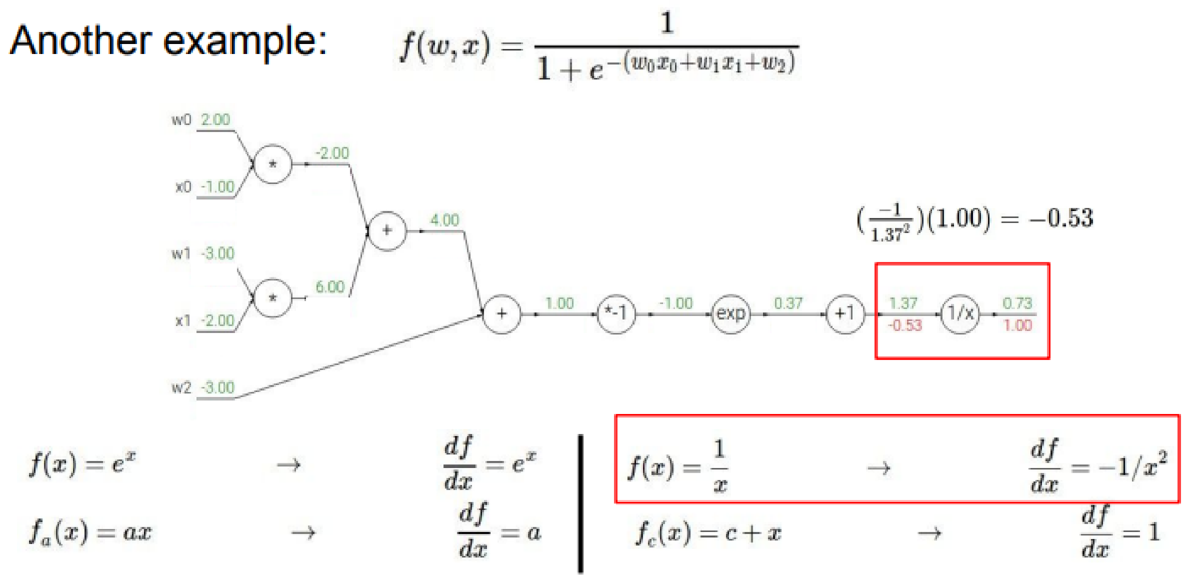
최종 변수에 대한 출력의 gradient는 1.

upstream gradient는 1. 주어진 f는 1/x 이고, 이것의 local gradient인 x에 대한 f의 미분은 -1/x^2. 따라서 -1/x^2를 x에 대입한다. (x)은 1.37이고,이 변수에 대한 최종 gradient는 1/-1.37^2 * 1 = -0.53.

다음 노드로 넘어가면 upstream gradient는 -0.53이다. 그리고 현재 노드는 +1이다. (x+상수)에 대한 local gradient는 1이다. chain rule를 사용하면 upstream gradient는 -0.53이고, 우리의 local gradient는 1이기 때문에 gradient 는 -0.53 이다.

한 단계 더 진행해보면 upstream gradient는 -0.53이고 local gradient는 exponential 노드에서, chain rule에 의해 gradient가 -0.53*e^x임을 알 수 있다. (x의) 값이 -1인 경우 (exponential 노드에서) 최종 gradient는 -0.2이다.

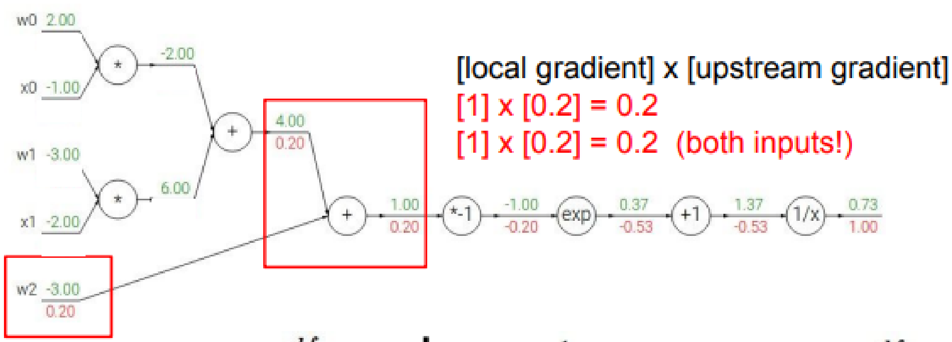
다음 노드는 -1과 곱하는 노드이다. upstream gradient는 -0.2이고 local gradient는 local gradient는 x에 대한 f의 미분임으로 -1이다 (1 * -1). 그렇기 때문에 gradient는 -1 * -0.2 = 0.2이다.



덧셈노드에서 두개의 노드와 연결된다. upstream gradient는 0.2이다. 이전의 예제에서 덧셈 노드를 가지고 있을 때 덧셈 노드에서, 각 입력에 대한 local gradient는 1이었다. 여기에서 local gradient는 upstream gradient 0.2와 한번 곱해질 것임으로 1 * 0.2 = 0.2이다. 이것은 아래 브랜치에서도 똑같이 적용되 총 gradient는 0.2 이다.

w_0과 x_0이 있는곳까지 가면 여기에서는 곱셈 노드를 가지고 있다. 이전에 보았던 곱셈 노드에서 input에 대한 gradient는 한 인풋에 대하여 다른 인풋의 값이었다. 이 경우 w_0에 대한 gradient는 -0.2이다. 아래에 있는 다른 하나(x0)는 -1이기 때문에 -1과 0.2를 곱해 -0.2를 얻게된다. x0에 대해서도 같은 과정을 반복하면 0.2 * 2를 하게 되므로 0.4를 얻는다.

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



$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$ | $f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$ | $f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$

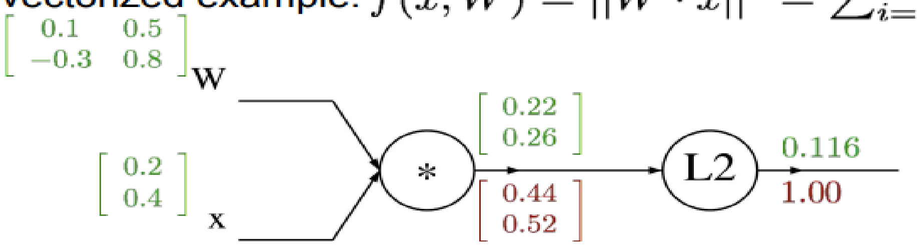
<벡터화 된 예제>

x와 W에 대한 함수 f는 x에 의해 곱해진 W의 L2와 같다. 그리고 이 경우 x를 n-차원이라고 하고, W는 n*n이라고 한다. 최종출력에 대한 gradient는 1이다.

한 노드 뒤로 이동하면 L2 이전의 중간 변수인 q에 대한 gradient를 찾아야된다. q는 2차원의 벡터이다. 우리가 찾고 싶은 것은 의 각각의 요소가 f의 최종 값에 어떤 영향을 미치는지이다. 밑에 식을 보면 도함수를 구했을 때 벡터의 형태로 쓰면 벡터 q에 대해서 2를 곱한 것이라는 것을 알 수 있다. 따라서 각 요소를 2로 곱한 gradient 0.44와 0.52를 얻었다.

W의 gradient를 찾기 위해 다시 chain rule를 사용한다.

A vectorized example: $f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$



$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = ||q||^2 = q_1^2 + \cdots + q_n^2$$

$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\boxed{\nabla_q f = 2q}$$

신경망(Neural Network)

아래 사진의 위에 식은 선형 변환식이고 다른 하나는 2층짜리 신경망의 레이어이다. 위에는 W1과 x의 행렬 곱이며 이것의 중간값을 얻고 max(0, W)라는 비선형 함수를 이용해 선형 레이어 출력의 max를 얻는다. 선형 레이어들을 쌓는다면 결국 선형 함수가 되기 때문에 비선형 변환은 매우 중요하다. 예제를 보면 첫번째 선형 레이어를 가지고 있고 그 다음 비선형 레이어를 가지고 있다 그리고 그 위에 선형 레이어를 추가함으로 최종적으로 score 함수를 얻을 수 있다. 신경망은 함수들의 집합이다. 비선형의 복잡한 함수를 만들기 위해서 간단한 함수들을 계층적으로 여러개 쌓아올린다.