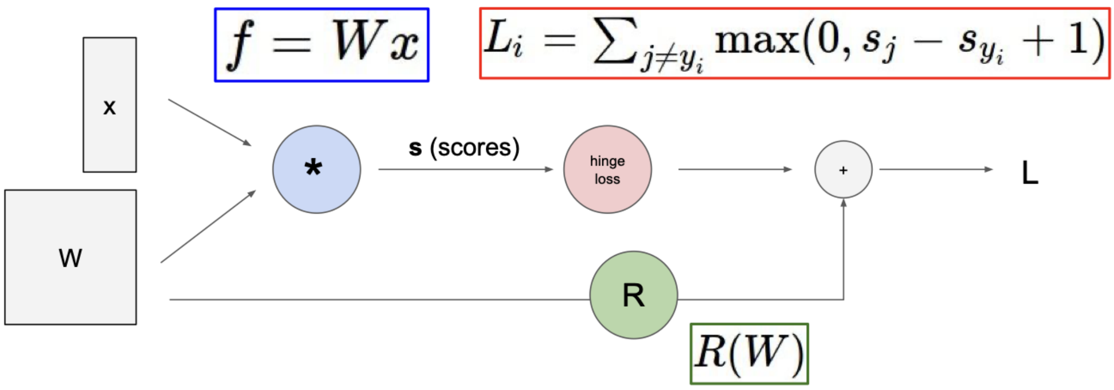
Backpropagation and Neural Networks

Backpropagation

1. Computational graphs



   a. Useful for backpropagation

      i. input: x, W

      ii. function: multiplication of x with W

      iii. output vector of scores fed into the hinge loss

      iv. final loss = hinge loss + regularization

2. Backpropagation

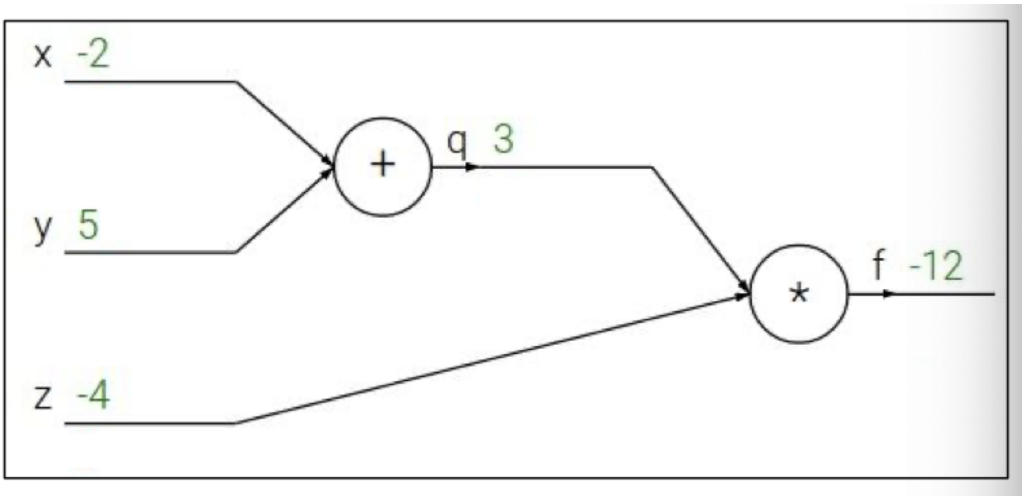
   a. Recursively use the chain rule in order to compute the gradient w.r.t every variable in the computational graph

   b. Simple example

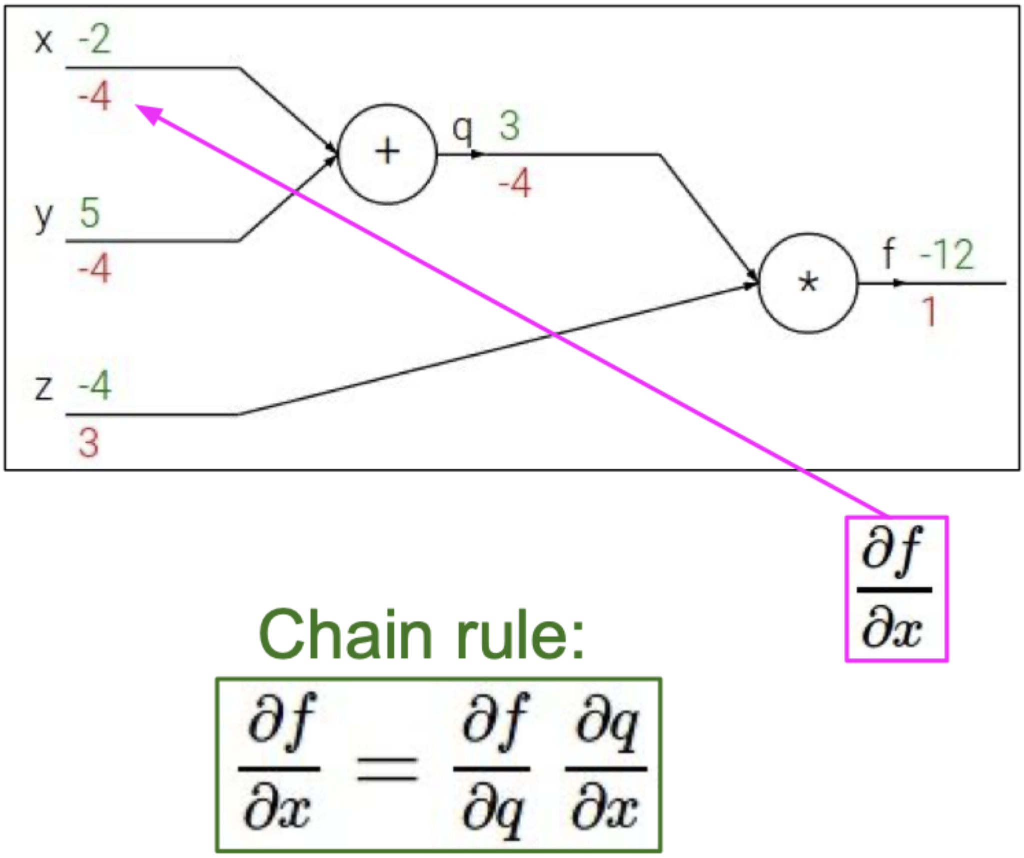
      i. $f(x, y, z) = (x + y)z$

         - $q = x + y, f = qz$

      ii. forward



iii. backward



- $\frac{\partial f}{\partial f} = 1, \frac{\partial f}{\partial z} = q = 3, \frac{\partial f}{\partial q} = z = -4$
- $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = -4 \times 1 = -4, \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = -4 \times 1 = -4$

c. gradients vs local gradient

i. gradients: $\dfrac{\partial L}{\partial z}$

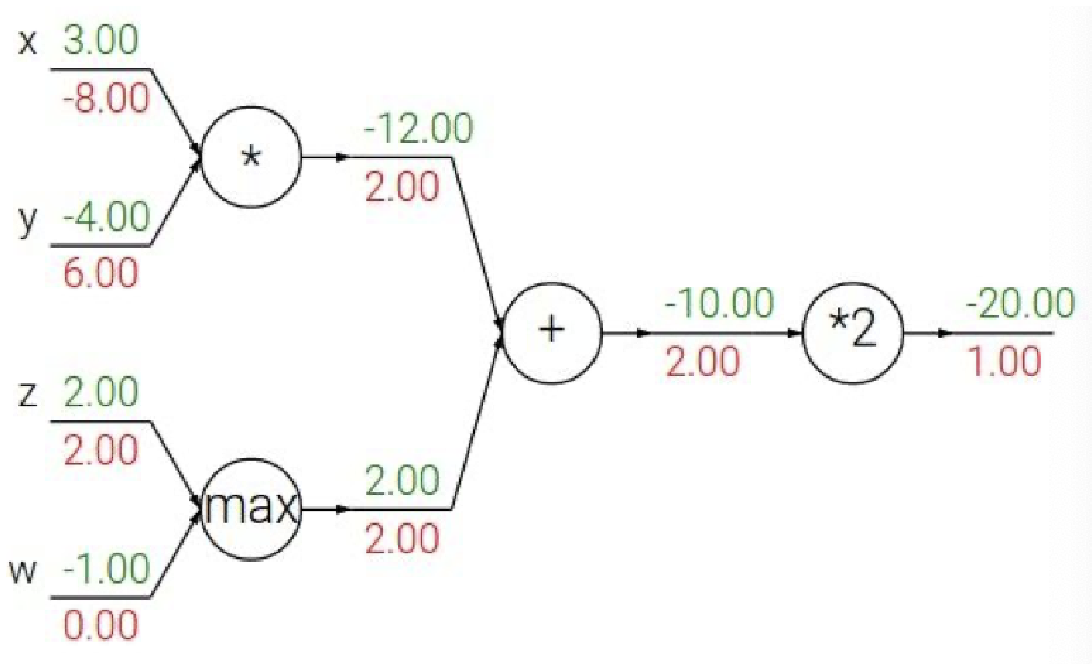
- upstream gradients which is just a value

ii. local gradient: $\dfrac{\partial z}{\partial x}, \dfrac{\partial z}{\partial y}$

- upstream gradients are multiplied with local gradient

d. Check the slides for more complex example

e. Patterns in backward flow



i. add gate: gradient distributor

ii. max gate: gradient router

- 하나의 값이 두 branch로 갈라지는데, 둘 중 하나의 branch에만 영향을 주고 나머지 하나는 0이 됨

iii. mul gate: gradient switcher (scaler)

- 값의 크기를 변화시킴

f. Gradients for vectorized code

i. local gradient is Jacobian matrix

- $\dfrac{\partial z}{\partial x}$: derivative of each element of z w.r.t each element of x

ii. vectorized operations
    - $f(x) = max(0, x)$ with 4096-d input vector and 4096-d output vector
        - Jacobian matrix = size [4096 x 4096] of diagonal matrix
        - diagonal b/c each vector element affects the corresponding element

g. Modularized implementation
    i. forward API
        - input: x, y
        - output: z (loss)
    ii. backward API
        - input: dz
        - output: [dx, dy] (input gradients)

3. Summary
    a. neural nets will be very large
        i. impractical to write down gradient formula by hand for all parameters
    b. backpropagation is a recursive application of the chain rule along a computational graph to compute the gradients of all input, parameters, and intermediates
    c. implementations maintain a graph structure where the nodes implement the forward, backward API
    d. forward pass computes result of an operation and save any intermediates needed for gradient computation in memory
    e. backward pass applies the chain rule to compute the gradient of the loss function w.r.t the inputs

Neural Networks

1. Linear function vs Neural network
    a. Linear function: $f = Wx$
    b. 2-layer Neural network: $f = W_2 max(0, W_1 x)$

i. 3072-d vector x

- input of the network

ii. $h = max(0, W_1 x)$

- 100-d score vector right after the non-linearity
- W_1 corresponds to the templates of the inputs

iii. $s = f = W_2 max(0, W_1 x)$

- 10-d final score vector
- 10 is the number of classes
- W_2 performs weighted sum of all the templates

2. Non-linearity

a. Important element of neural network

i. b/c just stacking linear layers (without non-linearity), the function collapses like a single linear function

b. Activation functions

i. Sigmoid

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

ii. tanh

iii. ReLU

- $max(0, x)$

iv. Leaky ReLU

- $max(0.1x, x)$

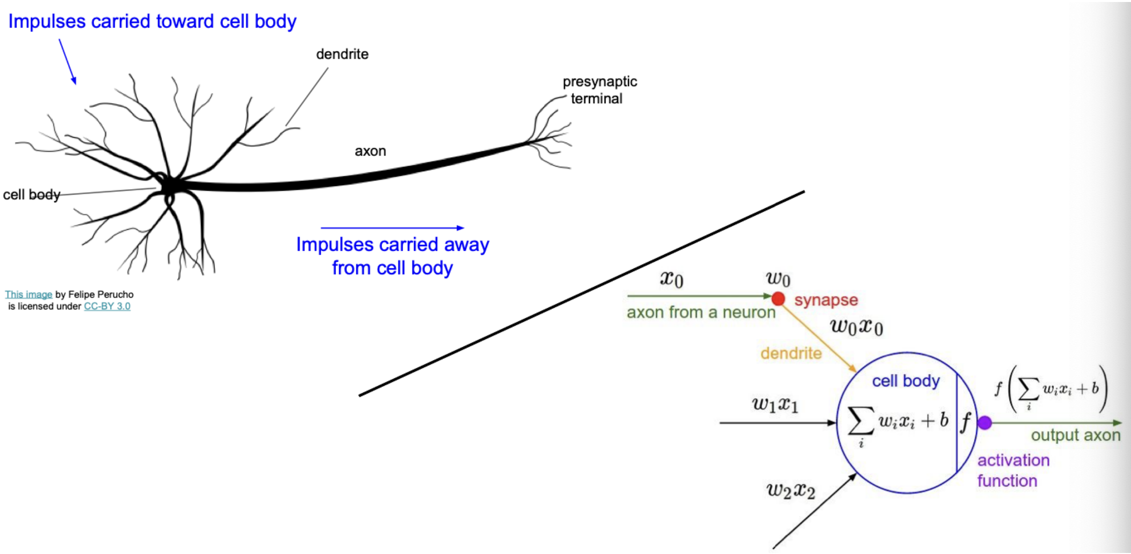
v. Maxout

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

vi. ELU

- x (x≥0)
- $\alpha(e^x - 1)$ (x<0)

3. Neurons



a. Biological neurons
   i. many different types
   ii. dendrites can perform complex non-linear computations
   iii. synapses are not a single weight but a complex non-linear dynamical system
   iv. rate code may not be adequate

4. Architectures
   a. Fully-connected layers
      i. 하나의 레이어와 다른 레이어의 모든 element가 모두 연결되어있는 상태
   b. 2-layer neural net = 1-hidden-layer neural net
   c. 3-layer neural net = 2-hidden-layer neural net

5. Summary
   a. arrange neurons into fully-connected layers
   b. the abstraction of a layer has the nice property that it allows us to use efficient vectorized code (e.g., matrix multiplies)
   c. neural networks are not really *neural*