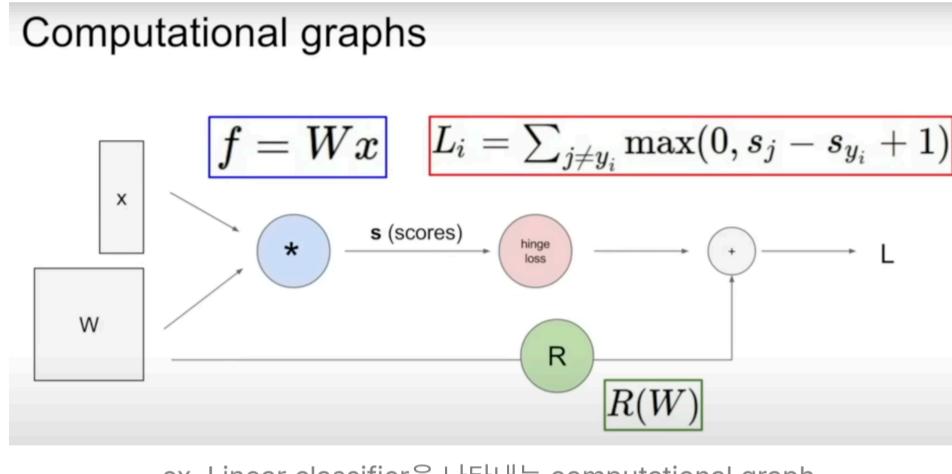


Introduction to Neural Networks

Computational graph를 통해 Analytic gradient 계산을 이해해보자

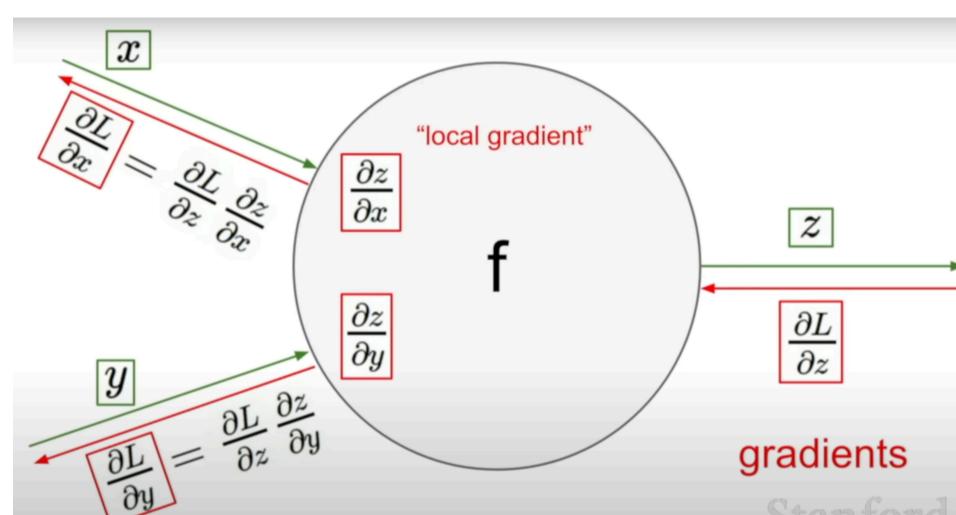
Computational graphs

: (복잡한) 함수를 (분해해서) 나타내기 위해서 사용하는 그래프. 각 노드는 계산 단계를 의미

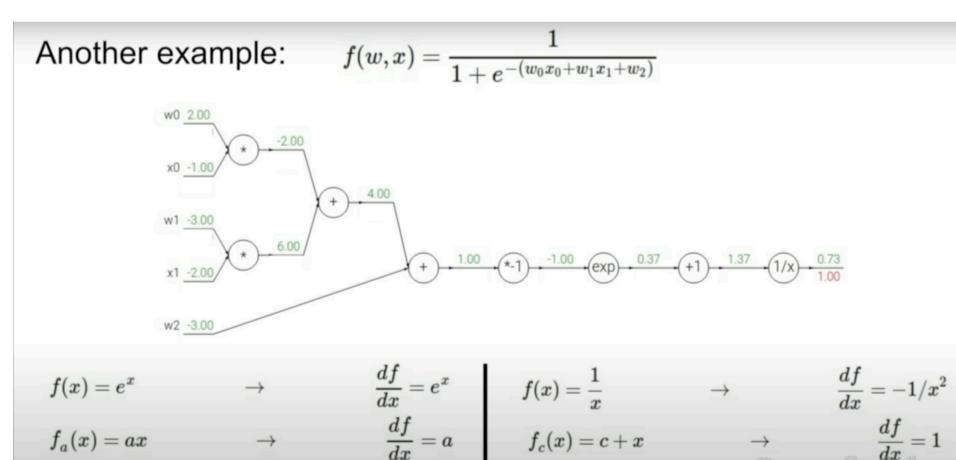


특히 Backpropagation(역전파) 테크닉을 사용할 때 나타내기 좋음

→ Computational graph로 모든 계산 과정을 break down해서 chain rule 사용



직접 연결된 노드의 gradient(=local gradient)를 계산하는 것은 쉽지만, 꽤 멀리 떨어진 노드의 gradient를 구하는 것은 복잡함. 이 과정은 computational graph를 통해 chain rule 계산을 더 직관적으로 이해하고 편하게 나타낼 수 있음. Backprop을 진행하면 upstream에서 오는 값을 local gradient로 곱해 연결된 노드에 보내 input node까지 전달하는 과정을 거침. 이 과정은 바로 직전의 노드 말고 다른 것은 신경쓰지 않아도 됨



위에서 보이듯이 computational graphs를 사용해서 backprop 할 때, 단순히 local gradient만 계산한 후 upstream에서 오는 값에 chain rule을 사용한 곱연산의 반복임. 이 과정을 통해서 모든 변수의 기울기를 구할 수 있음. 그렇기에 analytic gradient를 단순히 계산하는 것보다 쉬움

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}} \quad \sigma(x) = \frac{1}{1 + e^{-x}} \quad \text{sigmoid function}$$

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x))\sigma(x)$$

sigmoid gate

$$(0.73) * (1 - 0.73) = 0.2$$

addition, multiply 등의 단순한 계산외에도 복잡한 function을 하나의 압축된 노드로 요약해서 사용하기도 함. 이 과정에서 복잡한 수식을 간단히 정리해서 계산에 사용. 대표적인 예시인 시그모이드 함수

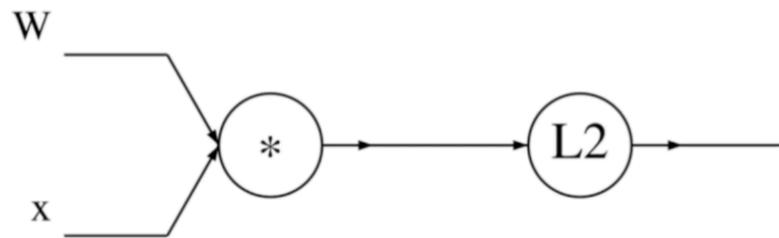
Vectorized Operations

위 계산은 스칼라인 경우이지만, 실제로 우리는 벡터나 행렬 연산에 사용할 것

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

위와 같을 때, $x \in \mathbb{R}^n, W \in \mathbb{R}^{n \times n}$ 라고 가정해보자.

위 계산을 Computational Graph로 나타내면 다음과 같다.



여기서 chain rule을 계산하기 위해 q 를 정의하면 다음과 같다.

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

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

그리고 임의로 W, x 를 다음으로 정의해보자.

$$W = \begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix}, x = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$$

W, x 에 임의의 값을 부여했으니, 아까 정의한 $q, f(q)$ 를 구할 수 있다.

$$q = W \cdot x = \begin{bmatrix} 0.22 \\ 0.26 \end{bmatrix}$$

$$f(q) = \|q\|^2 = 0.22^2 + 0.26^2 = 0.46$$

이제 여기서 backpropagation을 진행하면, 다음은 trivial하다.

$$\frac{\partial f}{\partial f} = 1$$

다음 변수 q 의 기울기를 구하면 다음과 같다.

$$\frac{\partial f}{\partial q_i} = 2q_i = \begin{bmatrix} 0.44 \\ 0.52 \end{bmatrix}$$

그리고 다음과 같이 정리할 수 있다.

$$\nabla_q f = 2q$$

스칼라 버전의 computational graph 연산에서 확인했듯이 곱연산의 backprop은 switcher 역할

\Rightarrow 그러므로 $\frac{\partial q_k}{\partial W_{i,j}}(q \text{ w.r.t } W_{i,j})$ 는 다음과 같다.

$$\frac{\partial q_k}{\partial W_{i,j}} = 1_{k=i} \cdot x_j$$

- $1_{k=i}$ 는 indicator function

chain rule을 사용해 f w.r.t $W_{i,j}$ 를 계산하면,

$$\begin{aligned} \frac{\partial f}{\partial W_{i,j}} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} \\ &= \sum_k (2q_k)(1_{k=i}x_j) \\ &= 2q_i x_j \end{aligned}$$

그러면 다음과 같이 정리할 수 있다.

$$\nabla_W f = 2q \cdot x^T$$

위와 같은 방식으로, $\frac{\partial q_k}{\partial x_i}(q \text{ w.r.t } x_i)$ 는 다음과 같다.

$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

chain rule을 사용해 f w.r.t x_i 를 계산하면,

$$\begin{aligned} \frac{\partial f}{\partial x_i} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} \\ &= \sum_k 2q_k W_{k,i} \end{aligned}$$

다음과 같이 정리할 수 있다.

$$\nabla_x f = 2W^T \cdot q$$

항상 체크해야 할 것

“변수 기울기 차원의 shape = 변수 차원의 shape”

각 기울기는 element가 function의 최종 output에 어느 정도 영향을 주는지 나타냄

Modularized implementation: forward / backward API

```
class ComputationalGraph(object):
    # ...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of the backprop (chain rule applied)
        return inputs_gradients
```

Summary

Neural nets은 굉장히 방대해서 모든 파라미터를 기울기 수식으로 나타내는 것은 impractical

⇒ Backprop 테크닉

Backpropagation

: computational graph를 사용해서 chain rule의 recursive application을 사용하는 테크닉

→ 모든 계산을 역방향으로 거슬러 올라가서 inputs, parameters와 같은 모든 intermediates의 기울기를 계산할 수 있음

forward

: 계산 결과를 구하고, 이후 메모리에 gradient 계산에 사용할 intermediate value 저장

backward

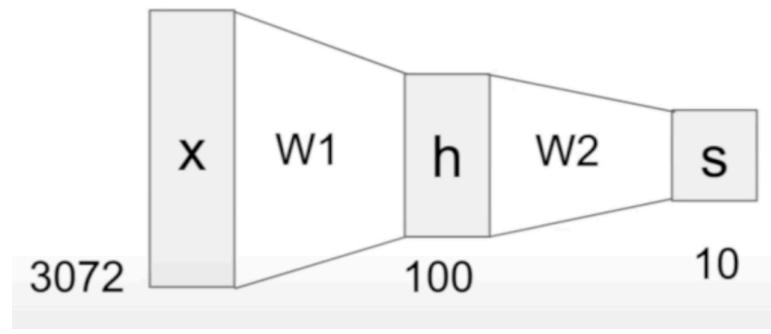
: upstream gradient를 가져와 계산한 local gradient와 함께 chain rule을 사용해 input에 대한 기울기를 구함. 이 기울기는 다시 다음 연결된 노드로 upstream gradient값으로 전달

Neural Networks

가장 단순하게 Neural Network를 나타내는 방법은 n-layer을 쌓는 것

→ multiple stages of hierarchical computation

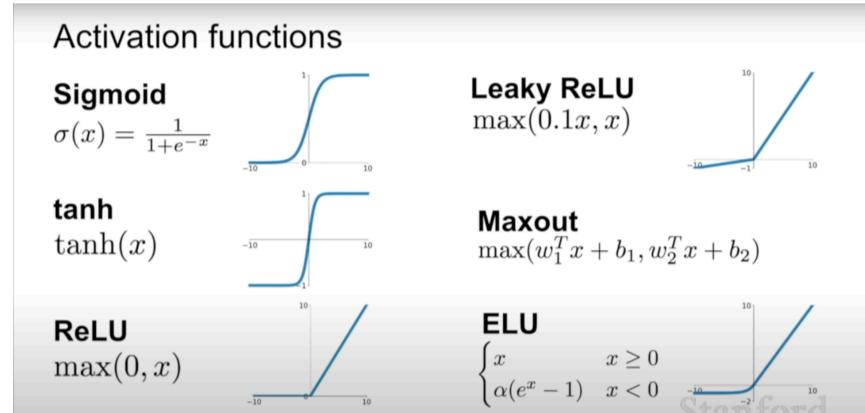
- (Before) Linear score function : $f = Wx$
- (Now) 2-layer Neural Network : $f = W_2 \max(0, W_1x)$



이전 강의에서 언급했듯이 linear classifier은 하나의 템플릿만 가짐. 그러므로 한 카테고리에 있는 다양성을 학습하지 못하는 단점
 → ex. 세상에는 다양한 자동차 색이 있지만 linear classifier의 자동차 카테고리 template이 빨간 자동차만의 모습인 경우

반면, Neural Network는 다수의 층을 쌓음. W1까지는 linear function과 같이 하나의 template을 가지지만 (non-linear function을 지나) h layer에서 각 layer마다 다른 template을 가질 수 있음. 이후 h의 모든 vector를 매개하는 W2가 h layer의 특성을 결합.

- Non-linear function



- Neural Networks Architectures

