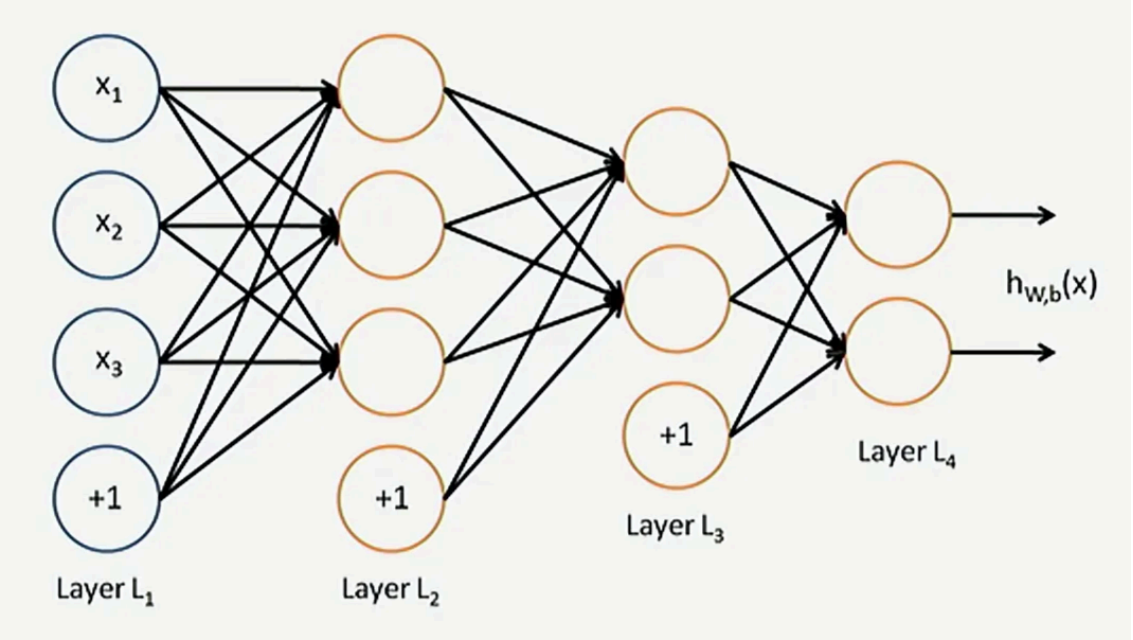
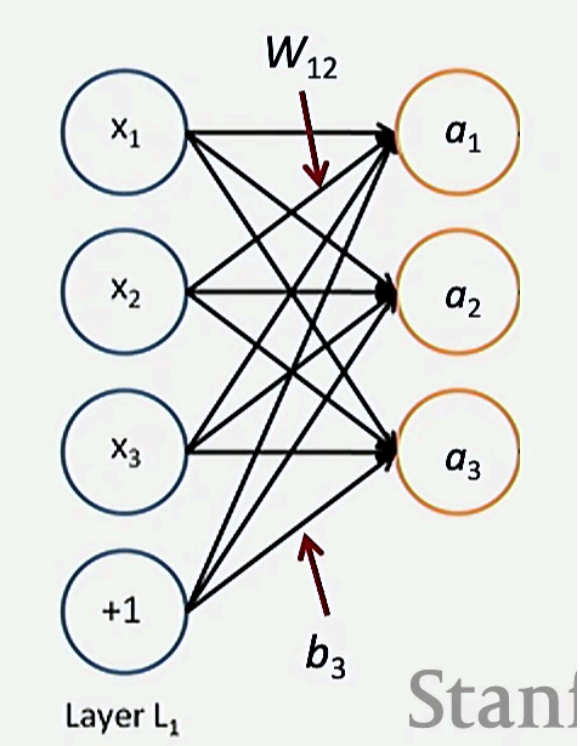
Lecture3

Spring 2024 : Backpropagation, Neural Network

Running several logistic regressions at the same time



non-linear



one ouput feeding with weights → next layer input (+bias term)

입력 없이도 출력을 조정할 수 있게 해주는 값

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$
$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$
$$etc.$$

$---$

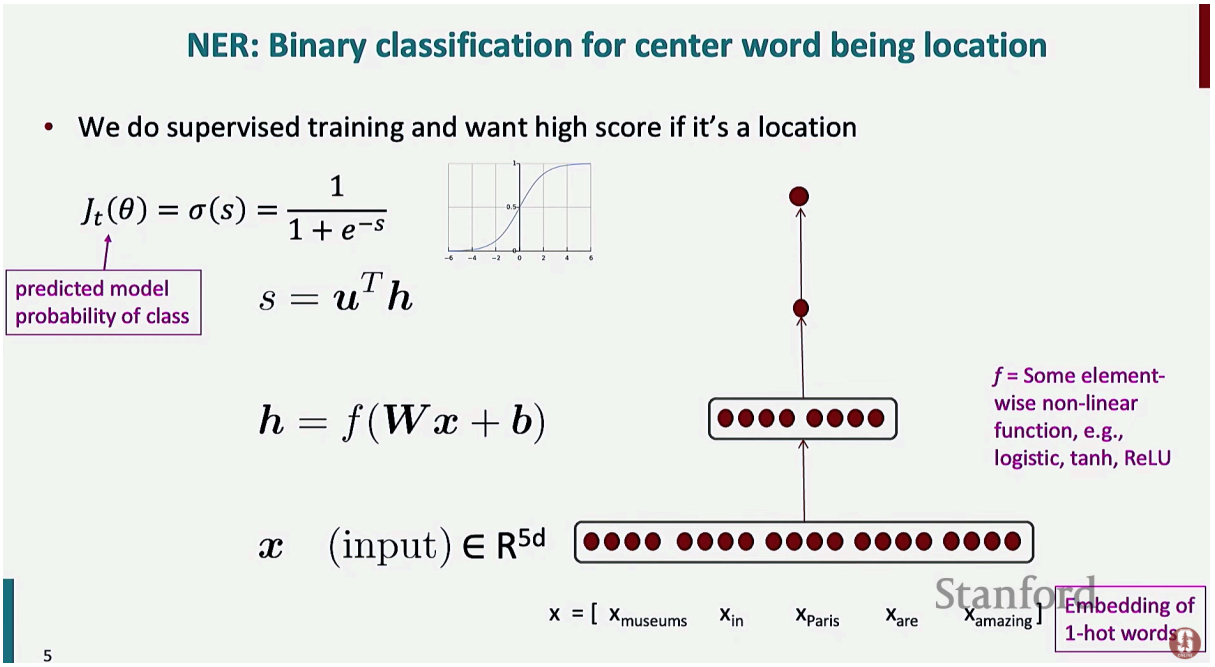
$$z = Wx + b$$
$$a = f(z)$$

$---$

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

collapse to vector → metrics

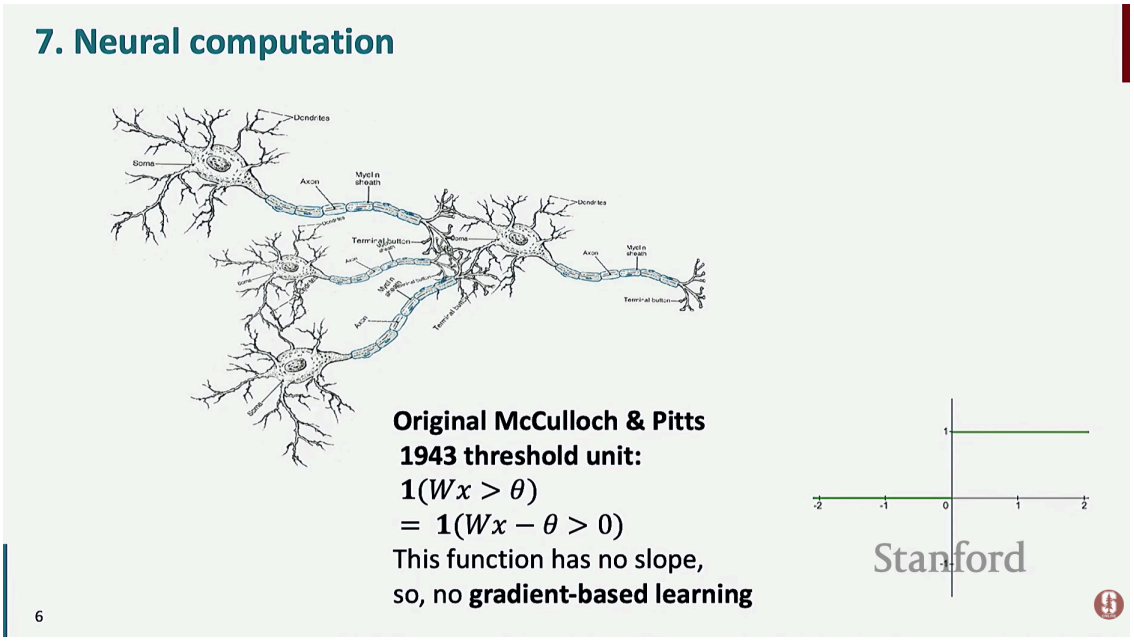
scalar function : applying to each element of the vector



- Hidden Layer

$$h = f(Wx + b)$$

function: non-linear ex) sigmoid

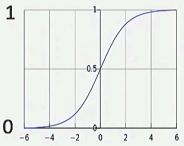


input과 weigths의 linear combination >θ에 따라 1 or 0 출력

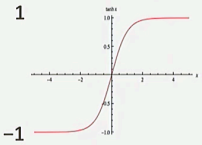
→no gradient

Non-linearities, old and new

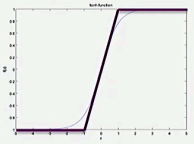
logistic ("sigmoid") $f(z) = \frac{1}{1+\exp(-z)}$



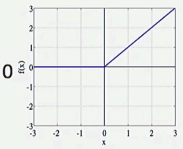
tanh $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



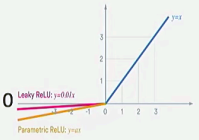
hard tanh $\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$



(Rectified Linear Unit) ReLU $\text{ReLU}(z) = \max(z, 0)$



Leaky ReLU / Parametric ReLU



tanh is just a rescaled and shifted sigmoid (2 × as steep, [−1,1]):

$$\tanh(z) = 2\text{logistic}(2z) - 1$$

Logistic and tanh are still used (e.g., logistic to get a probability)

However, now, for deep networks, the first thing to try is ReLU: it trains quickly and performs well due to good gradient backflow.

ReLU has a negative "dead zone" that recent proposals mitigate

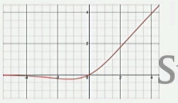
GELU/Swish often used with Transformers (BERT, RoBERTa, etc.)

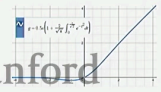
Swish arXiv:1710.05941

$\text{swish}(x) = x \cdot \text{logistic}(x)$

GELU arXiv:1606.08415

$\text{GELU}(x) = x \cdot P(X \leq x), X \sim N(0,1)$
$\approx x \cdot \text{logistic}(1.702x)$





Stanford

logistic("sigmoid") : 음의 값 x

tanh: logistic * scaling +shift

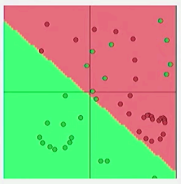
hard tanh: gradient :1 효과적

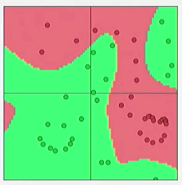
ReLU: 음수 입력에선 gradient 0 → "dead zone"

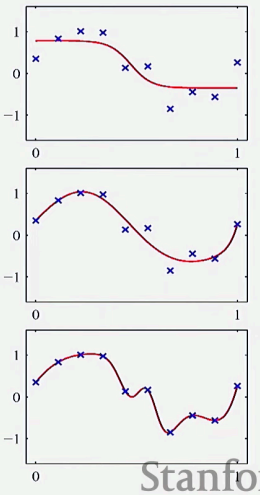
Leaky ReLU / Parametric ReLU: 음수도 기울기

Non-linearities (i.e., "*f*" on previous slide): Why they're needed

- Neural networks do function approximation, e.g., regression or classification
  - Without non-linearities, deep neural networks can't do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform: $W_1 W_2 x = Wx$
  - But, with more layers that include non-linearities, they can approximate any complex function!







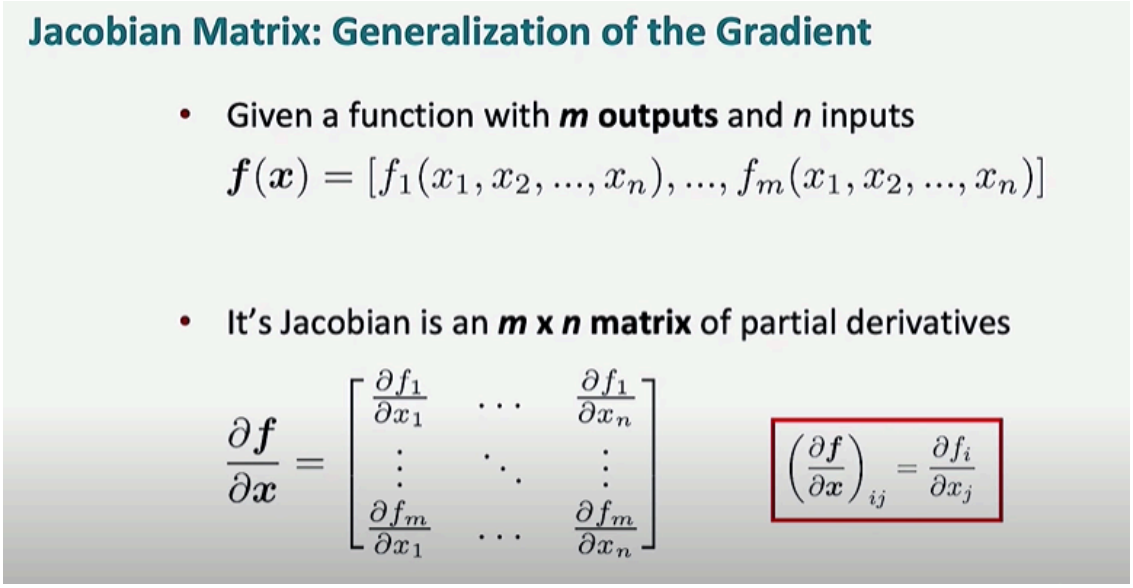
Stanford

왜 비선형적이어야하는가?

to make function approximation

층을 여러 개 쌓아도 모두 선형이면 결국 **하나의 선형 변환**과 같음:

$W_1 W_2 x = W x$



Jacobian Matrix: 각f_i에 대한 gradient vectors

$\boldsymbol{J} = \frac{\partial f}{\partial x} \in \mathbb{R}^{m \times n}$

$\left(\frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right)_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{\partial}{\partial z_j} f(z_i)$

i=j 일 때만 미분 값 존재

i≠j 이면 서로 독립 → 미분값 0

→ Diagonal Matrix 꼴

$$\frac{\partial}{\partial \boldsymbol{x}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b})=\boldsymbol{W}$$
$$\frac{\partial}{\partial \boldsymbol{b}}(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b})=\boldsymbol{I}\ \ (\text{Identity matrix})$$
$$\frac{\partial}{\partial \boldsymbol{u}}(\boldsymbol{u}^T\boldsymbol{h})=\boldsymbol{h}^{\boldsymbol{T}}$$

1.

- W는 **고정된 행렬**, x는 벡터
- Wx는 선형변환
- x에 대해 편미분하면 변화율이 W로 일정

2.

- b는 벡터
- Wx는 b와 무관하므로 미분하면 0
- $\frac{\partial b}{\partial b}=I$

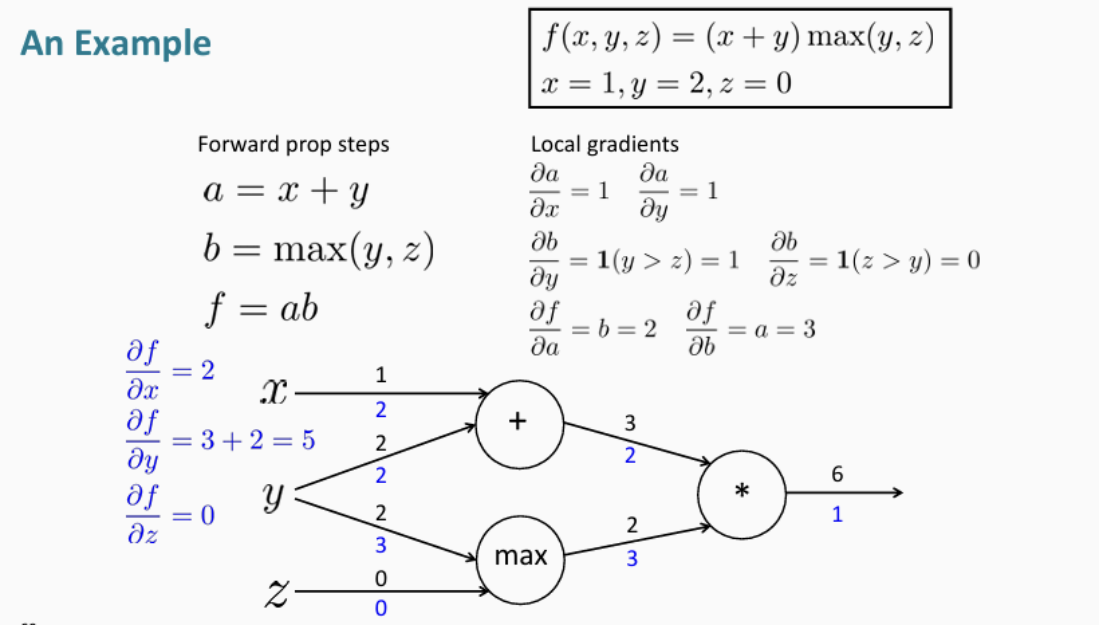
3.

- u^Th는 벡터 내적 → 결과는 스칼라 값
- u에 대해 미분하면 내적에서는 **상대 벡터(h)**가 남게 됨

$$\mathrm{u}=\begin{bmatrix}u_1\\u_2\\u_3\end{bmatrix},\quad h=\begin{bmatrix}h_1\\h_2\\h_3\end{bmatrix}\Rightarrow u^\top h=u_1h_1+u_2h_2+u_3h_3$$

$$\frac{\partial}{\partial u_1}(u^\top h)=h_1$$
$$\frac{\partial}{\partial u_2}(u^\top h)=h_2$$
$$\frac{\partial}{\partial u_3}(u^\top h)=h_3$$

$$\frac{\partial}{\partial u}(u^\top h)=\begin{bmatrix}h_1 & h_2 & h_3\end{bmatrix}=h^\top$$



$a = x + y = 1 + 2 = 3$
$b = \max(y, z) = \max(2, 0) = 2$
$f = ab = 3 \cdot 2 = 6$

$\frac{\partial f}{\partial a} = b = 2$
$\frac{\partial f}{\partial b} = a = 3$

$\frac{\partial a}{\partial x} = 1, \quad \frac{\partial a}{\partial y} = 1$

$(\text{Since } y > z) \Rightarrow \frac{\partial b}{\partial y} = 1, \quad \frac{\partial b}{\partial z} = 0$

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial x} = 2 \cdot 1 = \boxed{2}$
$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial y} + \frac{\partial f}{\partial b} \cdot \frac{\partial b}{\partial y} = 2 \cdot 1 + 3 \cdot 1 = \boxed{5}$
$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial b} \cdot \frac{\partial b}{\partial z} = 3 \cdot 0 = \boxed{0}$

□

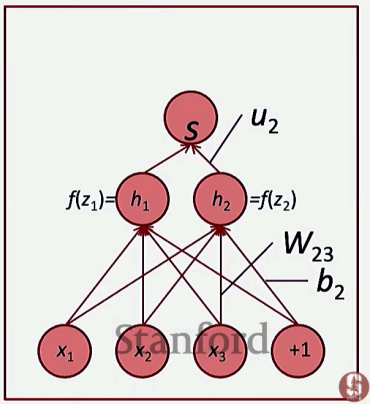
Deriving local input gradient in backprop

- For $\frac{\partial z}{\partial W}$ in our equation:

$$\frac{\partial s}{\partial W} = \delta \frac{\partial z}{\partial W} = \delta \frac{\partial}{\partial W}(Wx + b)$$

- Let's consider the derivative of a single weight W_{ij}
- W_{ij} only contributes to z_i
  - For example: W_{23} is only used to compute z_2 not z_1

$$\frac{\partial z_i}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} W_{i\cdot} x + b_i$$
$$= \frac{\partial}{\partial W_{ij}} \sum_{k=1}^{d} W_{ik} x_k = x_j$$



45

What shape should derivatives be?

Two options for working through specific problems:

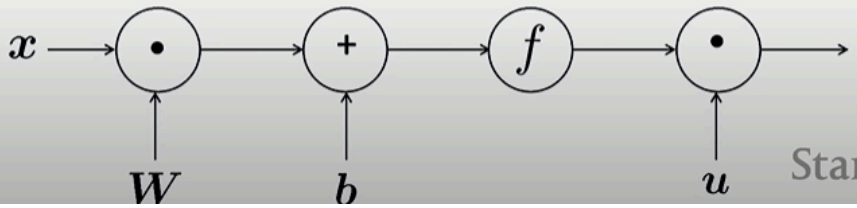
1. Use Jacobian form as much as possible, reshape to follow the shape convention at the end:
   - What we just did. But at the end transpose $\frac{\partial s}{\partial b}$ to make the derivative a column vector, resulting in δ^T
2. Always follow the shape convention
   - Look at dimensions to figure out when to transpose and/or reorder terms
   - The error message δ that arrives at a hidden layer has the same dimensionality as that hidden layer

Stanford

Computation Graphs and Backpropagation

- Software represents our neural net equations as a graph
  - Source nodes: inputs
  - Interior nodes: operations

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$

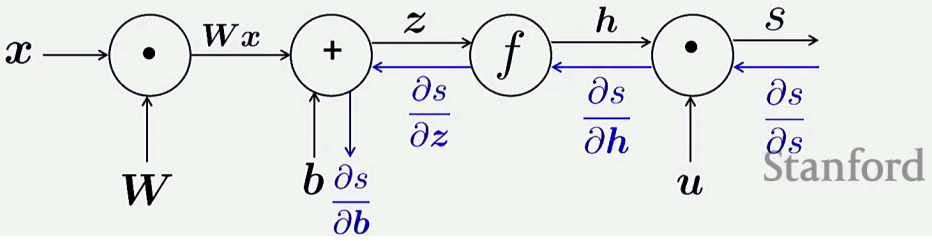


Forward Propagation

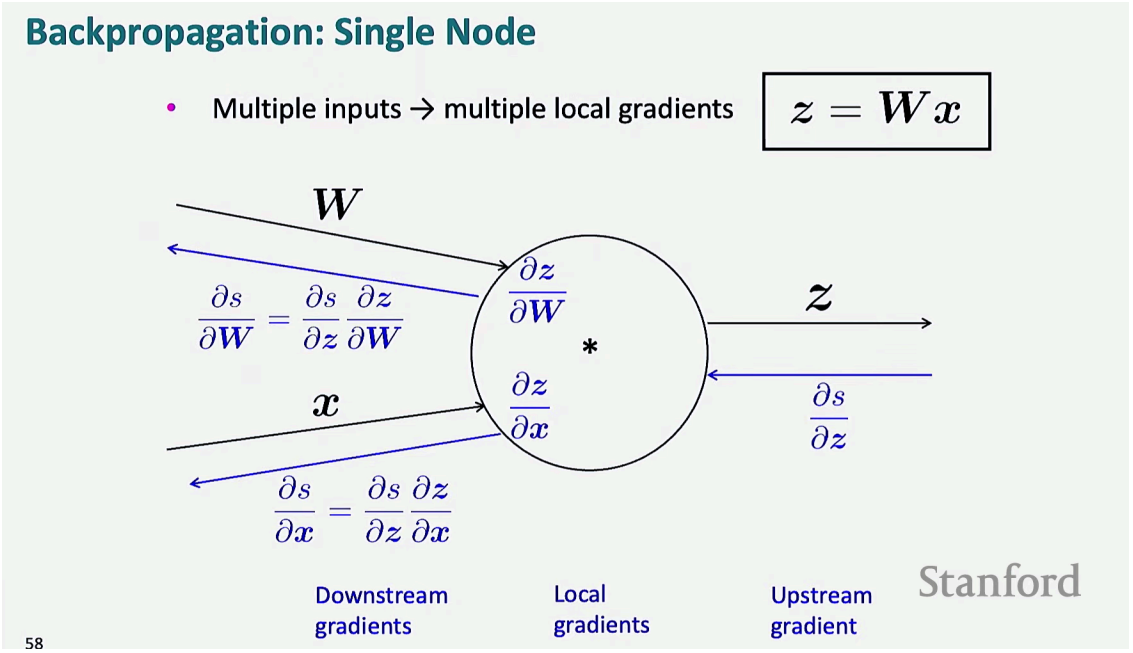
Backpropagation

- Then go backwards along edges
  - Pass along **gradients**

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$



52



Downstream gradients = local gradients * Upstream gradients

Same Big O

1. **정방향 계산 (Forward)**

$$z = Wx + b, \quad h = f(z)$$

시간 복잡도:

- $W \in \mathbb{R}^{n \times m}$, $x \in \mathbb{R}^m$
- → Wx의 계산 비용은 $O(nm)$

2. **역전파 계산 (Backward)**

우리는 다음을 계산해야 함:

- $\frac{\partial L}{\partial W}$, $\frac{\partial L}{\partial b}$, $\frac{\partial L}{\partial x}$
- 이 역시 **행렬-벡터 곱**, **원소곱**, **전치 곱** 등으로 구성되어 있음
- ⇒ 계산량도 **역시** $O(nm)$

Summary

We've mastered the core technology of neural nets! 🎉🎉🎉

- **Backpropagation:** recursively (and hence efficiently) apply the chain rule along computation graph
  - [downstream gradient] = [upstream gradient] x [local gradient]
- **Forward pass:** compute results of operations and save intermediate values
- **Backward pass:** apply chain rule to compute gradients