15-780 – Graduate Artificial Intelligence: Neural networks

Aditi Raghunathan

Recap of supervised ML

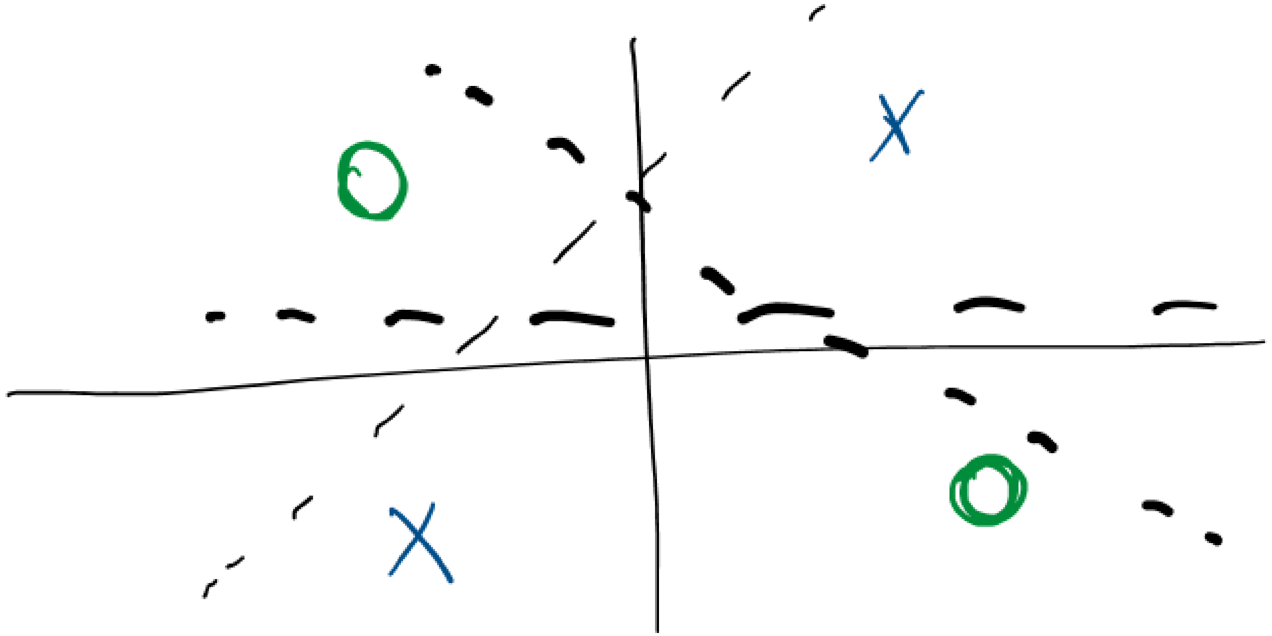
- Hypothesis function $h_\theta(x) = \theta^T x$ $\Longrightarrow$ MLP
- ~~Loss functions~~ $\rightarrow$ cross entropy for classification
- Optimization $\rightarrow$ GD $\rightarrow$ SGD $\rightarrow$ variants like Adam

Linear classifiers

$$h_\theta(x) = \theta^T \underbrace{\cancel{x}}$$ diff features of x

$$= \theta^T \underbrace{\phi(x)}$$ carefully crafted features

Linear classifiers: XOR? → Minsky



Linear classifier -> "feature" learning

$$h_\theta(x) = \theta^T \underset{\sim}{x} \quad \text{new featurization}$$

$$= \theta^T \underset{\sim}{\phi(x)} \quad \text{"hand-crafted"}$$

→ Vision : SIFT, HoG features

→ kernels

$\boxed{\text{Goal: learn features as well!}}$

First attempt at feature learning

original input in $\mathbb{R}^n$

Goal: learn features

$$\underbrace{\phi : \mathbb{R}^n \longrightarrow \mathbb{R}^d}$$ linear

$$h_\theta(x) = \theta^T \phi(x) \qquad \theta \in \mathbb{R}^{d \times k}$$

$$\phi(x) = \theta_2^T x$$

$$\boxed{h_\theta(x) = \theta_1^T \theta_2^T x} \rightarrow \text{features are linear}$$

$$h_\theta(x) = \tilde{\theta}^T x$$

Second attempt

Takeaway: need non-linearity

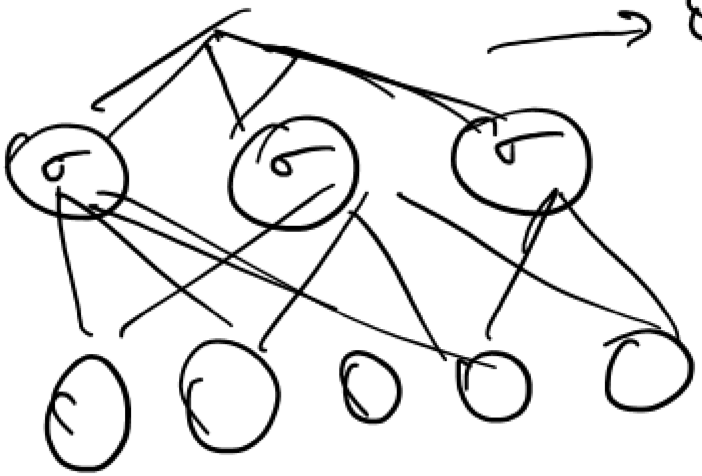
$$\phi(x) = \sigma(\theta_2^T x)$$

$$\boxed{h_\theta(x) = \theta_1^T \sigma(\theta_2^T x)}$$

σ: non-linearity

$\Downarrow$

activation fn

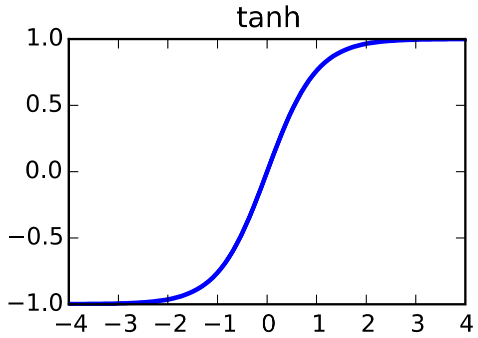


$\rightarrow \theta_1$ layer

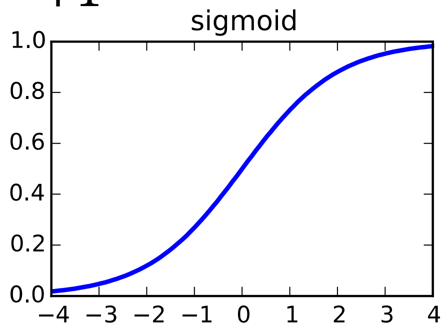
$\theta_2^T x$ layer

Activation functions

Hyperbolic tangent: $f(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$



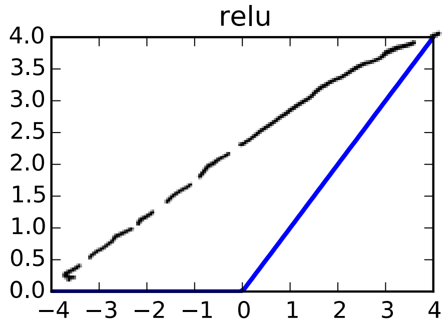
Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$



Rectified linear unit (ReLU): $f(x) = \max\{x, 0\}$

off when $x < 0$

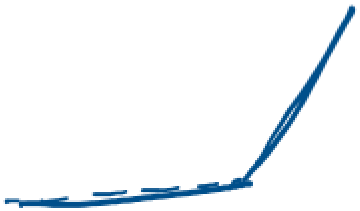
x when $x > 0$



XOR example

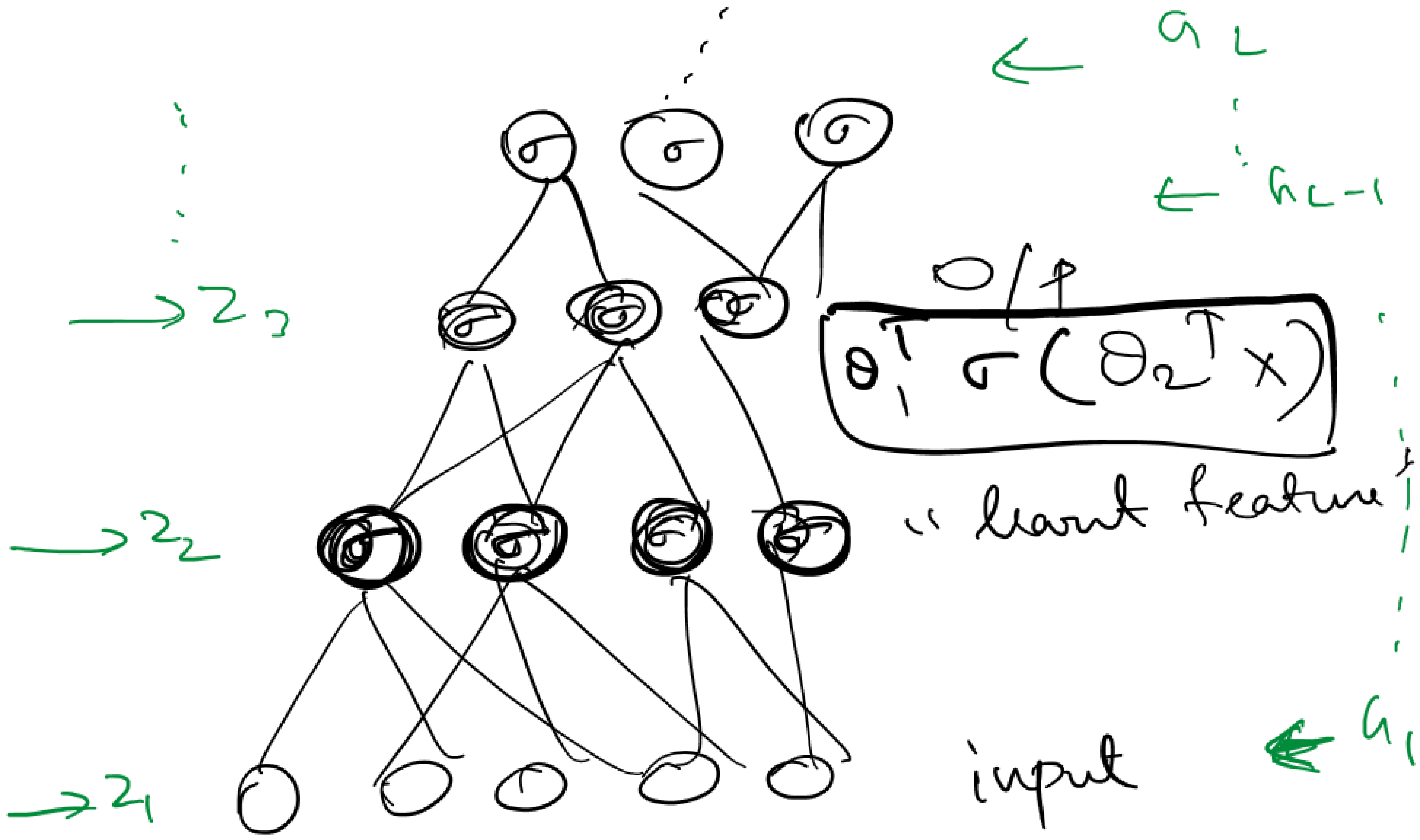
$$h_\theta(x) = \theta_1^T \sigma(\theta_2^T x)$$

σ : ReLU



$$\text{XOR:} \quad \text{ReLU}(x_1 - x_2) + \text{ReLU}(x_2 - x_1)$$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



a_L
a_{L-1}
z_3
z_2
z_1
O/P
$\theta_1^T \sigma(\theta_2^T x)$
"learnt feature"
input
a_1

"Deep" neural networks

- Multi-layer perceptron

repeat the "hidden computation"

$$h_\theta(x) = W_L^T \left(\sigma\left(W_{L-1}^T \sigma\left(\cdots \cdots W_2^T \sigma(W_1^T x) \right)\right)\right) \cdots$$

$$h_\theta(x) = W_2^T \left[\sigma(W_1^T x)\right]$$

parameters "θ": $\{W_1, W_2 \cdots W_L\}$

→ Fully connected network, feed-forward network

Universal function approximation

Theorem (1D case): Given any smooth function $f : \mathbb{R} \to \mathbb{R}$, closed region $\mathcal{D} \subset \mathbb{R}$, and $\epsilon > 0$, we can construct a one-hidden-layer neural network $\hat{f}$ such that

$$\max_{x \in \mathcal{D}} \left| f(x) - \hat{f}(x) \right| \leq \epsilon$$

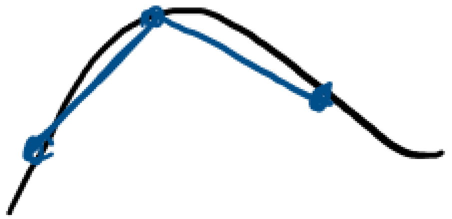
Proof: Select some dense sampling of points $\left(x^{(i)}, f\left(x^{(i)}\right)\right)$ over $\mathcal{D}$. Create a neural network that passes exactly through these points (see below). Because the neural network function is piecewise linear, and the function f is smooth, by choosing the $x^{(i)}$ close enough together, we can approximate the function arbitrarily closely.

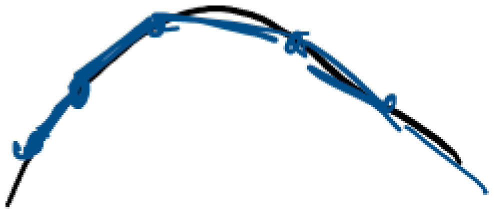
$h(x) = \mathrm{ReLU}(x - a)$

Universal function approximation

Assume one-hidden-layer ReLU network:

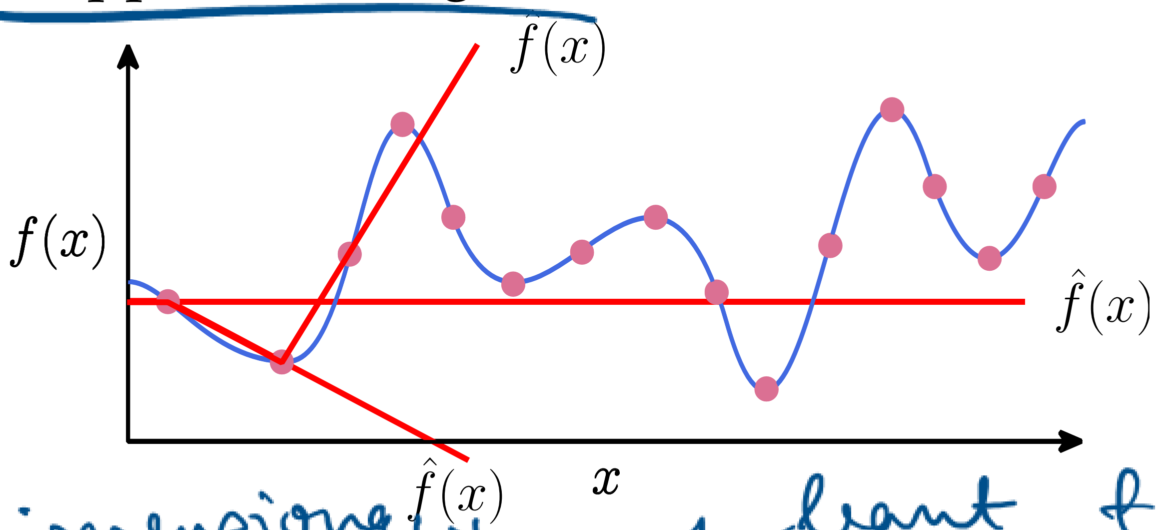
$$\hat{f}(x) = \sum_{i=1}^{d} \pm \max\{0, w_i x + b_i\}$$





Visual construction of approximating function.

# of points or "Kinks" ⇒ # hidden units



(dimensionality of latent feature)

Backpropagation

For SGD or variants, we need to compute **gradients of the loss** with respect to weights (parameters)

The gradient (recap)

A key concept in solving optimization problems is the notation of the gradient of a function (multi-variate analogue of derivative)

Derivative: $$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

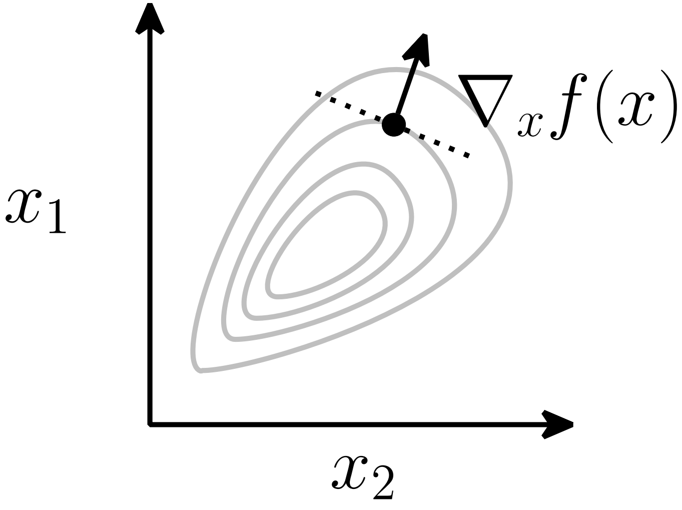
Partial derivative: A partial derivative of a function of several variables is derivative with respect to one of those variables with rest constant

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x + h\mathbf{e_i}) - f(x)}{h}$$

The gradient (recap)

For $f\colon \mathbb{R}^n \to \mathbb{R}$, gradient is defined as vector of partial derivatives

$$\nabla_x f(x) \in \mathbb{R}^n = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$



Points in "steepest direction" of increase in function f

$\boxed{\text{Chain Rule}}$:

$$\frac{\partial f(g(x))}{\partial (x)} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial (x)}$$

$(3x+2)^2$ $\qquad g(x): 3x+2$

$$\frac{\partial f(g(x))}{\partial x} = 2g(x) \cdot \frac{\partial g(x)}{\partial x}$$

$$= 2(3x+2) \cdot 3$$

Deep network notation

Single data point (row vector) $[\longleftarrow x \longrightarrow]$

Input : $X = Z_0 \in \mathbb{R}^{1 \times d_0}$

Intermediate layer : $\underbrace{Z_{i+1}}_{\in \mathbb{R}^{1 \times d_{i+1}}} = \sigma\big(\underbrace{Z_i}_{\in \mathbb{R}^{1 \times d_i}} \overbrace{W_i}^{\in \mathbb{R}^{d_i \times d_{i+1}}}\big)$

Output : $h_\theta(x) = Z_{L+1} \in \mathbb{R}^{k}$

Loss : $\ell(Z_{L+1}, y)$: crossentropy(Z_{L+1}, y)

$$z_{i+1} = \sigma(z_i w_i)$$

$$h_\theta(x) = z_L$$

$$\frac{\partial \ell(h_\theta(x), y)}{\partial w_i} = \frac{\partial \ell(z_{L+1}, y)}{\partial w_i}$$

chain rule

$$= \frac{\partial \ell}{\partial z_{L+1}} \cdot \frac{\partial z_{L+1}}{\partial z_L} \cdot \,\cdots\cdots\, \frac{\partial z_{i+2}}{\partial z_{i+1}} \cdot \frac{\partial z_{i+1}}{\partial w_i}$$

$$\frac{\partial l(z_{L+1}, y)}{\partial w_{i-1}} = \underbrace{\frac{\partial l}{\partial z_{L+1}} \cdot \frac{\partial z_{L+1}}{\partial z_L} \cdots \frac{\partial z_{i+2}}{\partial z_{i+1}}}_{} \cdot \frac{\partial z_{i+1}}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{i-1}}$$

$$\frac{\partial l(z_{L+1}, y)}{\partial w_i} = \underbrace{\frac{\partial l}{\partial z_{L+1}} \cdot \frac{\partial z_{L+1}}{\partial z_L} \cdots \frac{\partial z_{i+2}}{\partial z_{i+1}}}_{} \cdot \frac{\partial z_{i+1}}{\partial w_i}$$

avoid repeated computations

$$\frac{\partial l(z_{L+1}, y)}{\partial z_{i+1}} = G_{i+1}$$

$$G_i = \frac{\partial l(z_{L+1}, y)}{\partial W_i} \in \mathbb{R}^{d_i \times d_{i+1}}$$

$z_i \in \mathbb{R}^{1 \times d_i}$

$z_{i+1} \in \mathbb{R}^{1 \times d_{i+1}}$

$W_i \in \mathbb{R}^{d_i \times d_{i+1}}$

$$\frac{\partial l(z_{L+1}, y)}{\partial z_i} = \underbrace{\frac{\partial l(z_{L+1}, y)}{\partial z_{i+1}}}_{\in \mathbb{R}^{1 \times d_{i+1}}} \cdot \underbrace{\frac{\partial z_{i+1}}{\partial z_i}}_{\in \mathbb{R}^{d_{i+1} \times d_i}}$$

$z_{i+1} = \sigma(z_i W_i)$

$$\boxed{G_i = \Big(\underbrace{G_{i+1}}_{\mathbb{R}^{1 \times d_{i+1}}} \odot \underbrace{\sigma'(z_i W_i)}_{\in \mathbb{R}^{1 \times d_{i+1}}}\Big) \underbrace{W_i^T}_{\mathbb{R}^{d_{i+1} \times d_i}}} \rightarrow \text{Backward iteration}$$

$\odot$: element wise product

$$G_i = \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_i} \qquad G_i = G_{i+1} \odot \sigma'(Z_i W_i) \cdot W_i^T$$

$\dfrac{\partial \ell(Z_{L+1}, y)}{\partial W_i}$: a bit tricky to derive from first principles

"Hack": write down different terms from chain rule and make dimensions match

$$\frac{\partial \ell(Z_{L+1}, y)}{\partial W_i} = \frac{\partial \ell(Z_{L+1}, y)}{\partial Z_{i+1}} \cdot \frac{\partial Z_{i+1}}{\partial W_i}$$

$$= G_{i+1} \quad \sigma'(Z_i W_i) \quad Z_i$$

$\mathbb{R}^{d_i \times d_{i+1}}$ $\mathbb{R}^{1 \times d_{i+1}}$ $\mathbb{R}^{1 \times d_{i+1}}$ $\mathbb{R}^{1 \times d_i}$

$$\frac{\partial \ell}{\partial W_i} = \underbrace{(Z_i)^T}_{\mathbb{R}^{d_i \times 1}} \Big(\underbrace{G_{i+1}}_{\mathbb{R}^{1 \times d_{i+1}}} \odot \sigma'(Z_i W_i) \Big)$$

Batched version : m samples at a time independently (talk more in Lec 9)

$Z_i \in \mathbb{R}^{m \times d_i}$

$$\begin{bmatrix} \leftarrow & z_i^{(1)} & \rightarrow \\ \leftarrow & z_i^{(2)} & \rightarrow \\ & \vdots & \\ \leftarrow & z_i^{(m)} & \rightarrow \end{bmatrix}$$

$G_i \in \mathbb{R}^{m \times d_i}$

$$G_i = \underbrace{G_{i+1}}_{\mathbb{R}^{m \times d_{i+1}}} \odot \underbrace{\sigma'(Z_i W_i)}_{\mathbb{R}^{m \times d_{i+1}}} \underbrace{W_i^T}_{\mathbb{R}^{d_{i+1} \times d_i}}$$

$$\underbrace{\frac{\partial \ell}{\partial W_i}}_{\mathbb{R}^{d_i \times d_{i+1}}} = \underbrace{(Z_i)^T}_{\mathbb{R}^{d_i \times m}} \Big(\underbrace{G_{i+1} \odot \sigma'(Z_i W_i)}_{\mathbb{R}^{m \times d_{i+1}}} \Big)$$

Terminology

z_i = intermediate values in the network $\Downarrow$ aka activations

z_1 = input $\qquad z_L \equiv$ output

$$z_i = \underbrace{\sigma}_{\text{activation function}}\left(W_{i-1}^T z_{i-1}\right)$$

W_i : "weights" & "parameters" → have to be estimated from data

learning process: find best W_i that $\underbrace{\text{fit data}}_{\text{loss}(z_L^{(i)},\, y^{(i)})}$

$x^{(i)} \longrightarrow z_L^{(i)}$

Backpropagation algorithm

- Initialize $Z_1 = X$
- For $i = 1 \ldots L$

$$Z_{i+1} = \sigma(Z_i W_i)$$

Forward pass

- For $i = L \ldots 1$ G_L: gradient of loss function (cross entropy)

$$G_i = G_{i+1} \odot \sigma'(Z_i W_i) W_i^T$$

Backward pass

$$\frac{\partial l}{\partial W_i} = Z_i^T G_{i+1} \odot \sigma'(W_i^T Z_i)$$

each element

Training algo: $W^{(t)} = W^{(t-1)} - \eta \boxed{\nabla_W l}$

$[W_1, W_2 \ldots W_L]$

Tr Loss: $\sum_{i=1}^{n} loss(x^{(i)}, y^{(i)})$

∇ Tr Loss $= \nabla \sum = \sum \nabla$

<u>Stochastic</u>: gradient of a batch of samples

Computation graph

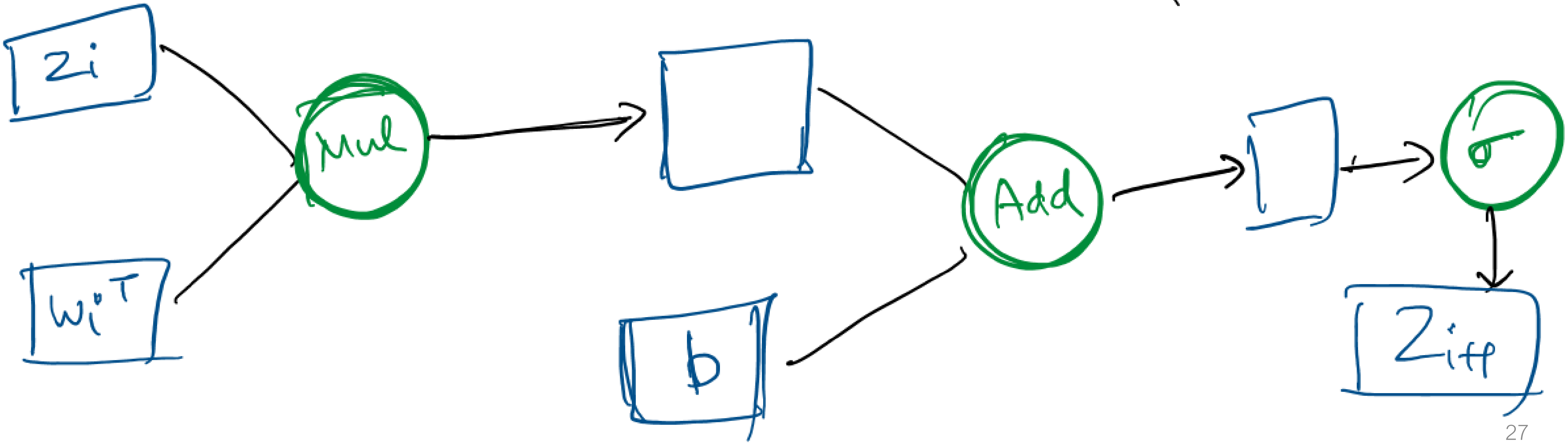
□ : variables

○ : functions

Directed Acyclic Graph

$$\cancel{z_{i+1} = \sigma(w_i^T z_i)}$$

$$z_{i+1} = \sigma(w_i^T z_i + b_i)$$



Computation graph

Directed Acyclic Graph to represent the functions computed

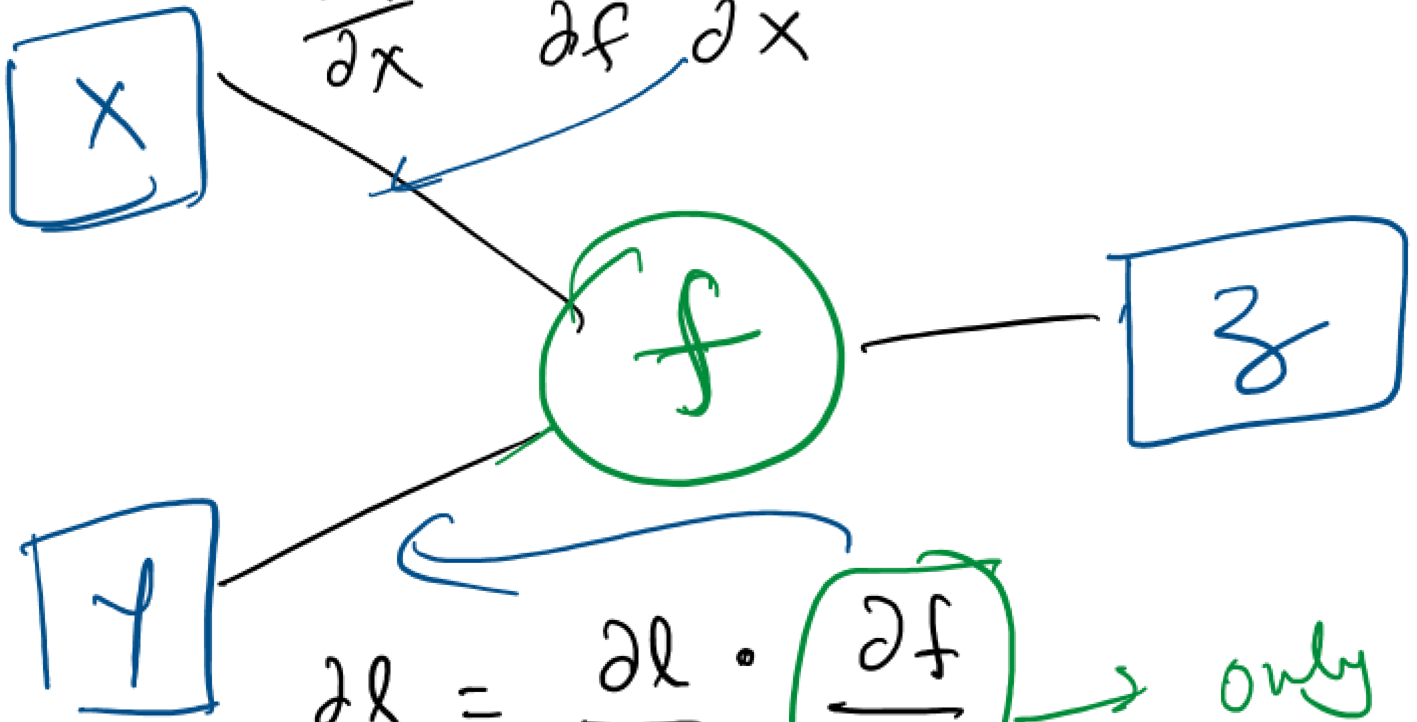
- Root node has the final expression, intermediate nodes have subexpressions

- Convenient to compute gradients, used in popular frameworks like pytorch and tensorflow

Computation graph example

Computation graph chain rule

$$\frac{\partial l}{\partial x} = \frac{\partial l}{\partial f}\cdot\frac{\partial f}{\partial x}$$



$$\frac{\partial l}{\partial z}$$

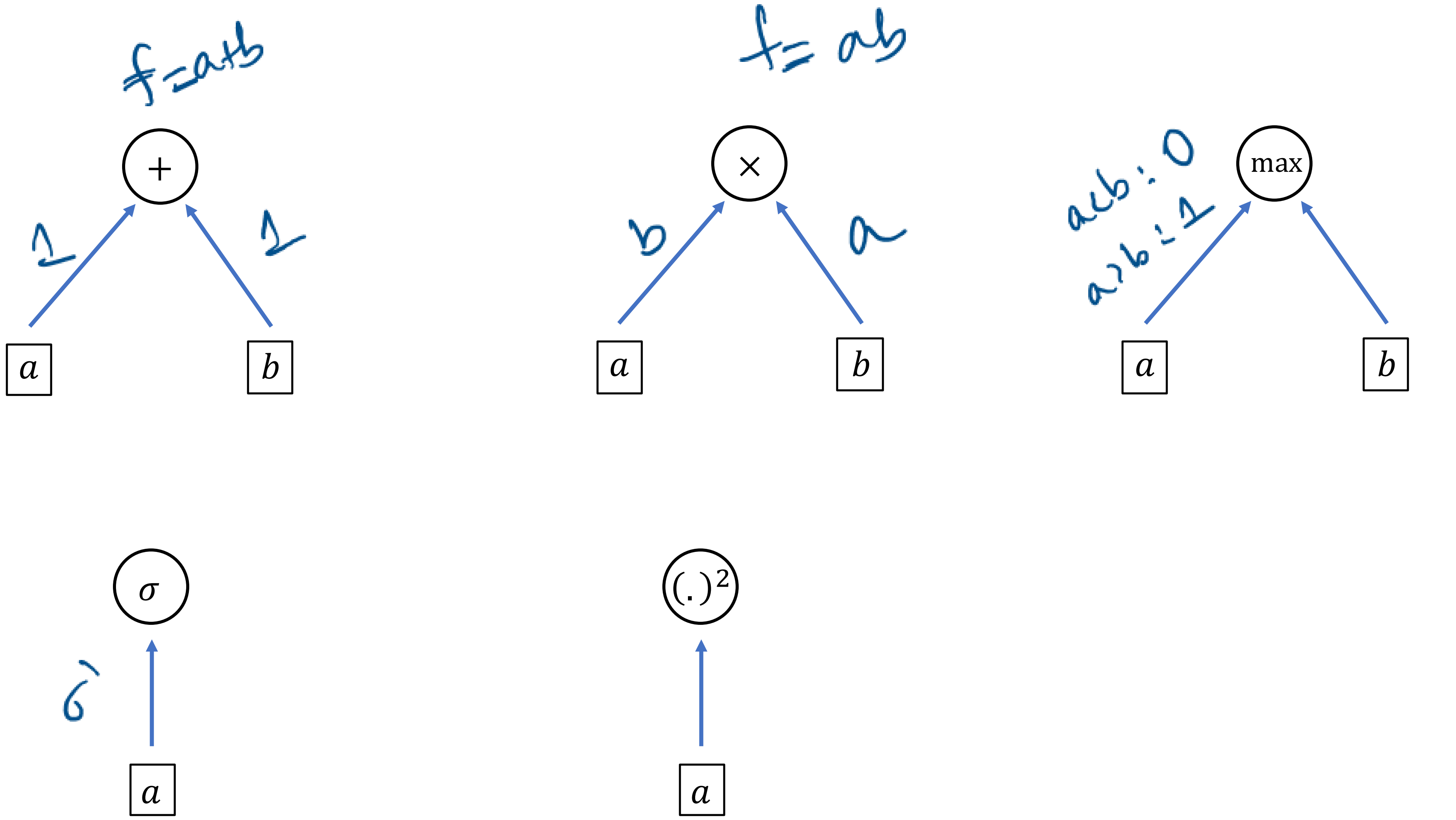
given

$$\frac{\partial l}{\partial y} = \frac{\partial l}{\partial z}\cdot\boxed{\frac{\partial f}{\partial y}}$$

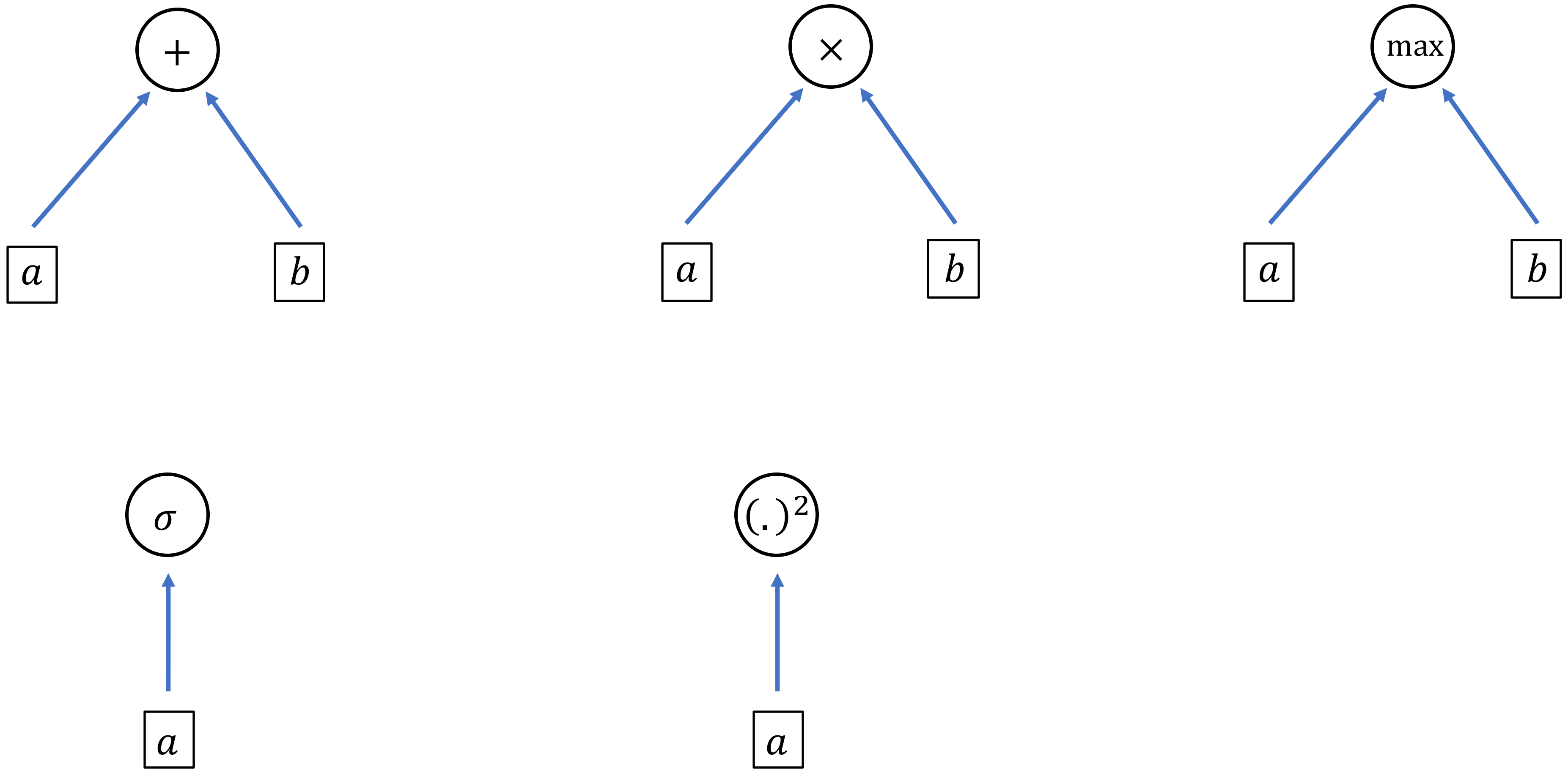
→ only depends on f, x, y

$\boxed{\text{Modular}}$

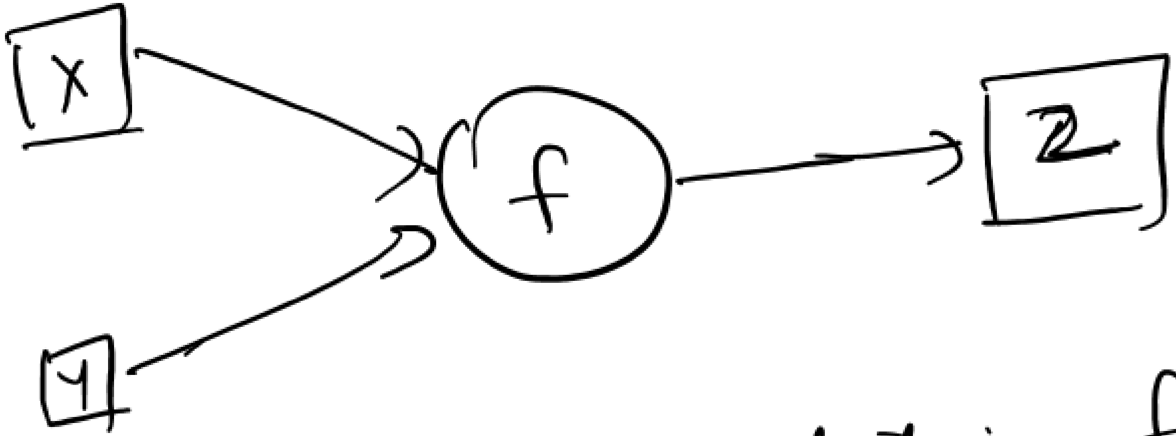
Computation graph: gradients along edges



Computation graph



Auto diff



Forward pass: implementation $f(x, y)$

Backward pass: Given $\frac{\partial l}{\partial z}$ vector

$\frac{\partial l}{\partial z} \cdot \frac{\partial z}{\partial x}$,

$\frac{\partial l}{\partial z} \cdot \frac{\partial z}{\partial y}$ ...

in general: inputs : $[x_1, x_2 \cdots x_n]$

output: $[z_1 \cdots z_d]$

$$\underbrace{\begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_2}{\partial x_1} & \cdots \\ & \ddots & \\ & & \frac{\partial z_d}{\partial x_n} \end{bmatrix}}_{\text{jacobian}} \longrightarrow \text{high dim} \quad f: \mathbb{R}^n \longrightarrow \mathbb{R}^d$$

Multi-layer perceptron recap

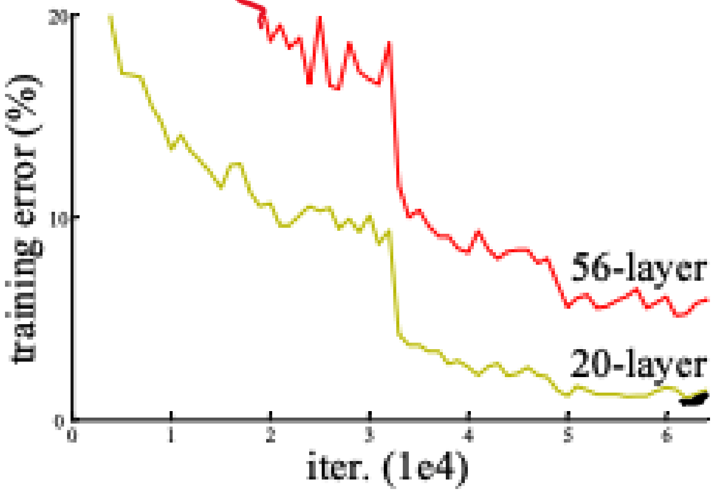
→ feedforward networks

→ " ~~more~~ layers " are better

→ Perceptron by Minski ≡ AI winter

Piazza poll question

56 layer



optimization issue

∃ 56-layer network

identity

20 layer

20 layer

Piazza poll question

→ ∃ a 56 layer network with low train error

[identity]

20 layer ["good"]

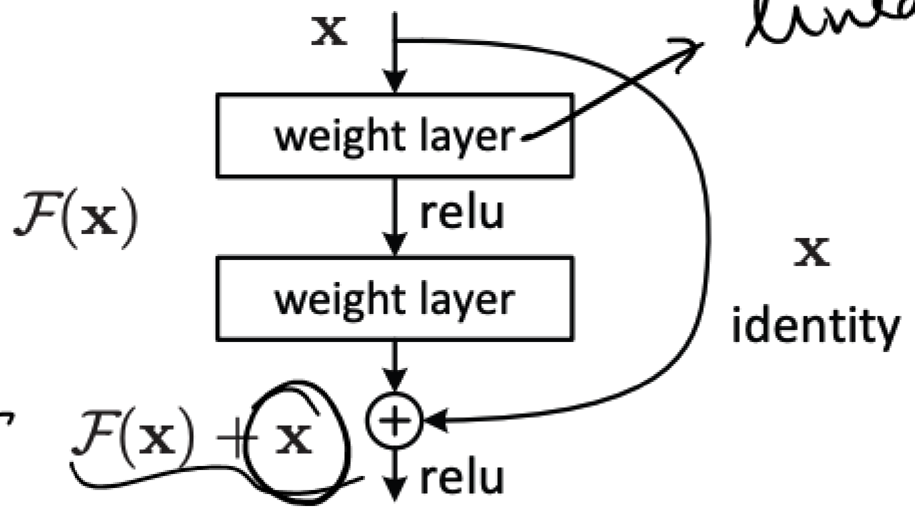
optimization issue

vanishing gradient: each "edge" = a multiplication and it could make the gradient really small

exploding

Residual connection

linear layer



$$\left[\frac{\partial f}{\partial x} + I \right]$$

Figure 2. Residual learning: a building block.

Residual connection

Two interpretations

1. Easier to preserve identity / good features

2. Addresses vanishing gradients due to “shortcut” connections

Normalization

→ z_i's also change a lot : compounding across layers

generic form:

$z_i \in \mathbb{R}^d$

⇓

z_i are activation of a layer ≡ layer norm

$$\hat{z}_i = \left(\frac{z_i - \mathbb{E}[z_i]}{\sqrt{\mathrm{var}(z_i) + \epsilon}} \right) \cdot \gamma + \lambda$$

← to avoid divide by zero

$\gamma, \lambda \in \mathbb{R}^d$

λ: bias
γ: scale

learnable

The canonical architecture: transformers