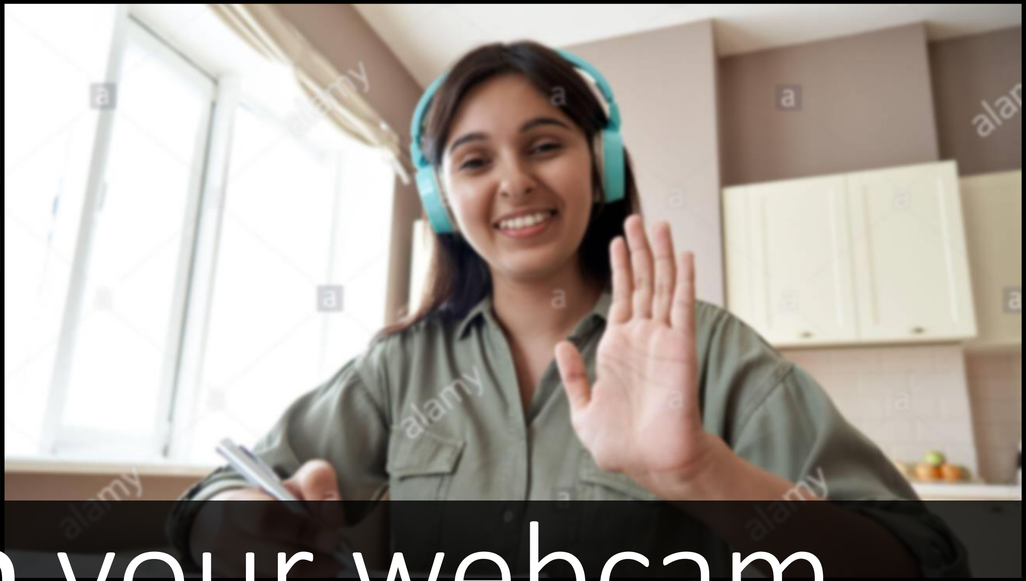
Ask Me Anything (AMA)

9B

CS 3244 Machine Learning

NUS | Computing
National University of Singapore



Please **turn on** your webcam

Mystery Student

Week 09A: Lecture Outline

1. Math Notation Primer
2. Backprop Revisiting
3. AMA

Average difference metrics **for test dataset**

Mean Absolute Error (MAE)

$$MAE = \frac{1}{m}\sum_{j=1}^{m}\left|\hat{y}_j - y_j\right|$$

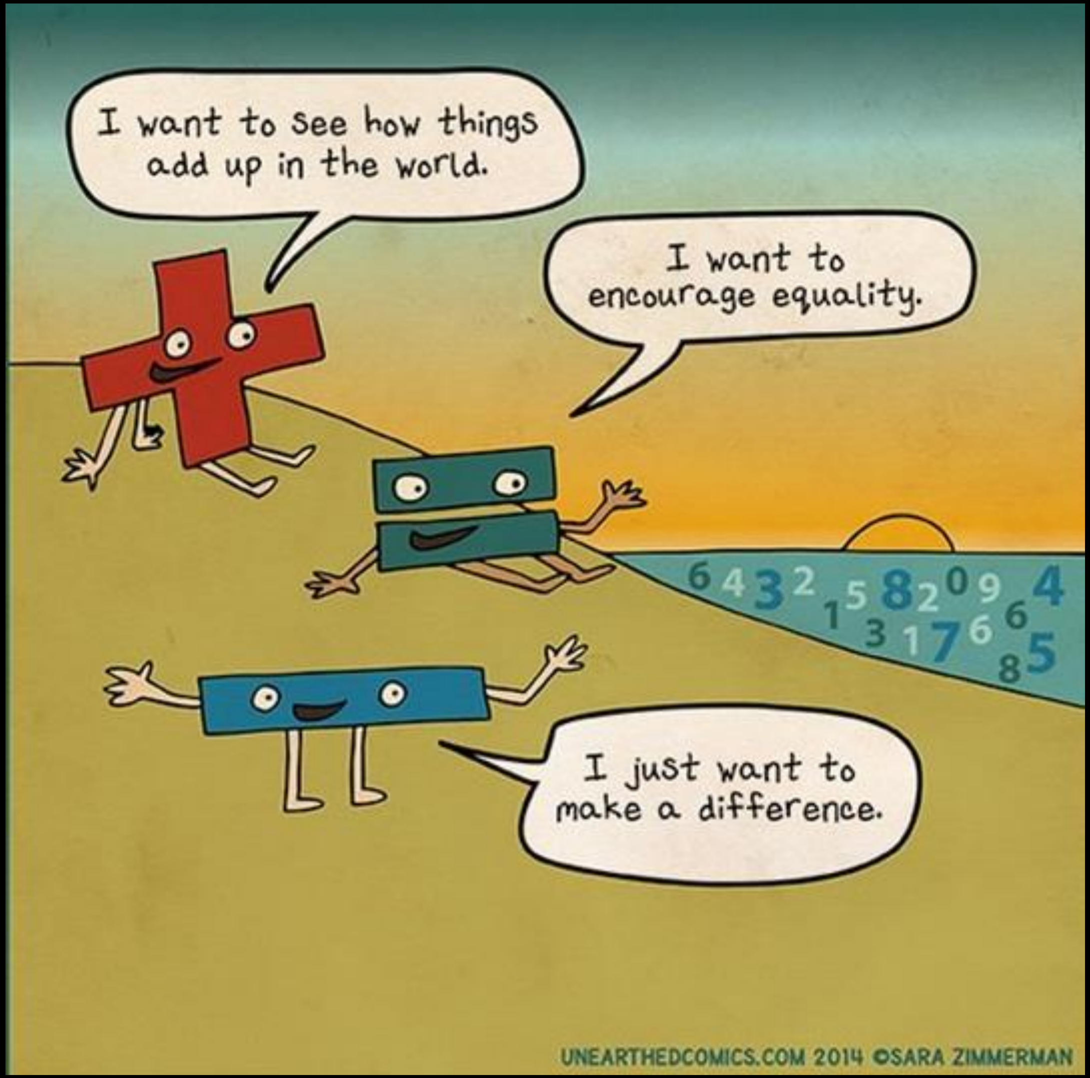
Mean Squared Error (MSE)

$$MSE = \frac{1}{m}\sum_{j=1}^{m}\left(\hat{y}_j - y_j\right)^2$$

Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\frac{1}{m}\sum_{j=1}^{m}\left(\hat{y}_j - y_j\right)^2}$$

MSE and RMSE penalize larger differences **more** than MAE



Source: https://themasterstable.files.wordpress.com/2014/09/math-jokes.png

Why Math in Machine Learning?

Idea → Math → Coding

Math enables **deterministic** (algorithmic) execution (implementation)

Math helps to **check idea** is correct (logically consistent)

Math helps to **check coding** is correct (bug free)

Notation

n = Number of features in $\boldsymbol{x}$
m = Number of instances in dataset

- **Scalar**: not bolded, lower case

$$x$$

- **Vector**: bolded, lower case

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

- **Matrix**: bolded, upper case

$$\boldsymbol{X} = \begin{pmatrix} x_{11} & \cdots & x_{m1} \\ \vdots & \ddots & \vdots \\ x_{1n} & \cdots & x_{mn} \end{pmatrix}$$

Functions with Vectors and Matrices

- Scalar-by-scalar:
  - $y(x) = wx$ for scaling input
- Scalar-by-vector:
  - $y(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w}^\top \boldsymbol{x} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = w_1 x_1 + w_1 x_2 + w_2 x_1 + w_2 x_2$ for **weighted sum**
- Vector-by-vector:
  - $\boldsymbol{y}(\boldsymbol{x}) = w\boldsymbol{x} = w\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} wx_1 \\ wx_2 \end{pmatrix}$ for **scaled** outputs (same weight)

Functions with Vectors and Matrices

- Vector-by-matrix:
  - $\boldsymbol{y}(\boldsymbol{X}) = \boldsymbol{W} \circ \boldsymbol{X} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} = \begin{pmatrix} w_{11}x_{11} & w_{12}x_{12} \\ w_{21}x_{21} & w_{22}x_{22} \end{pmatrix}$
  - Using **Hadamard** product $\circ$ for element-wise multiplication
  - For backpropagation (see later), convenient multiplication of corresponding elements between matrices
- Matrix-by-matrix:
  - $\boldsymbol{Y}(\boldsymbol{X}) = \boldsymbol{W} * \boldsymbol{X} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix}$

$$= \begin{pmatrix} w_{11}x_{11} + w_{12}x_{12} + w_{21}x_{21} + w_{22}x_{22} & w_{11}x_{12} + w_{12}x_{13} + w_{21}x_{22} + w_{22}x_{23} \\ w_{11}x_{21} + w_{12}x_{22} + w_{21}x_{31} + w_{22}x_{32} & w_{11}x_{22} + w_{12}x_{23} + w_{21}x_{32} + w_{22}x_{33} \end{pmatrix}$$

  - Using **Convolution** operator $*$ for element-wise multiplication then sum [W08b]
  - For computer vision filters (kernels)

Weighted Sum

Summation Series = Scalar

$$\sum_{r=0}^{n} w_r x_r$$

$$w_1 x_1 + \cdots + w_r x_r + \cdots + w_n x_n$$

Transposed Vector Multiplication = Scalar

$$\boldsymbol{w}^\top \boldsymbol{x} = \begin{pmatrix} w_1 & \cdots & w_r & \cdots & w_n \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_r \\ \vdots \\ x_n \end{pmatrix}$$

Vector Dot Product = Scalar

$$\boldsymbol{w} \cdot \boldsymbol{x} = \begin{pmatrix} w_1 \\ \vdots \\ w_r \\ \vdots \\ w_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_r \\ \vdots \\ x_n \end{pmatrix}$$

Transposed Matrix Multiplication = Vector

$$\boldsymbol{W}^\top \boldsymbol{x} = \begin{pmatrix} w_{11} & \cdots & w_{1r} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots & ⋰ & \vdots \\ w_{r1} & \cdots & w_{rr} & \cdots & w_{rn} \\ \vdots & ⋰ & \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nr} & \cdots & w_{nn} \end{pmatrix}^\top \begin{pmatrix} x_1 \\ \vdots \\ x_r \\ \vdots \\ x_n \end{pmatrix}$$

Gradient

- **Derivative**: d
  - $\frac{dy}{dx}$ is the derivative of y relative to x
- **Partial derivative**: ∂
  - $\frac{\partial y}{\partial x_1}$ is the derivative of y relative to x_1
  - But y also depends on other variables (e.g., x_2 so, we can also calculate $\frac{\partial y}{\partial x_2}$)
- **Gradient**: ∇
  - To calculate the derivative relative to *all* x_1 and x_2 together
  - $\nabla y(\boldsymbol{x})$ is the gradient of y relative to all variables $\boldsymbol{x} = (x_1, \ldots, x_n)^\top$

Vector in denominator means Derivative for each variable is put in separate, corresponding variable dimension

$$\nabla y(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{pmatrix} \partial y / \partial x_1 \\ \vdots \\ \partial y / \partial x_n \end{pmatrix} = \begin{pmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{pmatrix}^\top$$

  - Assumes Cartesian coordinates (linear, orthogonal)

Matrix Calculus – not in exam

n = Number of features in $\boldsymbol{x}$
m = Number of instances in dataset
N = Number of y prediction tasks

Scalar-by-Vector (= 1D Vector)

$$\frac{\partial y}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_n} \end{pmatrix}$$

Scalar-by-Matrix (= 2D Matrix)

$$\frac{\partial y}{\partial \boldsymbol{X}} = \begin{pmatrix} \frac{\partial y}{\partial x_{11}} & \cdots & \frac{\partial y}{\partial x_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial x_{n1}} & \cdots & \frac{\partial y}{\partial x_{nm}} \end{pmatrix}$$

Vector-by-Vector (= 2D Matrix)

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_N}{\partial x_n} \end{pmatrix}$$

Vector-by-Matrix (= 3D Matrix)

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{X}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_{11}} & \cdots & \frac{\partial y_1}{\partial x_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_{n1}} & \cdots & \frac{\partial y_1}{\partial x_{nm}} \end{pmatrix} \ddots \begin{pmatrix} \frac{\partial y_N}{\partial x_{11}} & \cdots & \frac{\partial y_N}{\partial x_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_N}{\partial x_{n1}} & \cdots & \frac{\partial y_N}{\partial x_{nm}} \end{pmatrix}$$

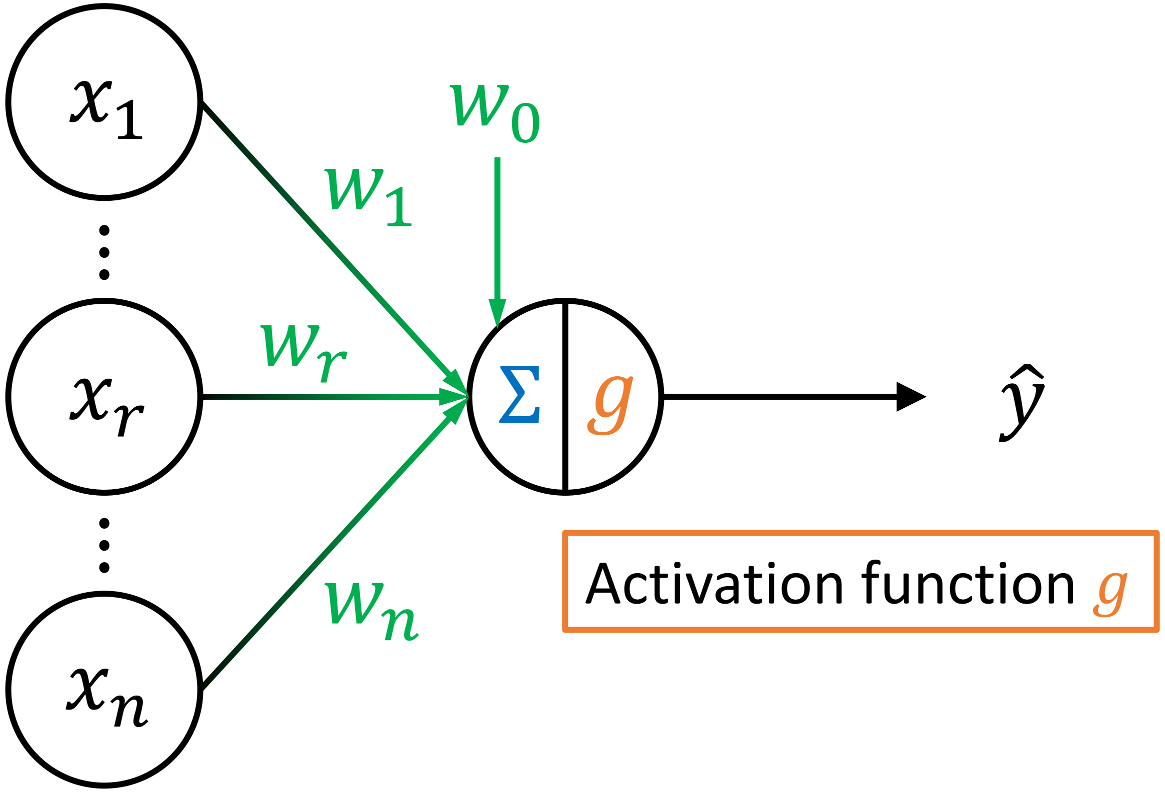
Along 3rd dimension

This math informs what matrix **shapes** you need to implement

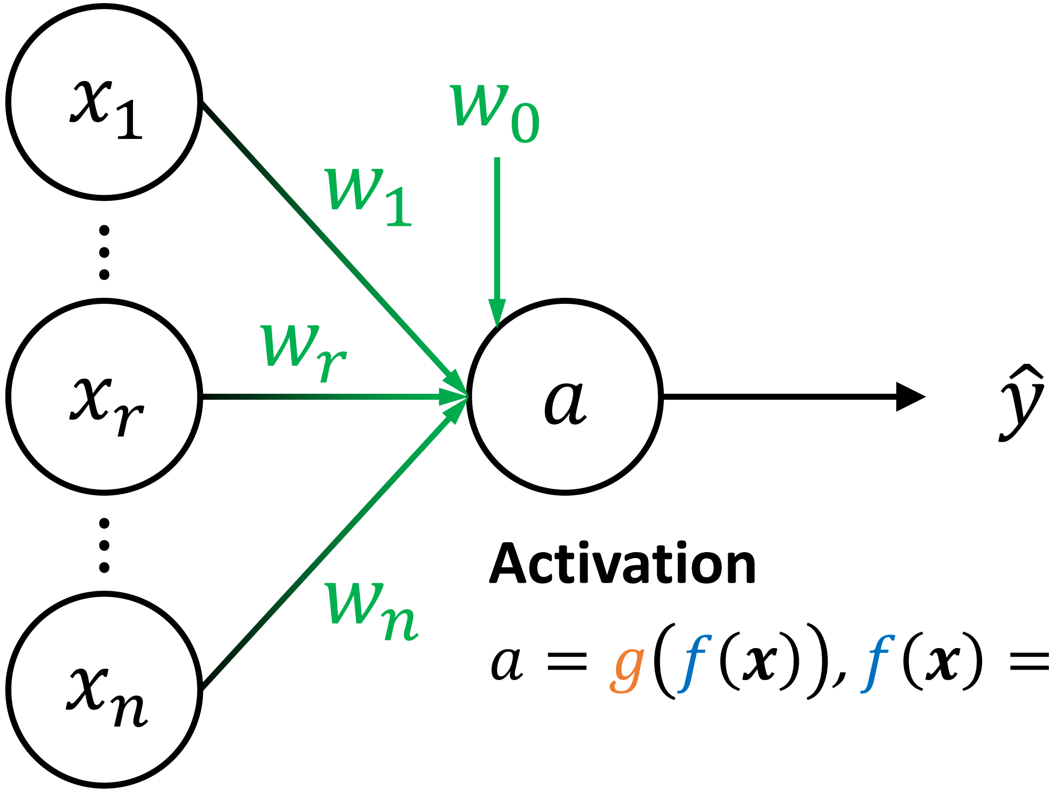
Backpropagation
“backprop”

Department of Computer Science
School of Computing

Single-Layer Perceptron



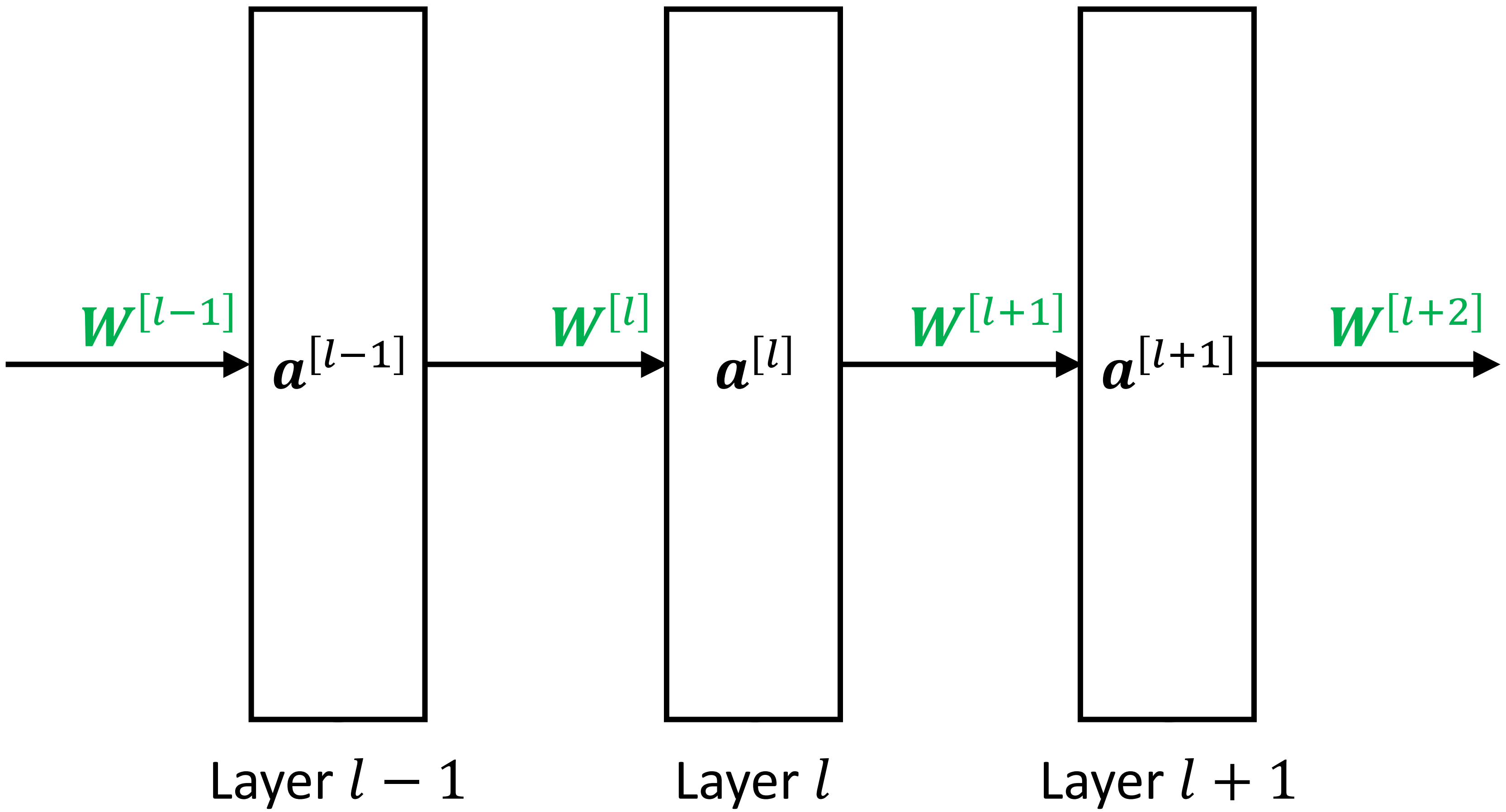
Single-Layer Perceptron



Activation

$a = g(f(\boldsymbol{x})), f(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x}$

Neural Network



Layer Activation

$$a = g(f(x)), f(x) = \boldsymbol{w} \cdot \boldsymbol{x}$$

Single-Layer Perceptron

Layer l Activation Function

Layer l Weights

$$\boldsymbol{a}^{[l]} = g^{[l]}\left(\left(\boldsymbol{W}^{[l]}\right)^{\mathsf{T}} \boldsymbol{a}^{[l-1]}\right)$$

Layer l in Neural Network

Layer l Activations

Layer $l-1$ Activations

Forward Propagation

$$g^{[1]}\left(\left(\boldsymbol{W}^{[1]}\right)^{\top} x^{[0]}\right) = \boldsymbol{a}^{[1]} \qquad g^{[l]}\left(\left(\boldsymbol{W}^{[l]}\right)^{\top} a^{[l-1]}\right) = \boldsymbol{a}^{[l]} \qquad g^{[L]}\left(\left(\boldsymbol{W}^{[L]}\right)^{\top} a^{[L-1]}\right) = \boldsymbol{a}^{[L]}$$

$$\boxed{\boldsymbol{x}^{[0]}} \quad \cdots \quad \xrightarrow{\boldsymbol{W}^{[l-1]}} \boxed{\boldsymbol{a}^{[l-1]}} \xrightarrow{\boldsymbol{W}^{[l]}} \boxed{\boldsymbol{a}^{[l]}} \xrightarrow{\boldsymbol{W}^{[l+1]}} \boxed{\boldsymbol{a}^{[l+1]}} \quad \cdots \quad \xrightarrow{\boldsymbol{W}^{[L]}} \boxed{\boldsymbol{a}^{[L]}} = \hat{y}$$

$$\hat{y}(\boldsymbol{x}) = g^{[L]}\left(\left(\boldsymbol{W}^{[L]}\right)^{\top} g^{[L-1]}\left(\cdots\left(g^{[l]}\left(\left(\boldsymbol{W}^{[l]}\right)^{\top} g^{[l-1]}\left(\cdots\left(g^{[1]}\left(\left(\boldsymbol{W}^{[1]}\right)^{\top} \boldsymbol{x}^{[0]}\right)\right)\right)\right)\right)\right)\right)$$

Forward Propagation

$$\boldsymbol{a}^{[l]} \equiv g^{[l]}(f^{[l]}),\ f^{[l]} \equiv \left(\boldsymbol{W}^{[l]}\right)^{\top} \boldsymbol{a}^{[l]}$$

$$g^{[1]}\left(f^{[1]}\left(x^{[0]}\right)\right) = \boldsymbol{a}^{[1]} \qquad g^{[l]}\left(f^{[l]}\left(\boldsymbol{a}^{[l-1]}\right)\right) = \boldsymbol{a}^{[l]} \qquad g^{[L]}\left(f^{[L]}\left(\boldsymbol{a}^{[L-1]}\right)\right) = \boldsymbol{a}^{[L]}$$

$$\boxed{\boldsymbol{x}^{[0]}} \quad \cdots \quad \xrightarrow{\boldsymbol{W}^{[l-1]}} \boxed{\boldsymbol{a}^{[l-1]}} \xrightarrow{\boldsymbol{W}^{[l]}} \boxed{\boldsymbol{a}^{[l]}} \xrightarrow{\boldsymbol{W}^{[l+1]}} \boxed{\boldsymbol{a}^{[l+1]}} \quad \cdots \quad \xrightarrow{\boldsymbol{W}^{[L]}} \boxed{\boldsymbol{a}^{[L]}} = \hat{y}$$

$$\hat{y}(\boldsymbol{x}) = g^{[L]}(f^{[L]}(g^{[L-1]}(\cdots(g^{[l]}(f^{[l]}(g^{[l-1]}(\cdots(g^{[1]}(f^{[1]}(\boldsymbol{x}^{[0]})))))))))$$

Even more Chain Rule:

Gradient of Neural Network

$$\hat{y}(\boldsymbol{x}) = g^{[L]}(f^{[L]}(g^{[L-1]}(\cdots(g^{[l]}(f^{[l]}(g^{[l-1]}(\cdots(g^{[1]}(f^{[1]}(\boldsymbol{x}^{[0]})))))))))$$

Gradient relative to $\boldsymbol{W}$

$$\hat{y}'(\boldsymbol{W}^{[L-1]}) = \frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[L-1]}} = \frac{\partial f^{[L]}}{\partial \boldsymbol{W}^{[L-1]}} \underbrace{\boxed{\frac{\partial g^{[L]}}{\partial f^{[L]}}}}_{\delta^{[L-1]}}$$

Reference

$$\boldsymbol{a}^{[l]} = g^{[l]}(f^{[l]})$$

$$f^{[l]} = \left(W^{[l]}\right)^{\top} \boldsymbol{a}^{[l-1]}$$

$$\hat{y}'(\boldsymbol{W}^{[l+1]}) = \frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[l+1]}} = \frac{\partial f^{[l+1]}}{\partial \boldsymbol{W}^{[l+1]}} \underbrace{\boxed{\frac{\partial g^{[l+1]}}{\partial f^{[l+1]}} \cdots \frac{\partial f^{[L]}}{\partial g^{[L-1]}} \frac{\partial g^{[L]}}{\partial f^{[L]}}}}_{\delta^{[l+1]}}$$

$$\hat{y}'(\boldsymbol{W}^{[l]}) = \frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[l]}} = \frac{\partial f^{[l]}}{\partial \boldsymbol{W}^{[l]}} \frac{\partial g^{[l]}}{\partial f^{[l]}} \frac{\partial f^{[l+1]}}{\partial g^{[l]}} \underbrace{\frac{\partial g^{[l+1]}}{\partial f^{[l+1]}} \cdots \frac{\partial f^{[L]}}{\partial g^{[L-1]}} \frac{\partial g^{[L]}}{\partial f^{[L]}}}$$

Recursive

$$\hat{y}'(\boldsymbol{W}^{[l]}) = \frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[l]}} = \frac{\partial f^{[l]}}{\partial \boldsymbol{W}^{[l]}} \frac{\partial g^{[l]}}{\partial f^{[l]}} \frac{\partial f^{[l+1]}}{\partial g^{[l]}} \delta^{[l+1]}$$

$$\hat{y}'(\boldsymbol{W}^{[1]}) = \frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[1]}} = \frac{\partial f^{[1]}}{\partial \boldsymbol{W}^{[1]}} \frac{\partial g^{[1]}}{\partial f^{[1]}} \cdots \frac{\partial g^{[l]}}{\partial f^{[l]}} \frac{\partial f^{[l+1]}}{\partial g^{[l]}} \frac{\partial g^{[l+1]}}{\partial f^{[l+1]}} \cdots \frac{\partial f^{[L]}}{\partial g^{[L-1]}} \frac{\partial g^{[L]}}{\partial f^{[L]}}$$

Even more Chain Rule:

Gradient of Neural Network

$$\hat{y}(\boldsymbol{x}) = g^{[L]}(f^{[L]}(g^{[L-1]}(\cdots(g^{[l]}(f^{[l]}(g^{[l-1]}(\cdots(g^{[1]}(f^{[1]}(\boldsymbol{x}^{[0]})))))))))$$

Gradient relative to $\boldsymbol{W}^{[l]}$

$$\hat{y}'(\boldsymbol{W}^{[l]}) = \frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[l]}} = \frac{\partial f^{[l]}}{\partial \boldsymbol{W}^{[l]}} \frac{\partial g^{[l]}}{\partial f^{[l]}} \frac{\partial f^{[l+1]}}{\partial g^{[l]}} \delta^{[l+1]}$$

Reference

$$\boldsymbol{a}^{[l]} = g^{[l]}(f^{[l]})$$

$$f^{[l]} = (W^{[l]})^{\top} \boldsymbol{a}^{[l-1]}$$

$$\boxed{\frac{\partial f^{[l]}}{\partial \boldsymbol{W}^{[l]}} = \boldsymbol{a}^{[l-1]}} \quad \boxed{\frac{\partial g^{[l]}}{\partial f^{[l]}} = g'^{[l]}(f^{[l]})} \quad \boxed{\frac{\partial f^{[l+1]}}{\partial g^{[l]}} = \frac{\partial f^{[l+1]}}{\partial \boldsymbol{a}^{[l]}} = W^{[l+1]}}$$

$$\hat{y}'(\boldsymbol{W}^{[l]}) = \boldsymbol{a}^{[l-1]} \ \underbrace{\boxed{g'^{[l]}(f^{[l]}) \ \boldsymbol{W}^{[l+1]} \ \delta^{[l+1]}}}_{\delta^{[l]}}$$

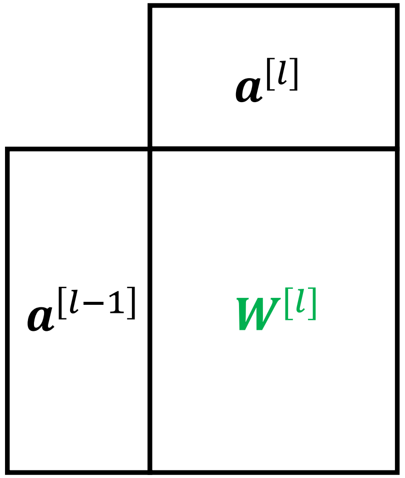
$$\boxed{\hat{y}'(\boldsymbol{W}^{[l]}) = \boldsymbol{a}^{[l-1]} \ \delta^{[l]}} \quad \boxed{\delta^{[l]} = g'^{[l]}(f^{[l]}) \ \boldsymbol{W}^{[l+1]} \ \delta^{[l+1]}}$$

Recursive

Matrix multiplication to match shape (not in exam)

nRows × nCols	$n^{[l-1]} \times n^{[l]}$	$n^{[l-1]} \times 1$	$n^{[l]} \times 1$	$n^{[l]} \times n^{[l+1]}$	$n^{[l+1]} \times 1$

$$\hat{y}'(\boldsymbol{W}^{[l]}) = \boldsymbol{a}^{[l-1]} \quad \delta^{[l]} \qquad \delta^{[l]} = g'^{[l]}(f^{[l]}) \quad \underbrace{\boldsymbol{W}^{[l+1]} \quad \delta^{[l+1]}}$$



$n^{[l]} \times 1$

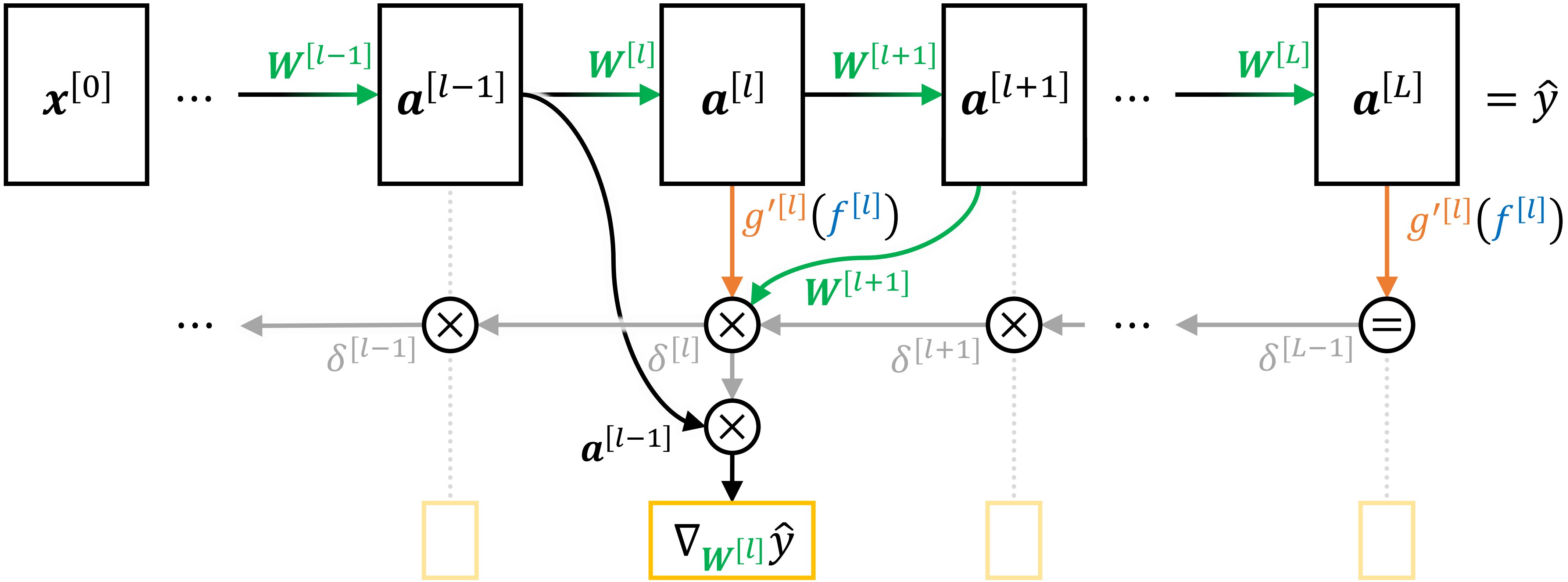
$n^{[l]} \times 1$ $\quad$ $n^{[l]} \times 1$

Hadamard product on same-shape matrices, since we are multiplying each dimension independently

$1 \times n^{[l]}$

$$\hat{y}'(\boldsymbol{W}^{[l]}) = \boldsymbol{a}^{[l-1]}\left(\delta^{[l]}\right)^{\top} \qquad \delta^{[l]} = \left[g'^{[l]}(f^{[l]})\right] \circ \left(\boldsymbol{W}^{[l+1]}\delta^{[l+1]}\right)$$

Backward Propagation



$\hat{y}'(\boldsymbol{W}^{[l]}) = \boldsymbol{a}^{[l-1]}\left(\boldsymbol{\delta}^{[l]}\right)^{\top}$ | $\boldsymbol{\delta}^{[l]} = \left[g'^{[l]}(f^{[l]})\right] \circ \left(\boldsymbol{W}^{[l+1]}\boldsymbol{\delta}^{[l+1]}\right)$

Backpropagation

Backpropagation **efficiently** computes the **gradient** by

- Avoiding duplicate calculations
- Not computing unnecessary intermediate values,
- Computing the **gradient** of ***each* layer**

Specifically, the gradient of the weighted input of each layer is calculated from back $[l+1]$ to front $[l]$):

$$\hat{y}'\left(\boldsymbol{W}^{[l]}\right) = \boldsymbol{a}^{[l-1]}\left(\delta^{[l]}\right)^{\top} \qquad \delta^{[l]} = \left[g'^{[l]}\left(f^{[l]}\right)\right] \circ \left(\boldsymbol{W}^{[l+1]}\delta^{[l+1]}\right)$$

Adapted from: https://en.wikipedia.org/wiki/Backpropagation

Auto Differentiation for Backprop

- Even with backprop, implementing the gradients is tedious
- Deep learning APIs have automated differentiation.
  - Tensor Flow autodiff
  - PyTorch autograd
  - Implement derivatives of many common functions
  - You just need to implement your layers and neurons; API will handle gradients
- Caution
  - If you want to implement **custom functions/layers** (not simple weighted sum)
  - They need to be **differentiable** to be able to calculate their **gradients**
  - Otherwise, backprop cannot update weights accurately


?
Questions!