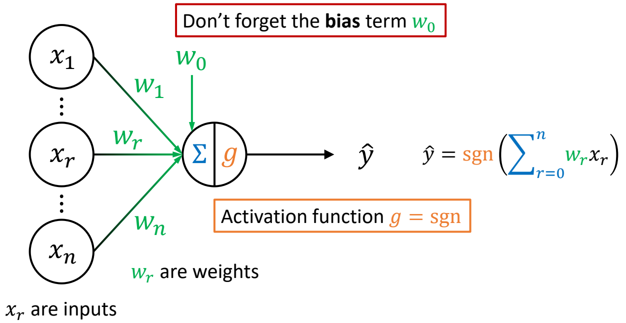
Backprop

9
B

CS 3244
Machine Learning

NUS | Computing
National University of Singapore

Perceptron



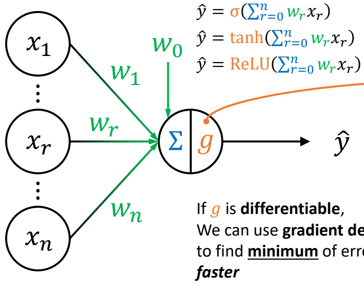
Perceptron Learning Algorithm

1. Initialize weights $\boldsymbol{w}$
   - Could be all zero, or random small values
2. For each instance i with features $\boldsymbol{x}^{(i)}$
   - Classify $\hat{y}^{(i)} = \mathrm{sgn}\left(\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)$
3. Select one misclassified instance
   - Update weights: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta(y - \hat{y})\boldsymbol{x}$
4. Iterate steps 2 to 3 until
   - Convergence (classification error < threshold), or
   - Maximum number of iterations

$$\begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_r \\ \vdots \\ w_n \end{pmatrix} \leftarrow \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_r \\ \vdots \\ w_n \end{pmatrix} + \eta(y - \hat{y}) \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_r \\ \vdots \\ x_n \end{pmatrix}$$

$$w_r \leftarrow w_r + \eta(y - \hat{y})x_r$$

Differentiable Activation Functions



$\hat{y} = \mathrm{sgn}\left(\sum_{r=0}^{n} w_r x_r\right)$
$\hat{y} = \sigma\left(\sum_{r=0}^{n} w_r x_r\right)$
$\hat{y} = \tanh\left(\sum_{r=0}^{n} w_r x_r\right)$
$\hat{y} = \mathrm{ReLU}\left(\sum_{r=0}^{n} w_r x_r\right)$

Step
$\mathrm{sgn}(x) = \begin{cases} +1 & z > 0 \\ -1 & z \le 0 \end{cases}$

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

tanh
$\tanh(x)$

ReLU
$\max(0, x)$

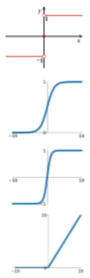
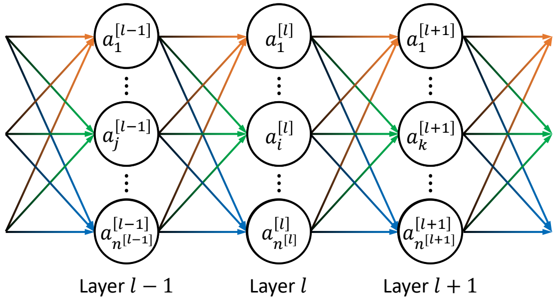


Image credit:
https://miro.medium.com/max/1400/0*sIJ-gbjlz0zrz8lb.png

If g is **differentiable**,
We can use **gradient descent**
to find **minimum** of error $\varepsilon = y - \hat{y}$
faster

Multi-Layer Perceptron (Neural Network)



Chain Rule

Consider composite function

$$g(x) = g\big(f(x)\big)$$

$$g = g(f), f = f(x)$$

$$g'(x) = \frac{dg}{dx} = \frac{dg}{df}\frac{df}{dx}$$

Intuition
Rate of change of g relative to x is the product of
- rates of change of g relative to f and
- rates of change of f relative to x

"If
- *a car travels 2x fast as a bicycle and*
- *the bicycle is 4x as fast as a walking man,*

then the car travels 2 × 4 = 8 times as fast as the man."
– George F. Simmons, Calculus with Analytic Geometry (1985)

Week 09B: Lecture Outline

1. Perceptron
2. Perceptron Learning Algorithm (PLA)
3. Activation Functions
4. Gradient Descent
5. Neural Networks

- Math Notation Primer

6. Backpropagation

Math Primer

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

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

Functions with Vectors and Matrices

- Scalar-by-scalar:
  - $y(x) = wx$ for **scaling** input
- Scalar-by-vector:
  - $y(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w}^\top \boldsymbol{x} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = w_1 x_1 + w_2 x_2$ 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)
  - $\boldsymbol{\sigma}(\boldsymbol{z}) = \frac{e^{\boldsymbol{z}}}{\boldsymbol{1}^\top(\boldsymbol{1}+e^{\boldsymbol{z}})}$ for **softmax**

Functions with Vectors and Matrices

- Matrix-by-matrix:
  - Using **Hadamard** product $\circ$ for element-wise multiplication
    - $\boldsymbol{y}(\boldsymbol{X}) = \boldsymbol{W} \circ \boldsymbol{X} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} = \begin{pmatrix} w_{11}x_{11} & w_{12}x_{12} \\ w_{21}x_{21} & w_{22}x_{22} \end{pmatrix}$
  - Using **Convolution** operator $*$ for element-wise multiplication then sum [W08b]
    - $\boldsymbol{Y}(\boldsymbol{X}) = \boldsymbol{W} * \boldsymbol{X} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix}$

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

    - For computer vision filters (kernels)

1D Vectors, and 2D Matrices and ≥2D Tensors are for convenient notation of multiple similar calculations.

Weighted Sum

Summation Series = Scalar

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

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

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 $\boldsymbol{x} = (x_1, \ldots, x_n)^\top$

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

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

  - Assumes Cartesian coordinates (linear, orthogonal)

Matrix Calculus

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

Scalar-by-Vector (1D Vector)

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

Scalar-by-Matrix (2D Matrix)

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

Vector-by-Vector (2D Matrix) – not in exam

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

Vector-by-Matrix (3D Tensor) – not in exam

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

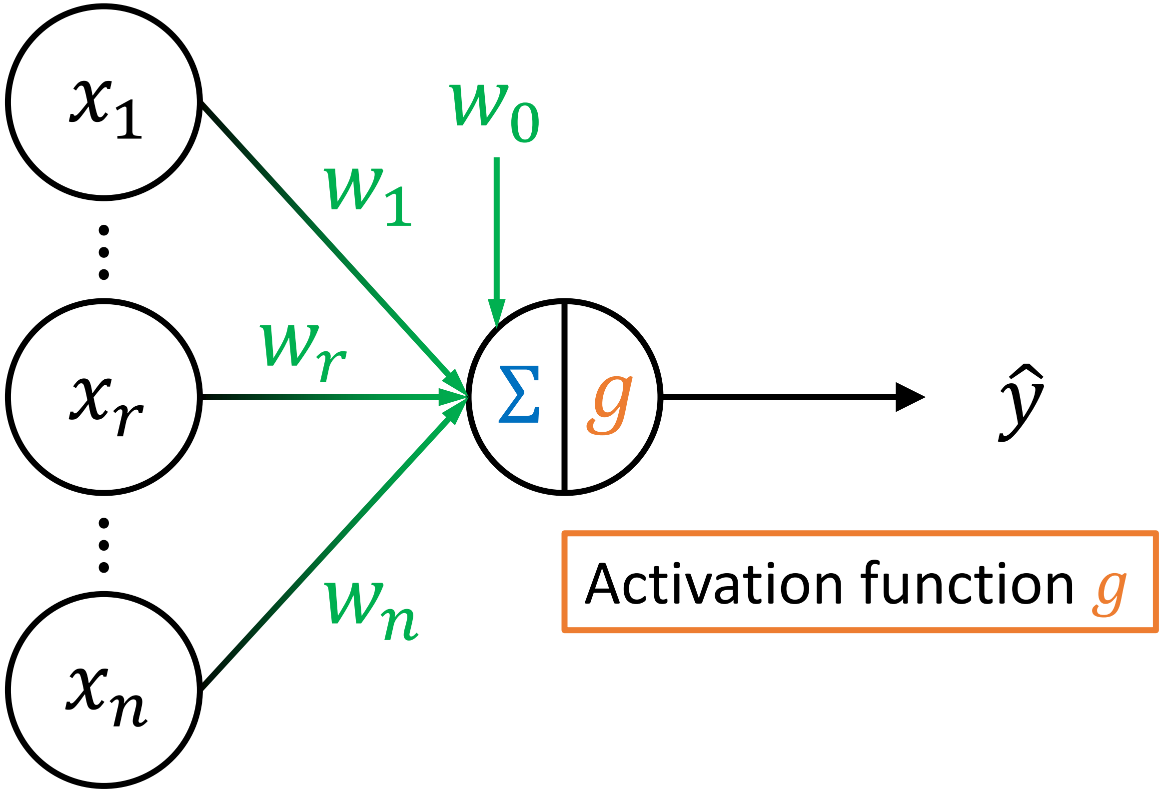
Along 3rd dimension

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

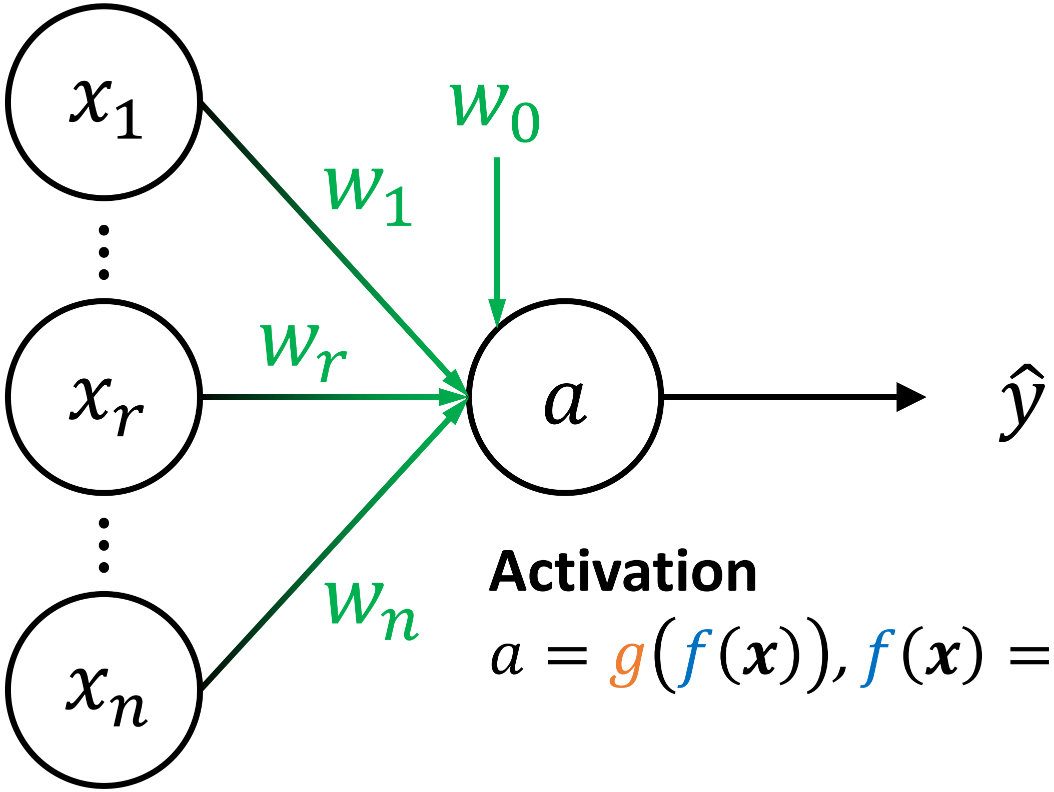
Neural Network
(recap)

NUS National University of Singapore | Department of Computer Science School of Computing

Single-Layer Perceptron



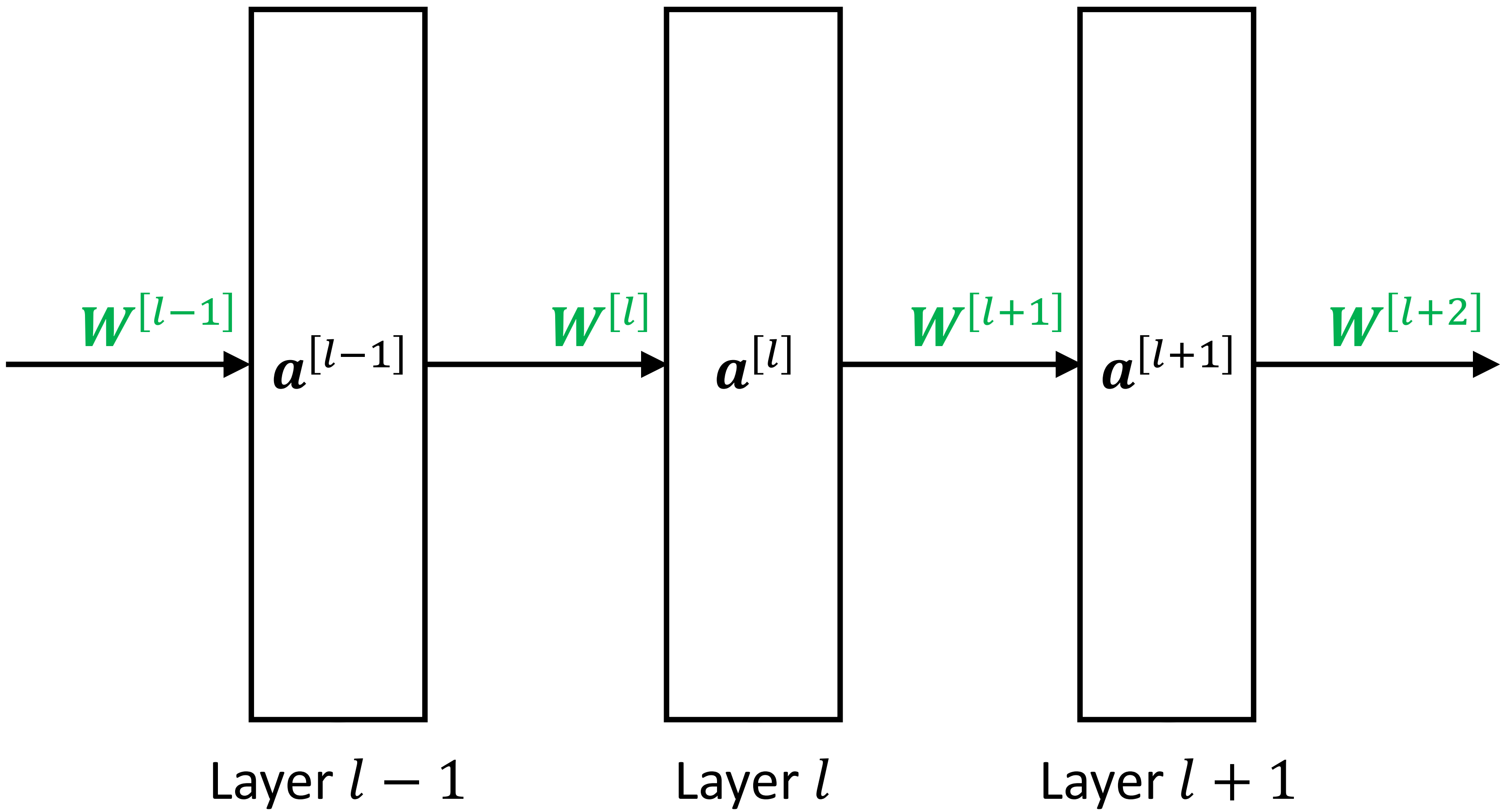
Single-Layer Perceptron



Activation

$a = g\big(f(\boldsymbol{x})\big), f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$

Neural Network



Layer Activation

$$a = g(f(x)), f(x) = \mathbf{w}^{\top} x$$

Single-Layer Perceptron

Layer l Activation Function

Layer l Weights

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

Layer l Activations

Layer $l-1$ Activations

Layer l in Neural Network

Gradient Descent Weight Update (Single Neuron)

Direction of fastest error increase

Old weight

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \varepsilon$$

New weight

Learning Rate

Gradient of error

$$\nabla \varepsilon = \frac{d\varepsilon}{d\mathbf{w}} = \begin{pmatrix} \partial\varepsilon/\partial w_1 \\ \vdots \\ \partial\varepsilon/\partial w_r \\ \vdots \\ \partial\varepsilon/\partial w_n \end{pmatrix}$$

Gradient Descent Weight Update (Neural Network)

$$\underset{\text{New weight}}{\boldsymbol{W}} \leftarrow \overset{\text{Old weight}}{\boldsymbol{W}} - \underset{\text{Learning Rate}}{\eta} \overset{\text{Direction of fastest error increase}}{\nabla \varepsilon}$$

Gradient of error

$$\nabla \varepsilon = \frac{d\varepsilon}{d\boldsymbol{W}} = \begin{pmatrix} \partial\varepsilon / \partial \boldsymbol{W}^{[1]} \\ \vdots \\ \partial\varepsilon / \partial \boldsymbol{W}^{[l]} \\ \vdots \\ \partial\varepsilon / \partial \boldsymbol{W}^{[L]} \end{pmatrix}$$

$$\frac{d\varepsilon}{d\boldsymbol{W}^{[l]}} = \frac{d\varepsilon}{d\hat{y}} \frac{d\hat{y}}{d\boldsymbol{W}^{[l]}}$$

$$\frac{d\varepsilon}{d\boldsymbol{W}^{[l]}} = \begin{pmatrix} \frac{\partial\varepsilon}{\partial w_{11}^{[l]}} & \cdots & \frac{\partial\varepsilon}{\partial w_{1m}^{[l]}} \\ \vdots & \ddots & \vdots \\ \frac{\partial\varepsilon}{\partial w_{n1}^{[l]}} & \cdots & \frac{\partial\varepsilon}{\partial w_{nm}^{[l]}} \end{pmatrix}$$

Gradient Descent for Neural Networks

Backpropagation

Department of Computer Science
School of Computing

Backpropagation

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

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

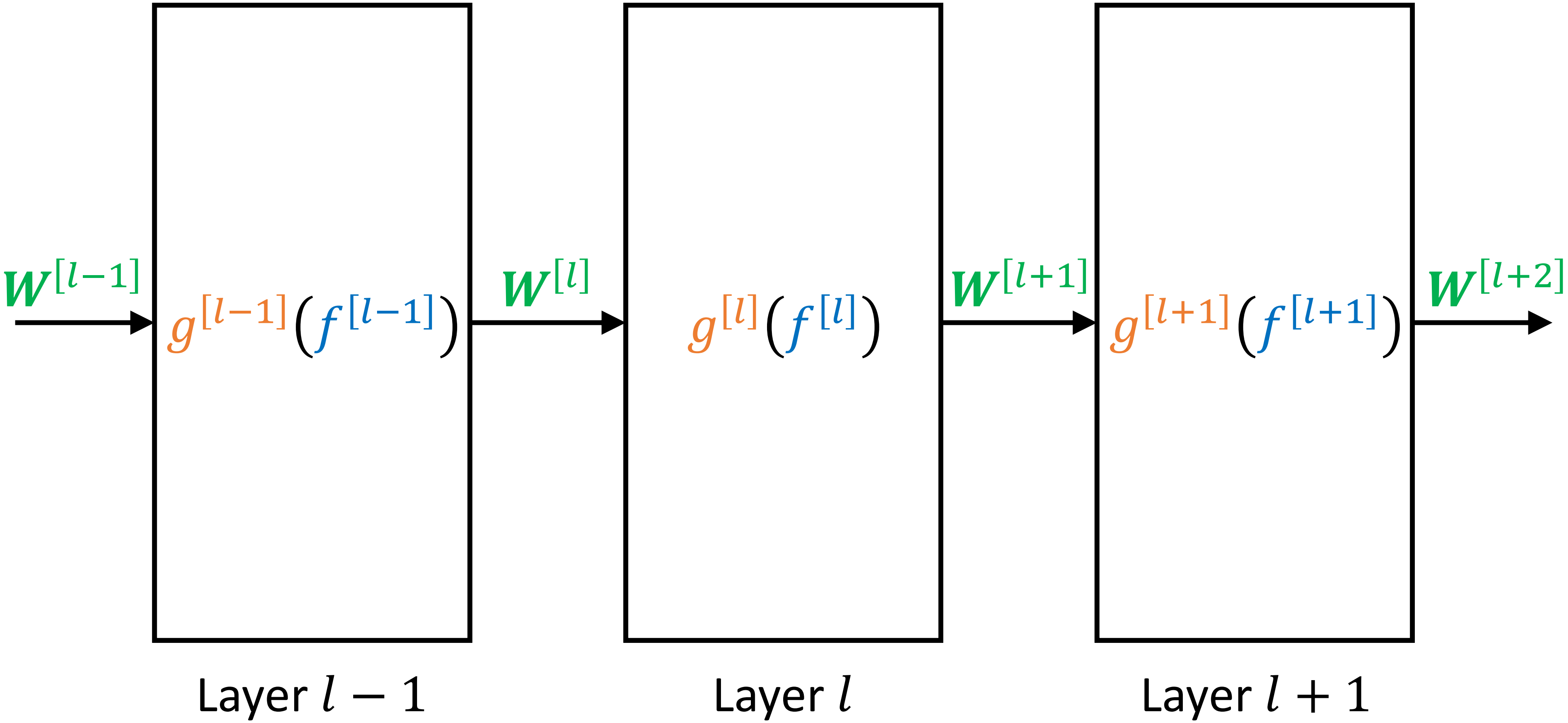
Specifically, the gradient of the weighted input of each layer

is calculated from back $[l+1]$ to front $[l]$:

$$\frac{d\hat{y}}{d\boldsymbol{W}^{[l]}} = a^{[l-1]}\left(\boldsymbol{\delta}^{[l]}\right)^{\mathsf{T}} \qquad \boldsymbol{\delta}^{[l]} = \left(\frac{dg^{[l]}}{df^{[l]}}\right)\left(\boldsymbol{W}^{[l+1]}\boldsymbol{\delta}^{[l+1]}\right)$$

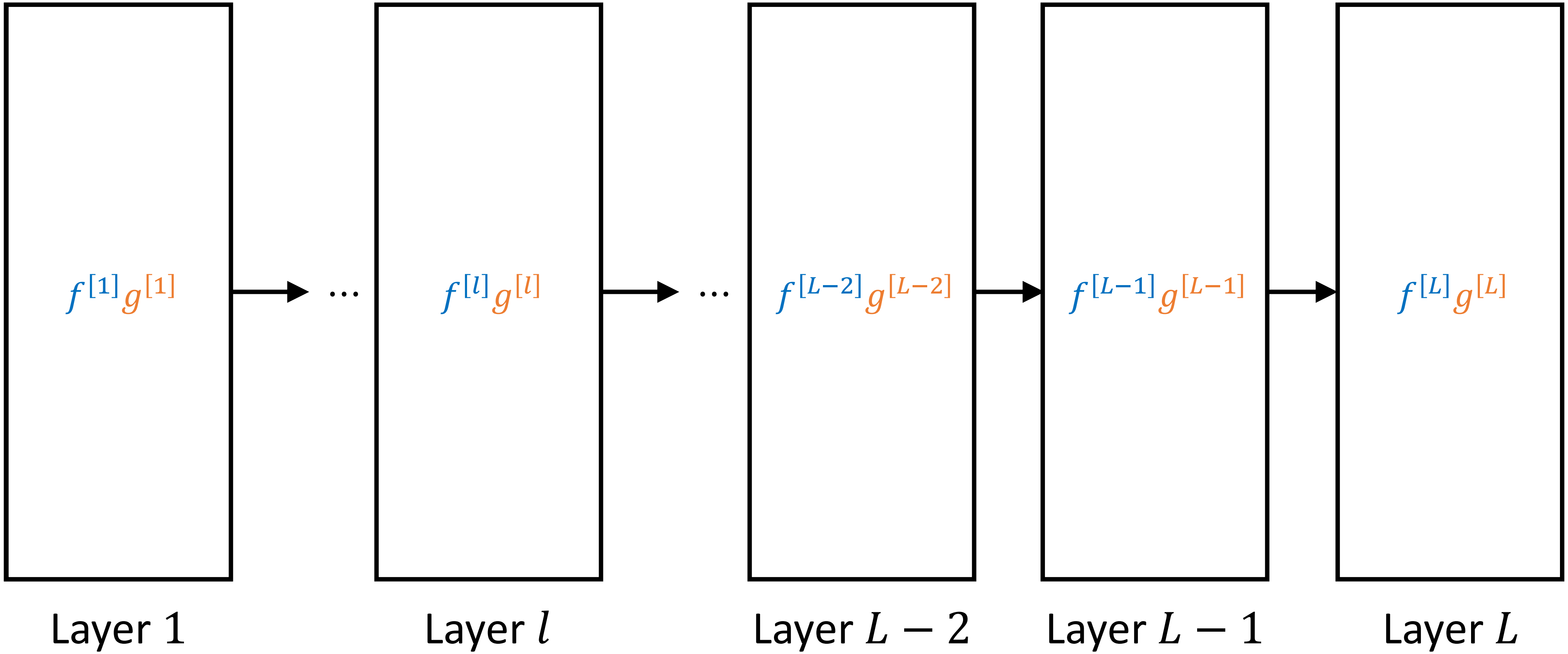
Adapted from: https://en.wikipedia.org/wiki/Backpropagation

Forward Propagation



Forward Propagation (Reverse Polish Notation)

$$(\boldsymbol{x}^{[0]}, \boldsymbol{W}^{[1]}) f^{[1]} g^{[1]} \dots f^{[l]} g^{[l]} \dots f^{[L-2]} g^{[L-2]} f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$



Forward Propagation (Reverse Polish Notation)

$$\left(x^{[0]}, \boldsymbol{W}^{[1]}\right) f^{[1]} g^{[1]} \ldots f^{[l]} g^{[l]} \ldots f^{[L-2]} g^{[L-2]} f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\left(a^{[L-1]}, \boldsymbol{W}^{[L]}\right) f^{[L]} g^{[L]} = \hat{y}$$

$$\left(a^{[L-2]}, \boldsymbol{W}^{[L-1]}\right) f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\left(a^{[L-3]}, \boldsymbol{W}^{[L-2]}\right) f^{[L-2]} g^{[L-2]} f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

Gradients of Layer Weights (Backwards)

$$\left(x^{[0]}, \boldsymbol{W}^{[1]}\right) f^{[1]} g^{[1]} \dots f^{[l]} g^{[l]} \dots f^{[L-2]} g^{[L-2]} f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\left(a^{[L-1]}, \boldsymbol{W}^{[L]}\right) f^{[L]} g^{[L]} = \hat{y}$$

$$\frac{df^{[L]}}{d\boldsymbol{W}^{[L]}} \frac{dg^{[L]}}{df^{[L]}} = \frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[L]}}$$

$$\left(a^{[L-2]}, \boldsymbol{W}^{[L-1]}\right) f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\frac{df^{[L-1]}}{d\boldsymbol{W}^{[L-1]}} \frac{dg^{[L-1]}}{df^{[L-1]}} \frac{df^{[L]}}{dg^{[L-1]}} \frac{dg^{[L]}}{df^{[L]}} = \frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[L-1]}}$$

$$\left(a^{[L-3]}, \boldsymbol{W}^{[L-2]}\right) f^{[L-2]} g^{[L-2]} f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\frac{df^{[L-2]}}{d\boldsymbol{W}^{[L-2]}} \frac{dg^{[L-2]}}{df^{[L-2]}} \frac{df^{[L-1]}}{dg^{[L-2]}} \frac{dg^{[L-1]}}{df^{[L-1]}} \frac{df^{[L]}}{dg^{[L-1]}} \frac{dg^{[L]}}{df^{[L]}} = \frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[L-2]}}$$

Gradients of Layer Weights (Backwards)

$$\left(x^{[0]}, \boldsymbol{W}^{[1]}\right) f^{[1]} g^{[1]} \dots f^{[l]} g^{[l]} \dots f^{[L-2]} g^{[L-2]} f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\left(a^{[L-1]}, \boldsymbol{W}^{[L]}\right) f^{[L]} g^{[L]} = \hat{y}$$

$$\frac{df^{[L]}}{d\boldsymbol{W}^{[L]}} \boxed{\boldsymbol{\delta}^{[L]}} = \frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[L]}}$$

$$\left(a^{[L-2]}, \boldsymbol{W}^{[L-1]}\right) f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\frac{df^{[L-1]}}{d\boldsymbol{W}^{[L-1]}} \boxed{\boldsymbol{\delta}^{[L-1]}} = \frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[L-1]}}$$

$$\left(a^{[L-3]}, \boldsymbol{W}^{[L-2]}\right) f^{[L-2]} g^{[L-2]} f^{[L-1]} g^{[L-1]} f^{[L]} g^{[L]} = \hat{y}$$

$$\frac{df^{[L-2]}}{d\boldsymbol{W}^{[L-2]}} \boxed{\boldsymbol{\delta}^{[L-2]}} = \frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[L-2]}}$$

Recursive Gradients of Layer Weights

$$\frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[l]}} = \frac{df^{[l]}}{d\boldsymbol{W}^{[l]}} \frac{dg^{[l]}}{df^{[l]}} \frac{df^{[l+1]}}{dg^{[l]}} \underbrace{\frac{dg^{[l+1]}}{df^{[l+1]}} \cdots \frac{df^{[L]}}{dg^{[L-1]}} \frac{dg^{[L]}}{df^{[L]}}}_{\delta^{[l+1]}}$$

$$\frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[l]}} = \frac{df^{[l]}}{d\boldsymbol{W}^{[l]}} \underbrace{\frac{dg^{[l]}}{df^{[l]}} \frac{df^{[l+1]}}{dg^{[l]}} \delta^{[l+1]}}_{\delta^{[l]}}$$

$$\delta^{[l]} = \frac{dg^{[l]}}{df^{[l]}} \frac{df^{[l+1]}}{dg^{[l]}} \delta^{[l+1]}$$

$$\frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[l]}} = \frac{df^{[l]}}{d\boldsymbol{W}^{[l]}} \delta^{[l]}$$

Reference

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

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

$$\boldsymbol{a}^{[l-1]} = g^{[l-1]}$$

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

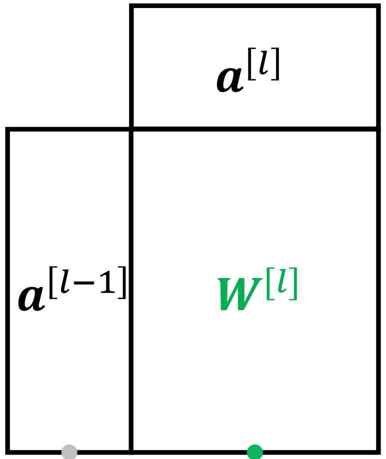
$$\frac{df^{[l+1]}}{dg^{[l]}} = \frac{df^{[l+1]}}{d\boldsymbol{a}^{[l]}} = \boldsymbol{W}^{[l+1]}$$

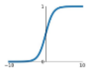
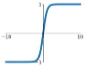
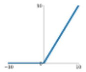
$$\boxed{\frac{\partial \hat{y}}{\partial \boldsymbol{W}^{[l]}} = \boldsymbol{a}^{[l-1]} \delta^{[l]}} \quad \boxed{\delta^{[l]} = \frac{dg^{[l]}}{df^{[l]}} \boldsymbol{W}^{[l+1]} \delta^{[l+1]}}$$

Recursive

Matrix multiplication to match shape (not in exam)

$$\frac{d\hat{y}}{d\boldsymbol{W}^{[l]}} = \begin{pmatrix} \frac{\partial \hat{y}}{\partial w_{11}^{[l]}} & \cdots & \frac{\partial \varepsilon}{\partial w_{1n^{[l]}}^{[l]}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \varepsilon}{\partial w_{n^{[l-1]}1}^{[l]}} & \cdots & \frac{\partial \varepsilon}{\partial w_{n^{[l-1]}n^{[l]}}^{[l]}} \end{pmatrix}$$

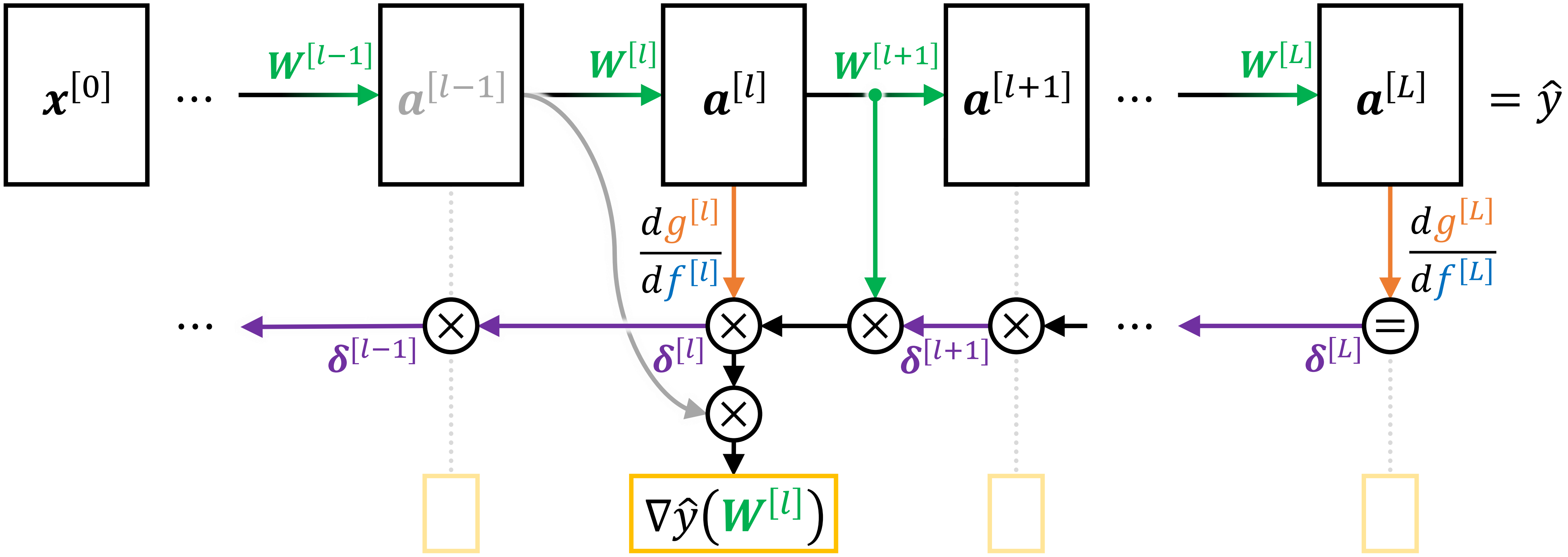


Activation $\hat{y} = g(f)$			$\frac{dg}{df}$
Sigmoid		$\frac{1}{1+e^{-f}}$	$(1-g)g$
tanh		$\tanh f$	$1-g^2$
ReLU		$\max(0, f)$	$[f > 0]$

$n_{\text{rows}} \times n_{\text{cols}}$ $\quad n^{[l-1]} \times n^{[l]}$ $\quad n^{[l-1]} \times 1$ $\quad 1 \times n^{[l]}$ $\quad n^{[l]} \times 1$ $\quad 1 \times 1$ $\quad n^{[l]} \times n^{[l+1]}$ $\quad n^{[l+1]} \times 1$

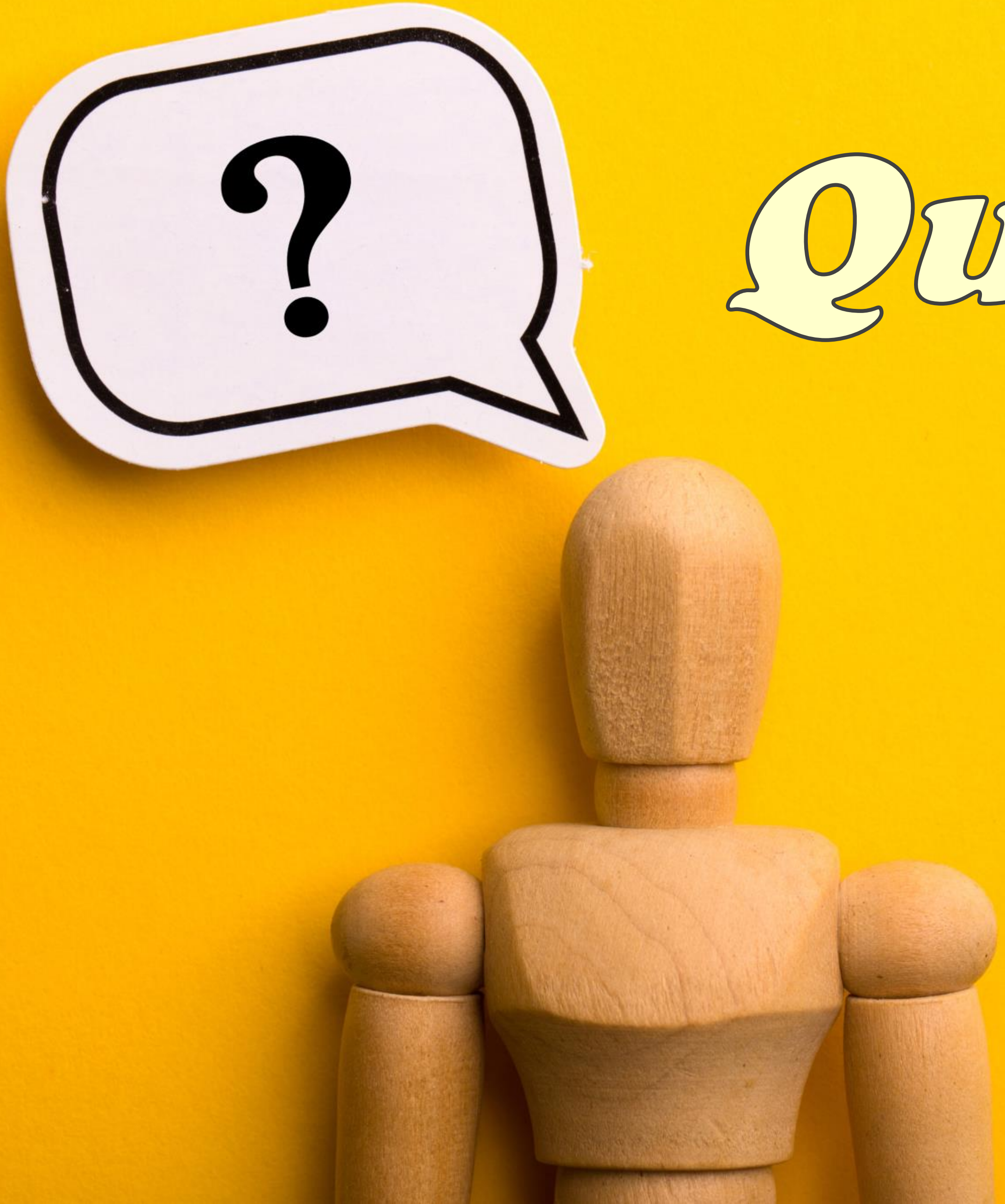
$$\frac{d\hat{y}}{d\boldsymbol{W}^{[l]}} = a^{[l-1]}\left(\boldsymbol{\delta}^{[l]}\right)^{\mathsf{T}} \qquad \boldsymbol{\delta}^{[l]} = \left(\frac{dg^{[l]}}{df^{[l]}}\right)\left(\boldsymbol{W}^{[l+1]}\boldsymbol{\delta}^{[l+1]}\right)$$

Backward Propagation



$\frac{d\hat{y}}{dW^{[l]}} = a^{[l-1]}\left(\delta^{[l]}\right)^{\top}$	$\delta^{[l]} = \left(\frac{dg^{[l]}}{df^{[l]}}\right)\left(W^{[l+1]}\delta^{[l+1]}\right)$

Questions!



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 $[l]$:

$$\boxed{\frac{d\hat{y}}{d\boldsymbol{W}^{[l]}} = a^{[l-1]}\left(\boldsymbol{\delta}^{[l]}\right)^{\mathsf{T}}} \quad \boxed{\boldsymbol{\delta}^{[l]} = \left(\frac{dg^{[l]}}{df^{[l]}}\right)\left(\boldsymbol{W}^{[l+1]}\boldsymbol{\delta}^{[l+1]}\right)}$$

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

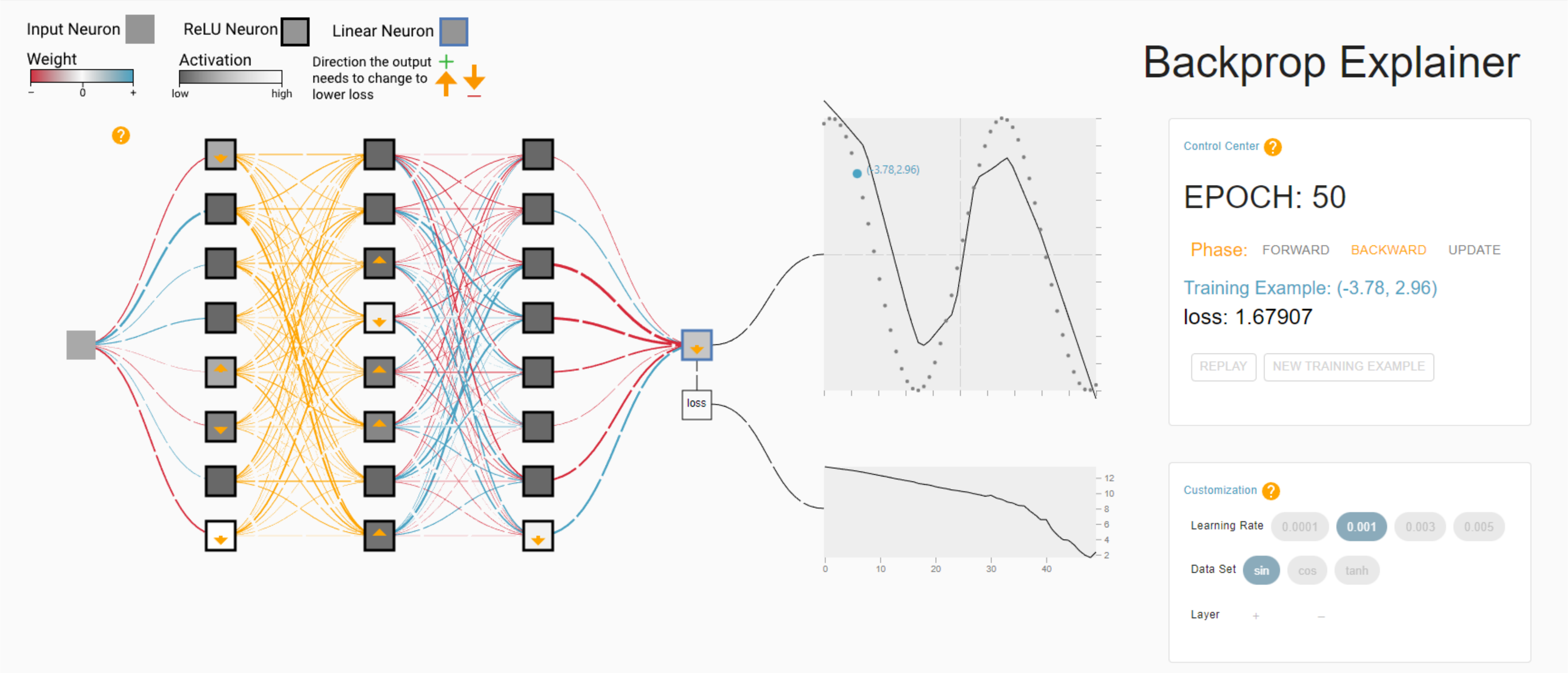
Online Demo

Backprop Explainer Quick Start

https://xnought.github.io/backprop-explainer/

1. Press ▶ to start training
2. Then press CLICK TO ANIMATE EPOCH # to see forward propagation, **backward propagation**, and update animation at the epoch #
3. To go back to fitting mode click ‹ GO BACK TO FITTING

Click on ? to reveal extra descriptions



Online Demo

Insert Web Page

This app allows you to insert secure web pages starting with https:// into the slide deck. Non-secure web pages are not supported for security reasons.

Please enter the URL below.

https:// xnought.github.io/backprop-explainer/

Note: Many popular websites allow secure access. Please click on the preview button to ensure the web page is accessible.

Practice Backprop during tutorial

National University of Singapore
School of Computing
CS3244: Machine Learning
Solution to Tutorial 07

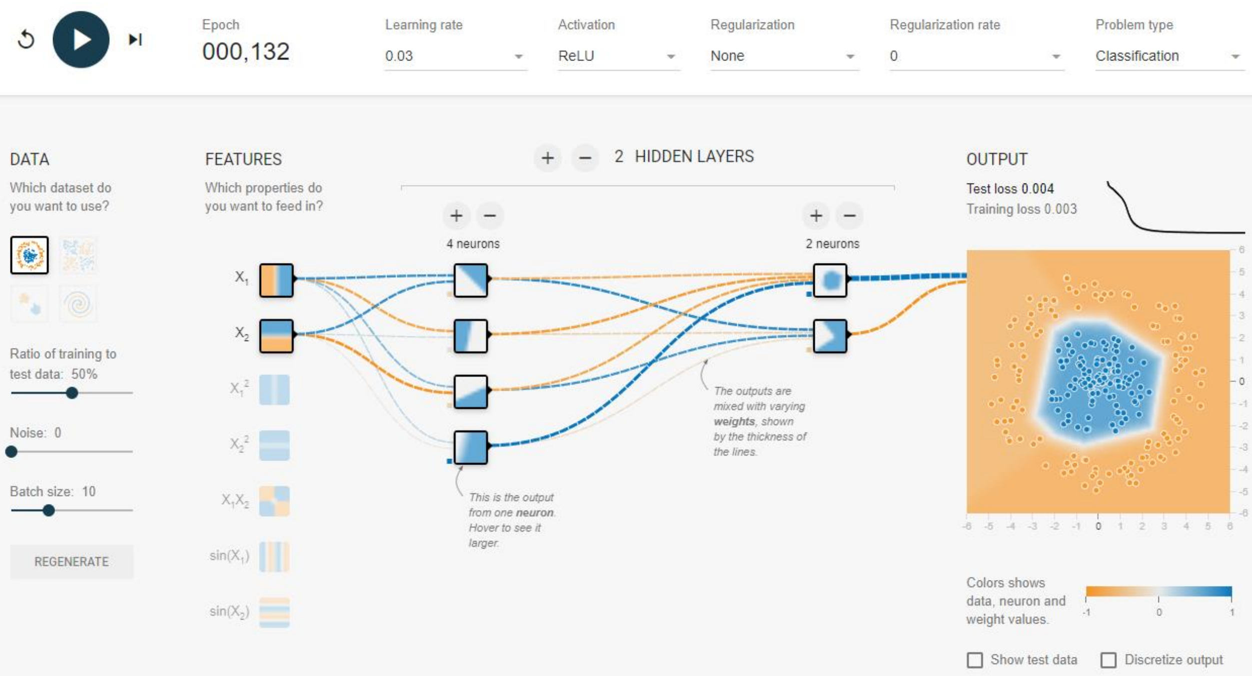
Perceptrons and Neural Networks

Colab Notebook : Perceptrons and Neural Networks

1. **Backpropagation algorithm.** In this question, we're going to use a neural network with **a**

Resources for self-study

- What is backpropagation really doing?, Backpropagation calculus - 3Blue1Brown
- A worked example of backpropagation - Alexander Schiendorfer
- TensorFlow Playground



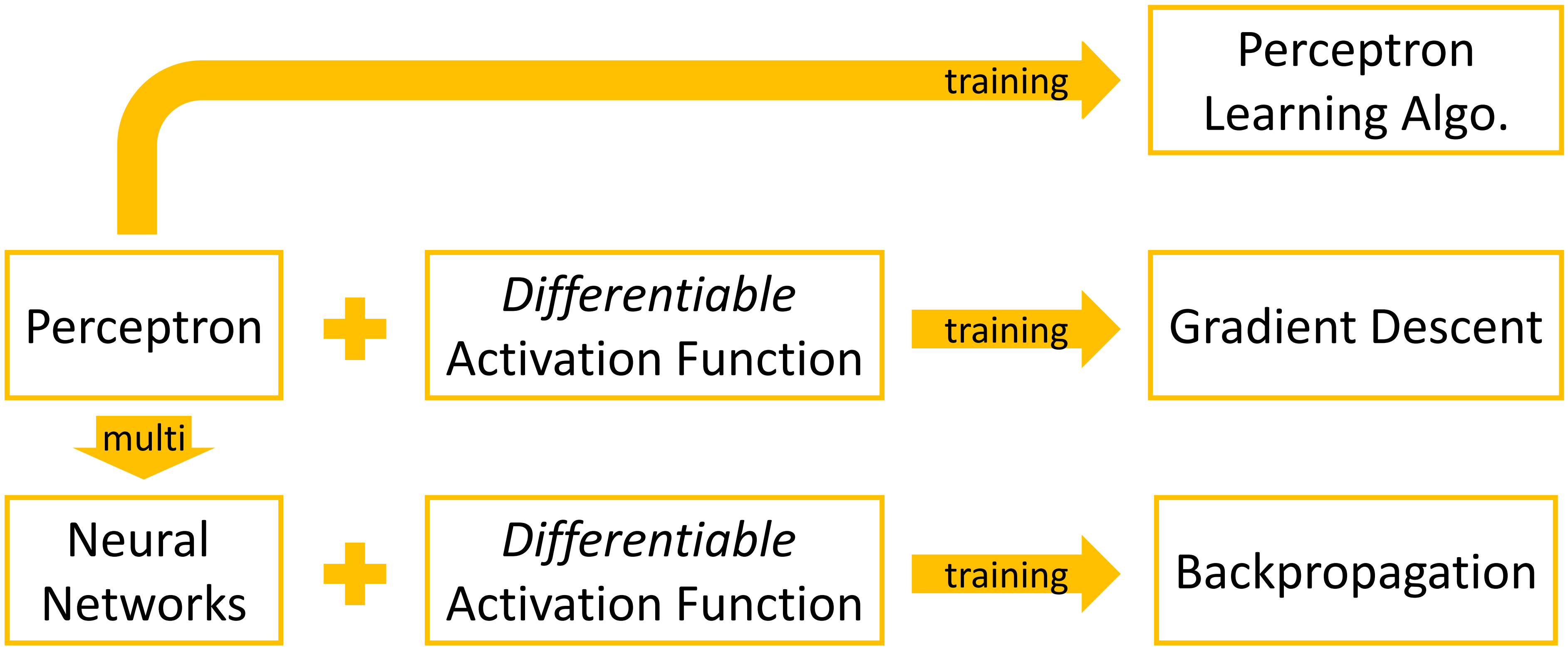
Auto Differentiation for Backprop

- 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

Wrapping Up

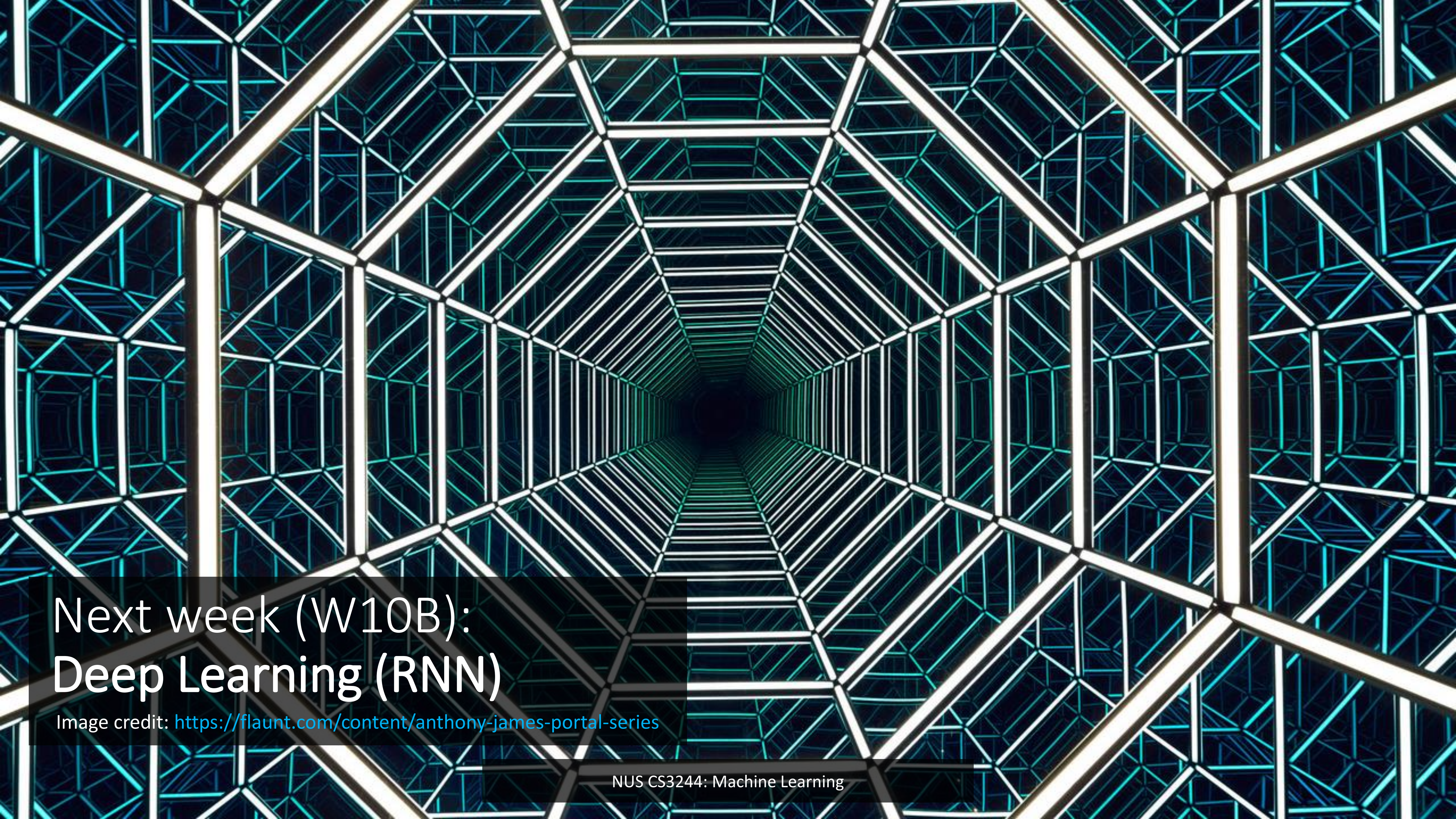

AI
HISTORY

What did we learn?



Next week (W10A):
Deep Learning (CNN)

Image credit: https://rigu.co.uk/kaleidoscope-filter-77mm



Next week (W10B):

Deep Learning (RNN)

Image credit: https://flaunt.com/content/anthony-james-portal-series

W10 Pre-Lecture Task (due before next Mon)

Watch

- Who Invented A.I.? - The Pioneers of Our Future by ColdFusion

Play

- https://distill.pub/2018/building-blocks/
  - Don't worry reading the whole article

Discuss

1. Identify what is strange, funny, or erroneous in the deep learning model in Building-Blocks
2. Take a screenshot of the issue and share with your tutorial mates
3. Try to explain why the model was behaving as identified
3. Post a 2–3 sentence description to the topic in your tutorial group: #tg-xx