CS2109S: Introduction to AI and Machine Learning

Lecture 9:
Backpropagation

27 October 2023

Announcement

- Midterm grading is still on progress, sorry :(

Recap

- Perceptron
  - Biological inspiration: brain, neural network, neuron
  - Perceptron Learning Algorithm:
    - $w \leftarrow w + \gamma\left(y^{(j)} - \hat{y}^{(j)}\right)x^{(j)}$ on a misclassified instance
- Neural Networks
  - Single-layer Neural Networks: AND, OR, NOR
  - Multi-layer Neural Networks: XNOR, Universal Approximation Theorem
  - Regression and Classification
- Neural Networks with Gradient Descent
- Neural Networks vs Other Models: learned feature mapping!

ERRATA

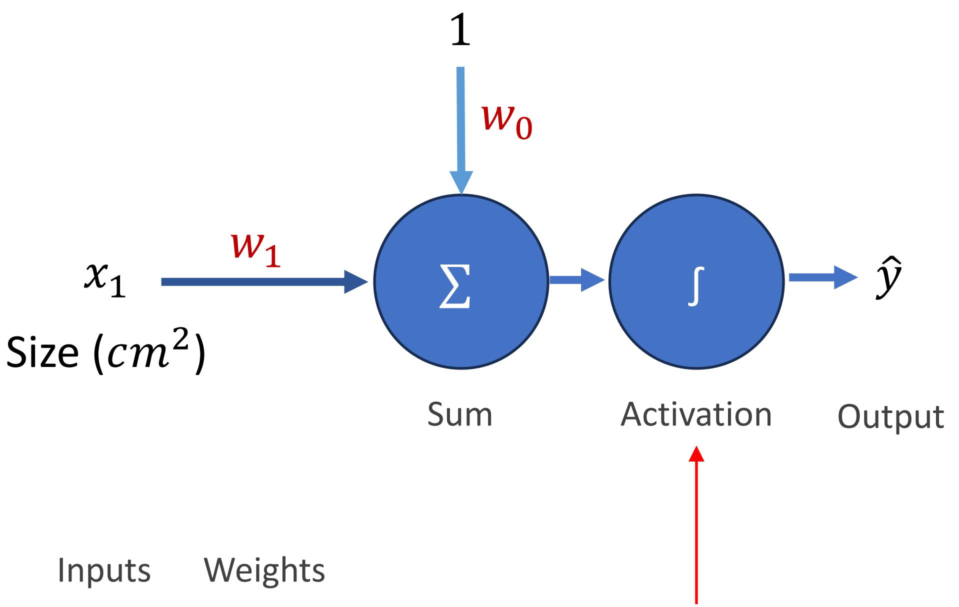
Frank Rossenblatt (1943)

Perceptron: An Example

$$\hat{y} = h_w(x) = g\left(\sum_{i=0}^{n} w_i x_i\right)$$

$$h_w(x) = \begin{cases} +1 & \text{if } w_0 + w_1 x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

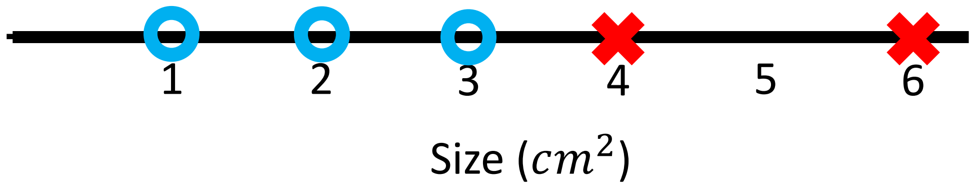
$$h_w(x) = \begin{cases} +1 & \text{if } -3.5 + 1x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$



Sign function

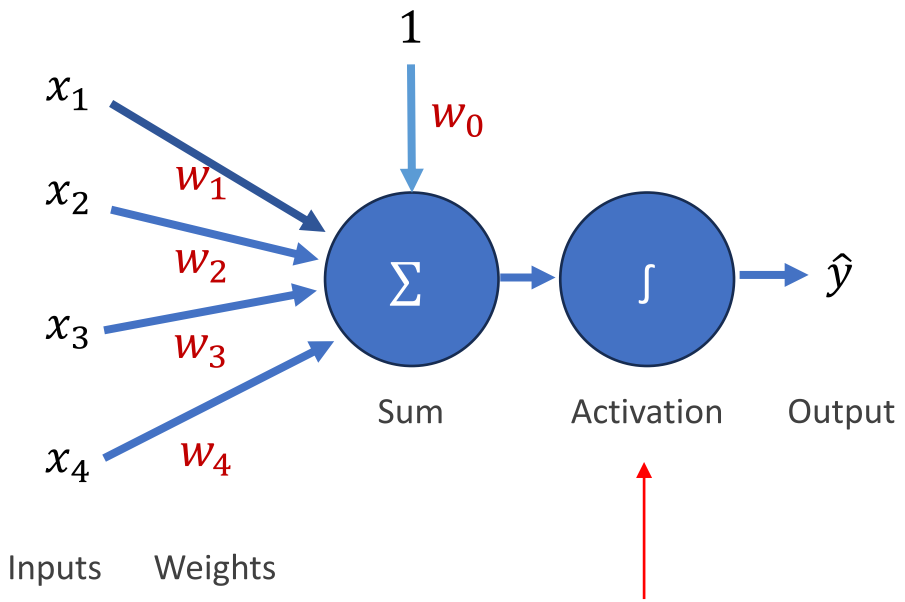
$$g(z) = \begin{cases} +1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

Cancer prediction: benign (-1), malignant (1)



ERRATA

Single-layer Neural Networks



$$\hat{y} = h_w(x) = g\left(\sum_{i=0}^{n} w_i x_i\right)$$

Activation Function (usually non-linear)

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

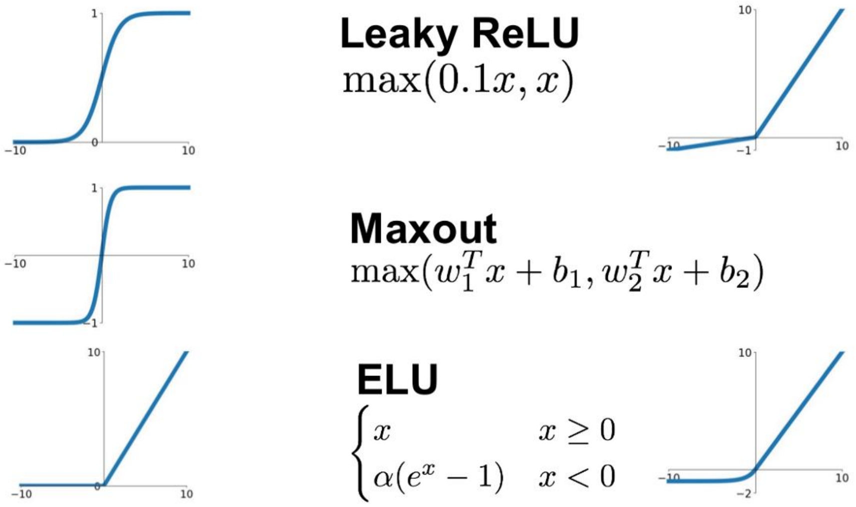
tanh
$\tanh(x)$

ReLU
$\max(0, x)$

Leaky ReLU
$\max(0.1x, x)$

Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



ERRATA

Background: Chain Rule

$$z(x) = h\left(g\left(f(x)\right)\right)$$

$$\frac{dz}{dx} = \frac{dz}{dh}\frac{dh}{dg}\frac{dg}{df}\frac{df}{dx}$$

ERRATA

Aside: Derivative of Sigmoid Function

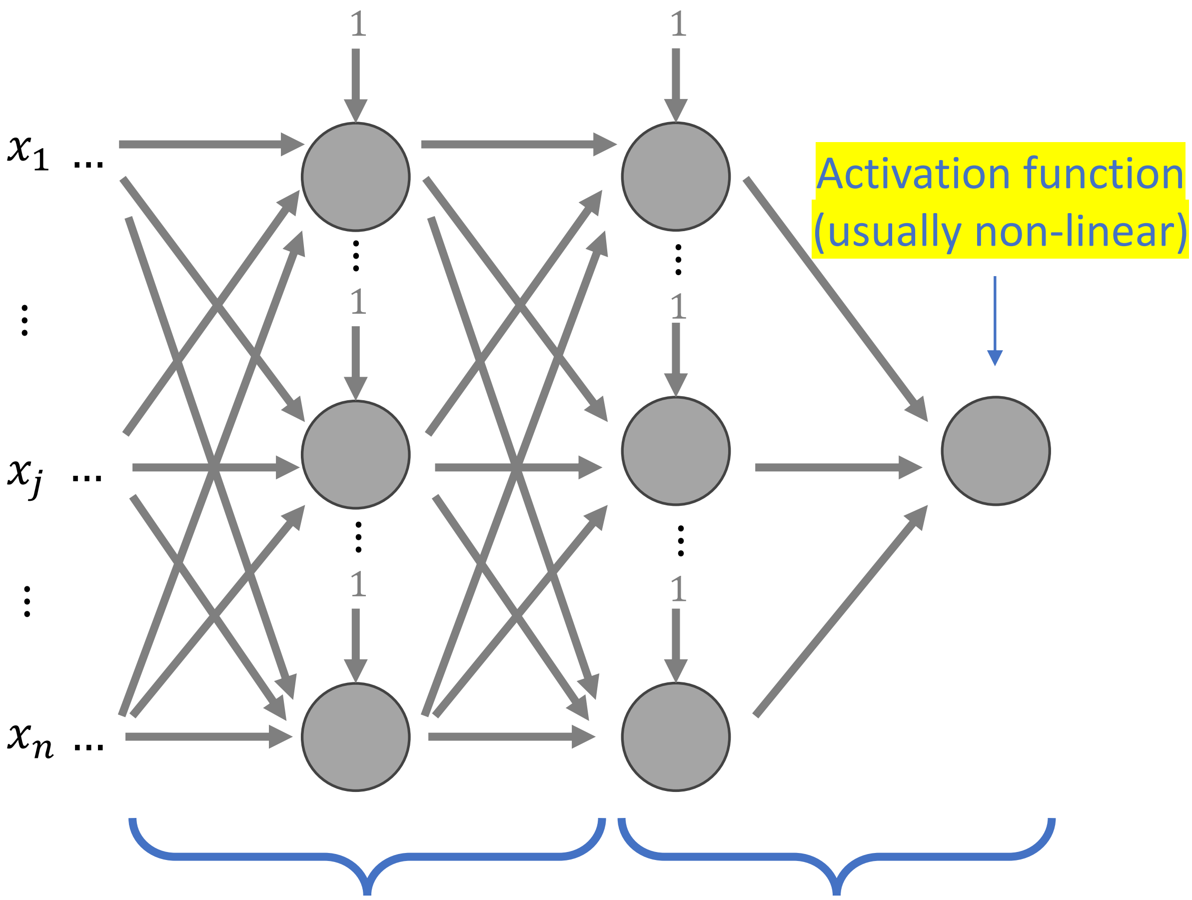
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\begin{aligned}
\sigma'(x) &= \frac{d\left(\frac{1}{1+e^{-x}}\right)}{dx} \\
&= \frac{d\left((1+e^{-x})^{-1}\right)}{dx} \\
&= -(1+e^{-x})^{-2}\,(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^{2}} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \boxed{\sigma(x)\big(1-\sigma(x)\big)}
\end{aligned}$$

ERRATA

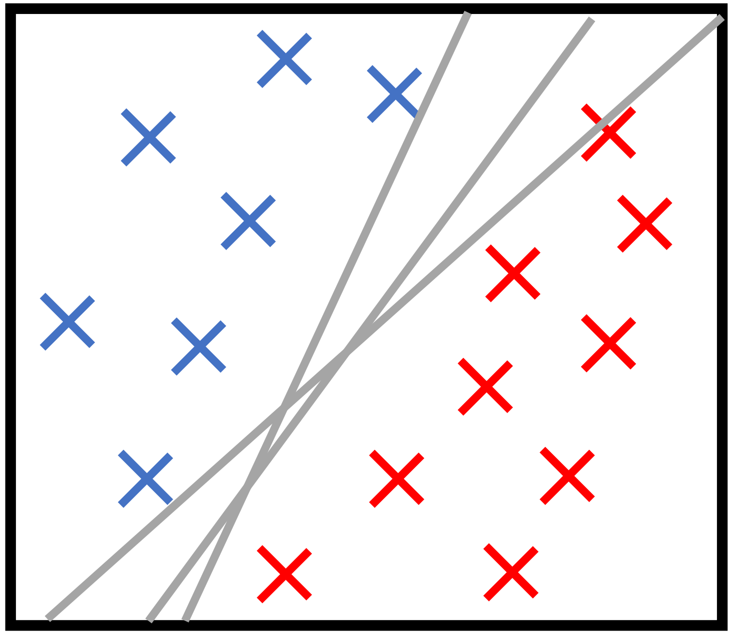
Neural Networks vs Other Methods

Credit: Internet meme, original source unknown



Learned Feature Mapping

Linear Model

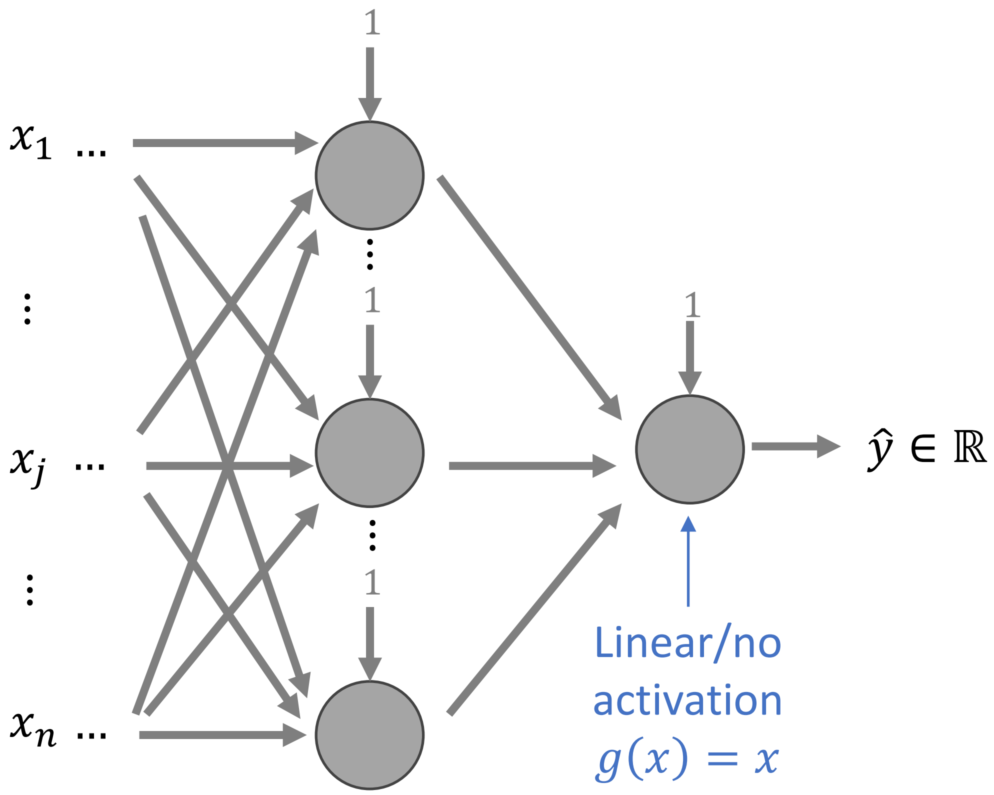


Non-linear, **non-robust** decision boundary

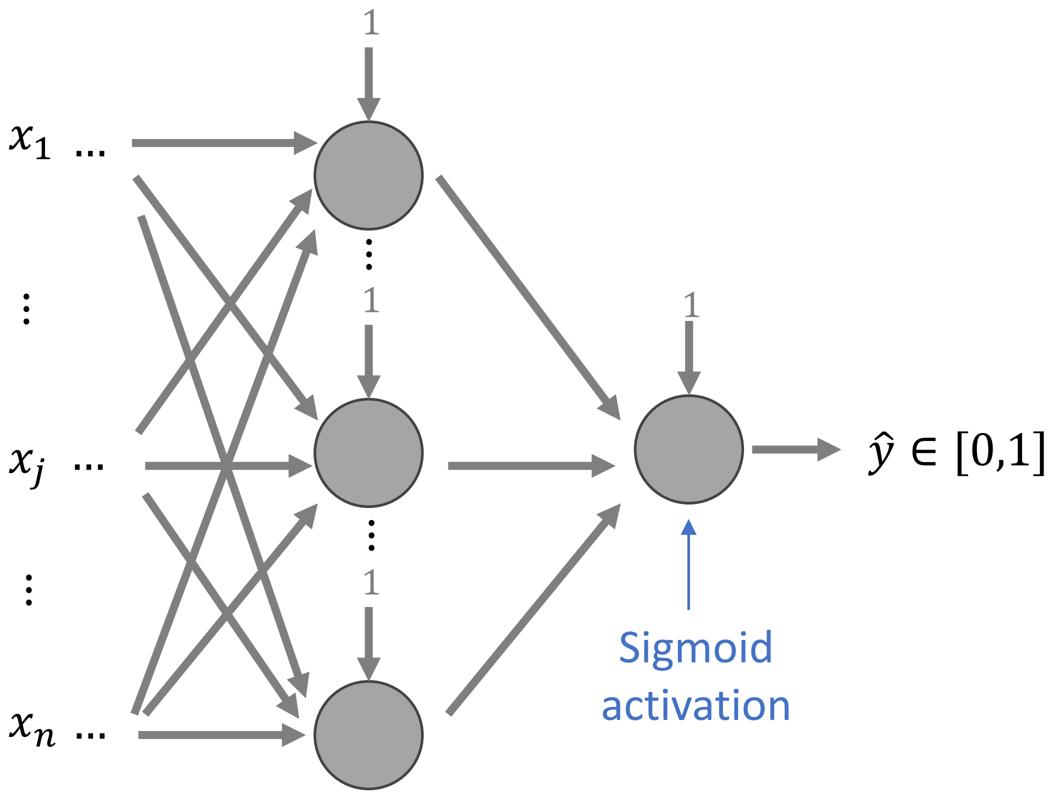
Prone to misclassification since the decision boundary can be **too close to data points**

ADDITION

Regression and Classification



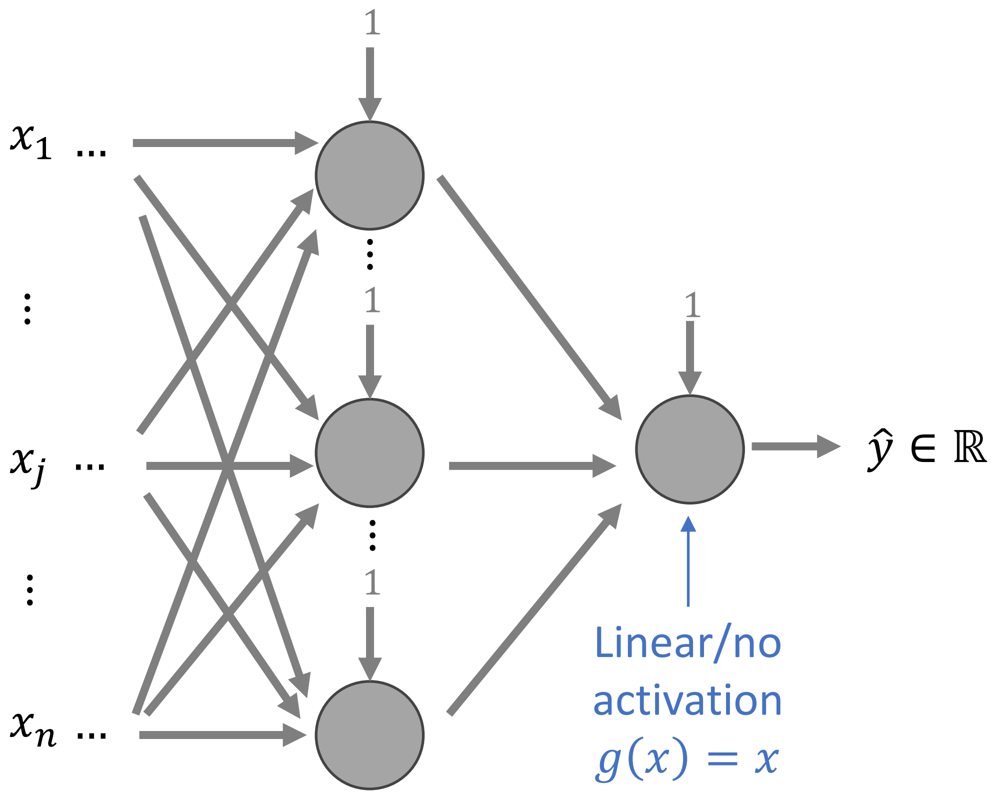
Regression



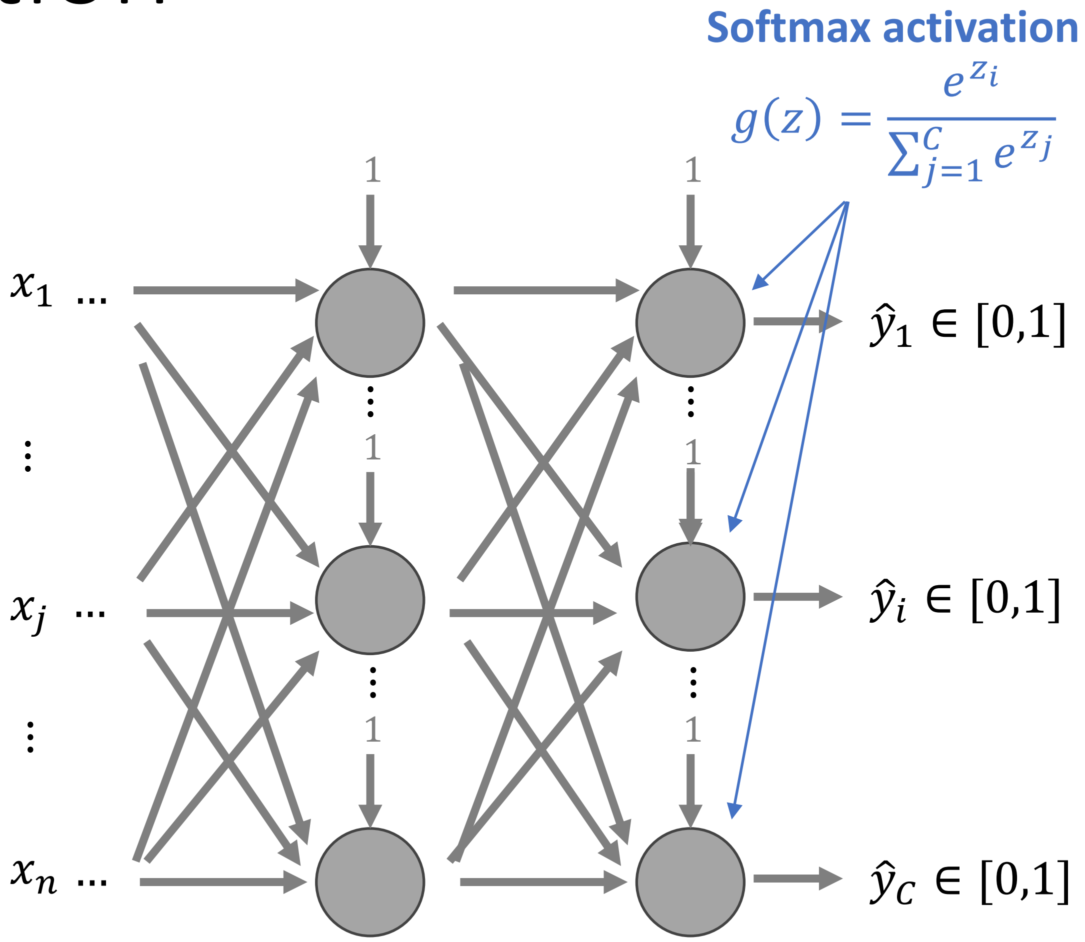
Binary Classification

ADDITION

Regression and Classification



Regression



Multi-class Classification

with C classes

$$\sum_{i=1}^{C} \hat{y}_i = 1$$

ADDITION

Multi-layer Neural Networks: $|x-1|$

Step

$\mathrm{sgn}(x) = \begin{cases} +1 & z \geq 0 \\ -1 & z < 0 \end{cases}$

Sigmoid

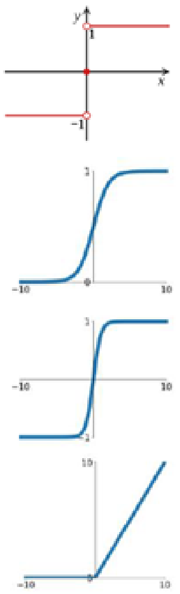
$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh

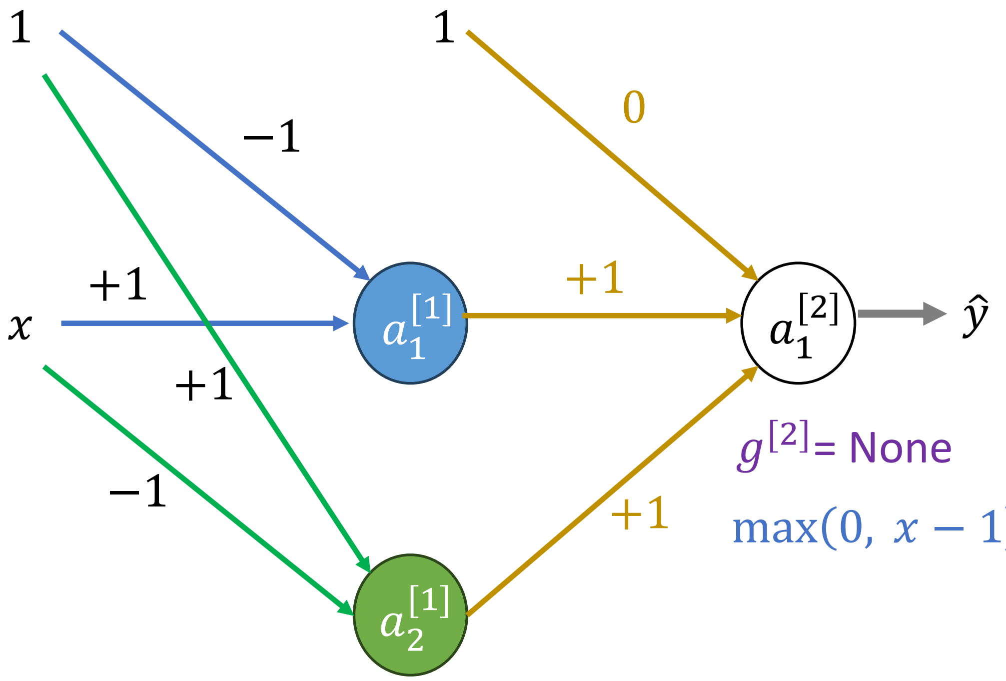
$\tanh(x)$

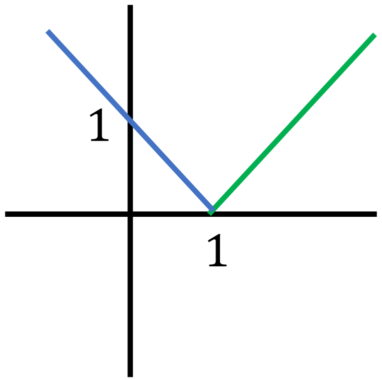
ReLU

$\max(0, x)$



Which activation function(s)?





$g^{[2]}$= None

$\max(0,\ x-1) + \max(0,\ 1-x)$

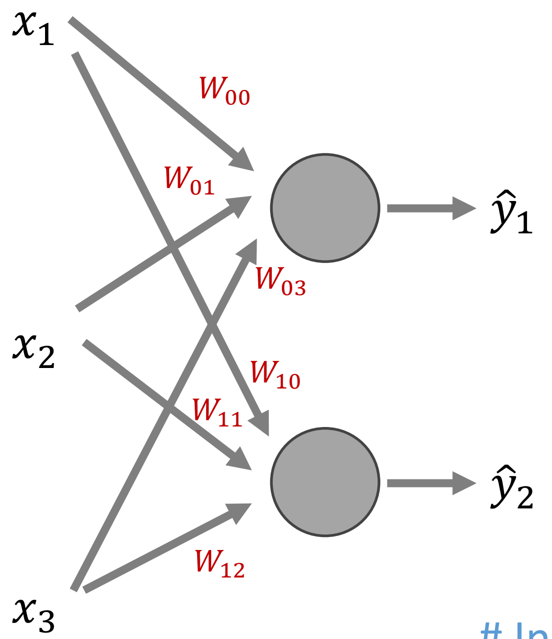
$g^{[1]}$= ReLU $\max(0, x)$

$\max(0,\ x-1)$

$\max(0,\ 1-x)$

ADDITION

Neural Networks and Matrix Multiplication (1)



Input (number of weights per neuron / input variables)

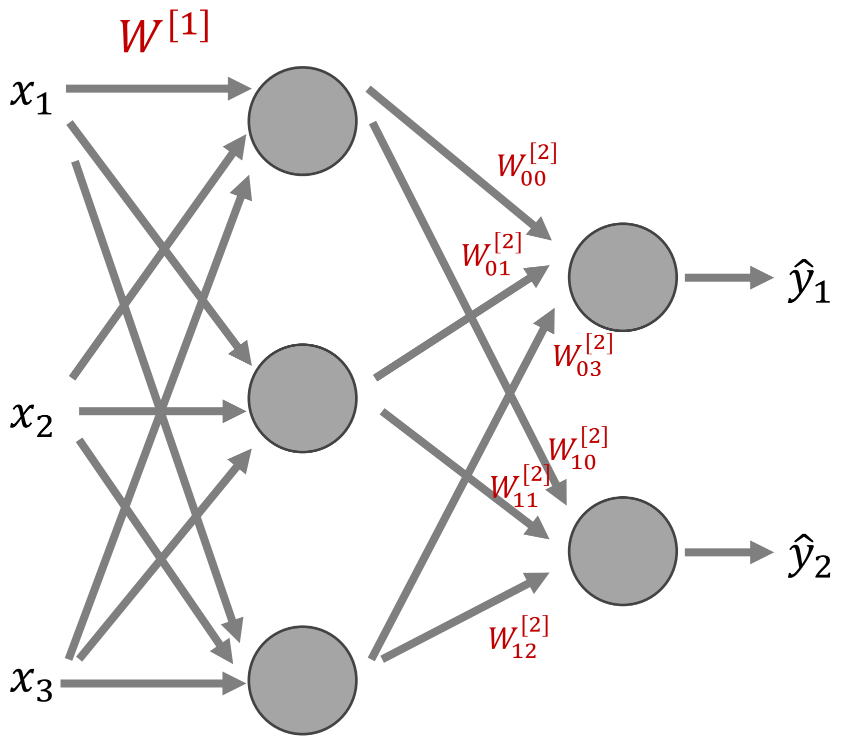
$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad W = \begin{bmatrix} W_{00} & W_{10} \\ W_{01} & W_{11} \\ W_{02} & W_{12} \end{bmatrix} \quad \hat{\boldsymbol{y}} = g(W^T \boldsymbol{x}) = g\left(\begin{bmatrix} W_{00} & W_{10} \\ W_{01} & W_{11} \\ W_{02} & W_{12} \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \right) = g\left(\begin{bmatrix} W_{00} & W_{01} & W_{02} \\ W_{10} & W_{11} & W_{12} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \right) = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix}$$

Output (number of layer's neurons / output variables)

ADDITION

Neural Networks and Matrix Multiplication (2)

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



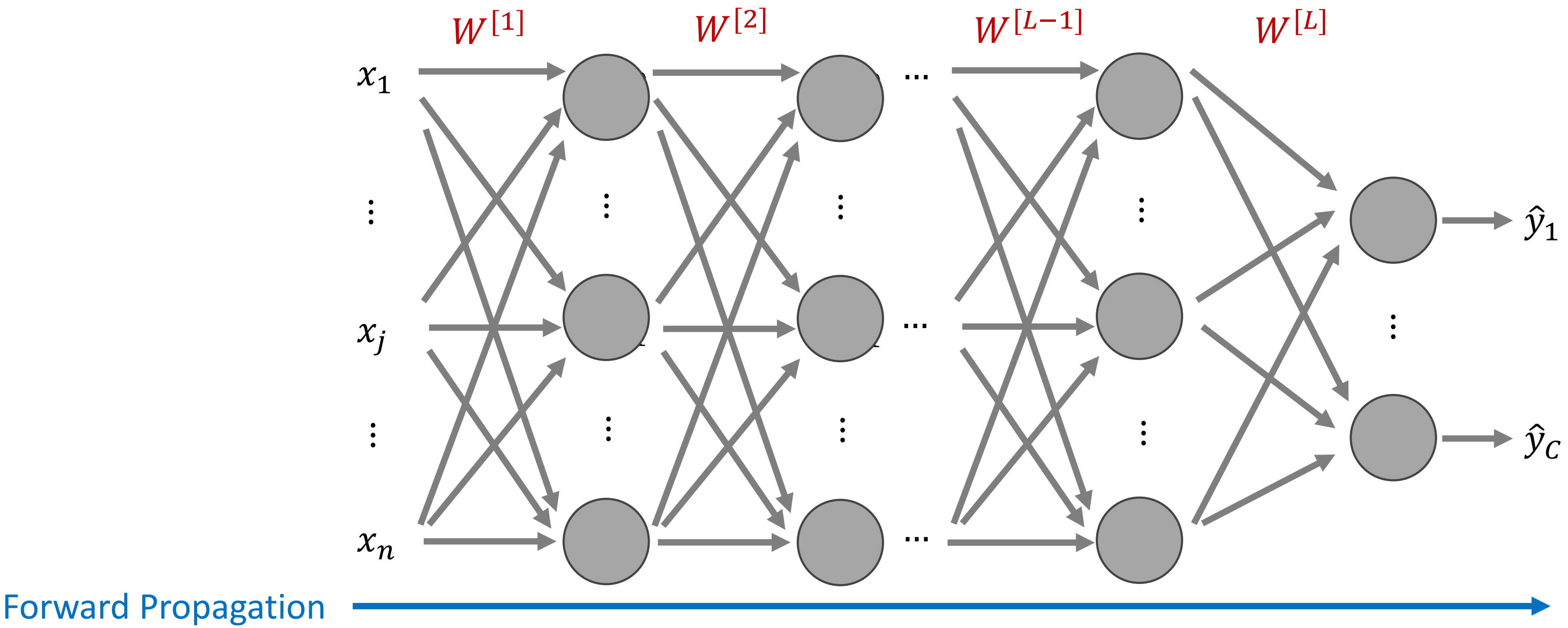
$$W^{[1]} = \begin{bmatrix} W^{[1]}_{00} & W^{[1]}_{10} & W^{[1]}_{20} \\ W^{[1]}_{01} & W^{[1]}_{11} & W^{[1]}_{21} \\ W^{[1]}_{02} & W^{[1]}_{12} & W^{[1]}_{22} \end{bmatrix}$$

$$W^{[2]} = \begin{bmatrix} W^{[2]}_{00} & W^{[2]}_{10} \\ W^{[2]}_{01} & W^{[2]}_{11} \\ W^{[2]}_{02} & W^{[2]}_{12} \end{bmatrix}$$

$$\hat{\boldsymbol{y}} = g^{[2]}\left(W^{[2]^T} g^{[1]}\left(W^{[1]^T} \boldsymbol{x}\right)\right) = g^{[1]}\left(\begin{bmatrix} W^{[2]}_{00} & W^{[2]}_{10} \\ W^{[2]}_{01} & W^{[2]}_{11} \\ W^{[2]}_{02} & W^{[2]}_{12} \end{bmatrix}^T g^{[2]}\left(\begin{bmatrix} W^{[1]}_{00} & W^{[1]}_{10} & W^{[1]}_{20} \\ W^{[1]}_{01} & W^{[1]}_{11} & W^{[1]}_{21} \\ W^{[1]}_{02} & W^{[1]}_{12} & W^{[1]}_{22} \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right)\right) = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix}$$

ADDITION

Neural Networks and Matrix Multiplication (3)



$$\hat{\boldsymbol{y}} = g^{[L]}\left(W^{[L]^T} \dots g^{[L-1]}\left(W^{[L-1]^T} \dots g^{[l]}\left(W^{[l]^T} \dots g^{[2]}\left(W^{[2]^T} g^{[1]}\left(W^{[1]^T}\boldsymbol{x}\right)\right)\right)\right)\right) = \begin{bmatrix} \hat{y}_1 \\ \dots \\ \hat{y}_C \end{bmatrix}$$

Outline

- Notation and Math Refresher
- Backpropagation
  - Backpropagation on different scenarios
  - Biological plausibility of backpropagation
- Automatic Differentiation
  - Reverse mode automatic differentiation
  - Comparison with other methods
- Introduction to PyTorch
  - Tensors
  - Modules & Functions
  - Loss function & Optimizers

Outline

- **Notation and Math Refresher**
- Backpropagation
  - Backpropagation on different scenarios
  - Biological plausibility of backpropagation
- Automatic Differentiation
  - Reverse mode automatic differentiation
  - Comparison with other methods
- Introduction to PyTorch
  - Tensors
  - Modules & Functions
  - Loss function & Optimizers

Notation

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

Scalar: not bolded, lower case

$$x$$

Vector: bolded, lower case

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

Matrix: bolded, upper case

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

Vector and Matrix Operations (1)

Scalar-by-scalar:

- $y(x) = wx$ scale x by w

Scalar-by-vector:

- $\boldsymbol{y}(\boldsymbol{x}) = w\boldsymbol{x} = w\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} wx_1 \\ wx_2 \end{bmatrix}$ scale **x** by w

Vector-by-vector:

- $y(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w}^\top \boldsymbol{x} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = [w_1\ w_2]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = w_1 x_1 + w_2 x_2$ weighted sum

Vector and Matrix Operations (2)

Vector-by-matrix:

- $\boldsymbol{y}(\boldsymbol{X}) = \boldsymbol{W}\boldsymbol{x} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{pmatrix} w_{11}x_1 + w_{12}x_2 \\ w_{21}x_1 + w_{22}x_2 \end{pmatrix}$

Matrix-by-matrix:

- $\boldsymbol{Y}(\boldsymbol{X}) = \boldsymbol{W}\boldsymbol{X} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} w_{11}x_{11} + w_{12}x_{21} & w_{11}x_{12} + w_{12}x_{22} \\ w_{21}x_{11} + w_{22}x_{21} & w_{21}x_{12} + w_{22}x_{22} \end{bmatrix}$

Hadamard product $\circ$ (element-wise product):

- $\boldsymbol{Y}(\boldsymbol{X}) = \boldsymbol{W} \circ \boldsymbol{X} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \circ \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} w_{11}x_{11} & w_{12}x_{12} \\ w_{21}x_{21} & w_{22}x_{22} \end{bmatrix}$
- For backpropagation (see later), element-wise multiplication between matrices

Vector and Matrix Operations (1)

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

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

Same

Vector Dot Product = Scalar

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

Multi-rows

Transposed Matrix Multiplication = Vector

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

Derivatives

Total 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 $\frac{\partial y}{\partial x_2}$)

Jacobian: ∇

- 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 $\boldsymbol{x} = [x_1, \ldots, x_n]^\top$

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

Matrix Calculus

Scalar-by-Vector (= 1D Vector)

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

Scalar-by-Matrix (= 2D Matrix)

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

Vector-by-Vector (= 2D Matrix)

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

Vector-by-Matrix (= 3D Matrix)

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

Along 3rd dimension

Outline

- Notation and Math Refresher
- **Backpropagation**
  - Backpropagation on different scenarios
  - Biological plausibility of backpropagation
- Automatic Differentiation
  - Reverse mode automatic differentiation
  - Comparison with other methods
- Introduction to PyTorch
  - Tensors
  - Modules & Functions
  - Loss function & Optimizers

RECAP

Gradient Descent

- Start at some w
- Pick a nearby w that reduces $J(w)$

$$w_j \leftarrow w_j - \gamma \frac{\partial J(w_0, w_1, \dots)}{\partial w_j}$$

- Repeat until minimum is reached

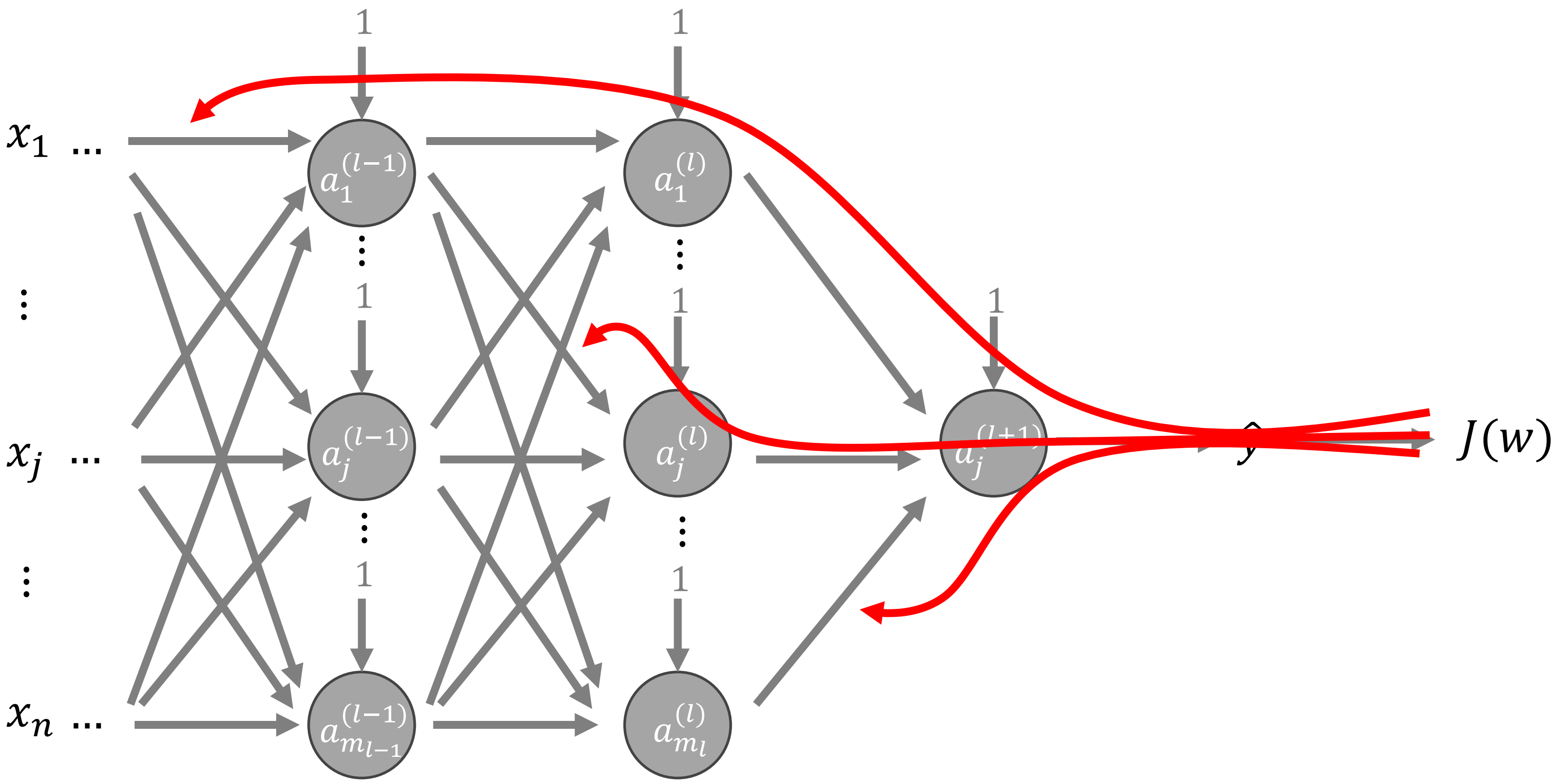
Learning Rate

Single-layer Neural Networks with Sigmoid

$$w_i \leftarrow w_i - \gamma(\hat{y} - y)\hat{y}(1 - \hat{y})x_i$$

Multi-layer Neural Networks?

Multi-layer Neural Networks



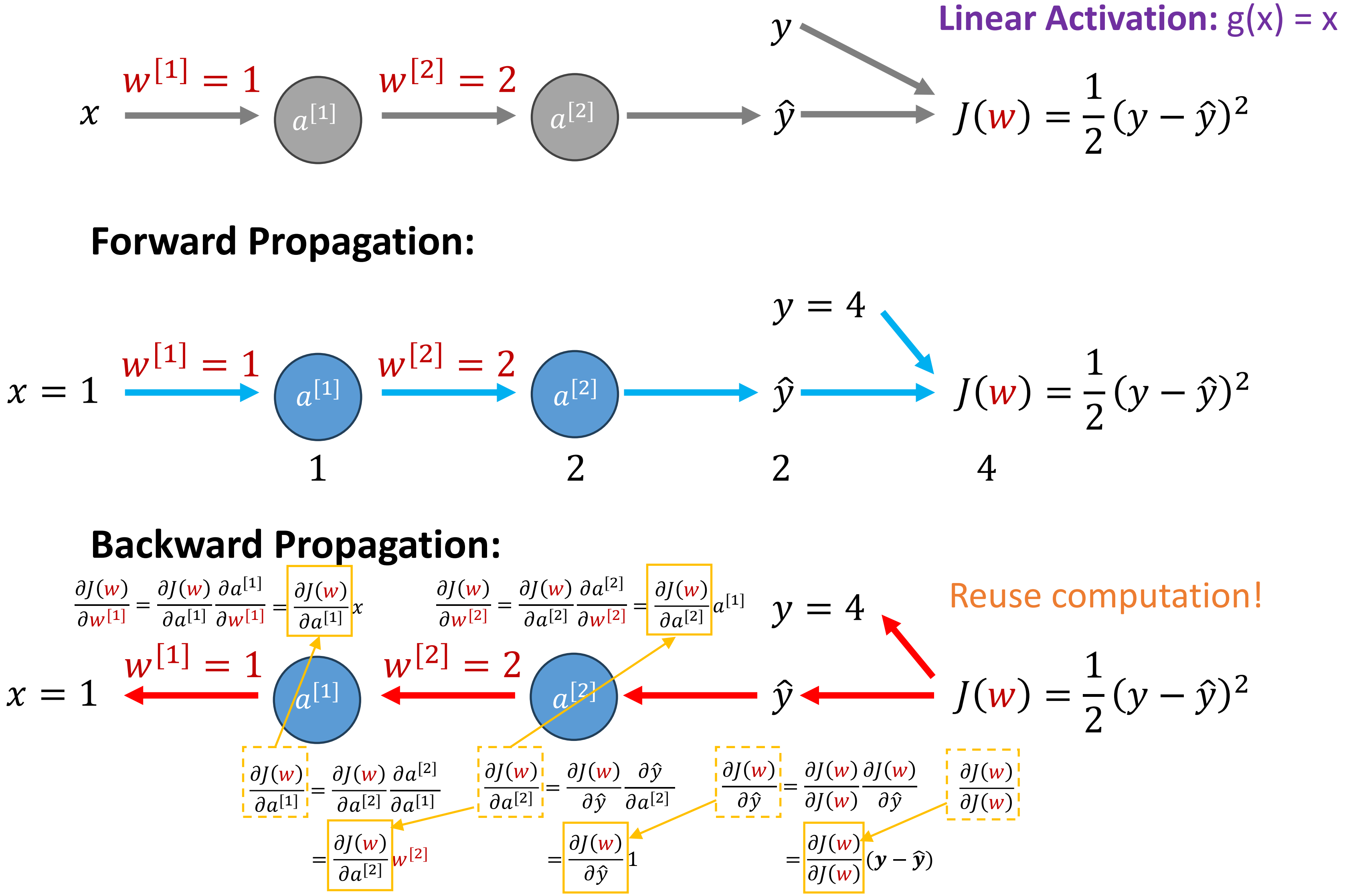
$$w_j \leftarrow w_j - \gamma \frac{\partial J(w_0, w_1, \dots)}{\partial w_j}$$

How to get $\frac{\partial J(w_0, w_1, \dots)}{\partial w_j}$ for all w_j?

Want:

- Can be implemented as a program
- Efficient

Backpropagation



Let $v_1, \dots, v_N$ be a *topological ordering* of the computation graph (i.e., parents comes before children).

$$v_N = J(w)$$

Forward propagation

For i in $1, \dots, N$

Compute v_i

Backward propagation

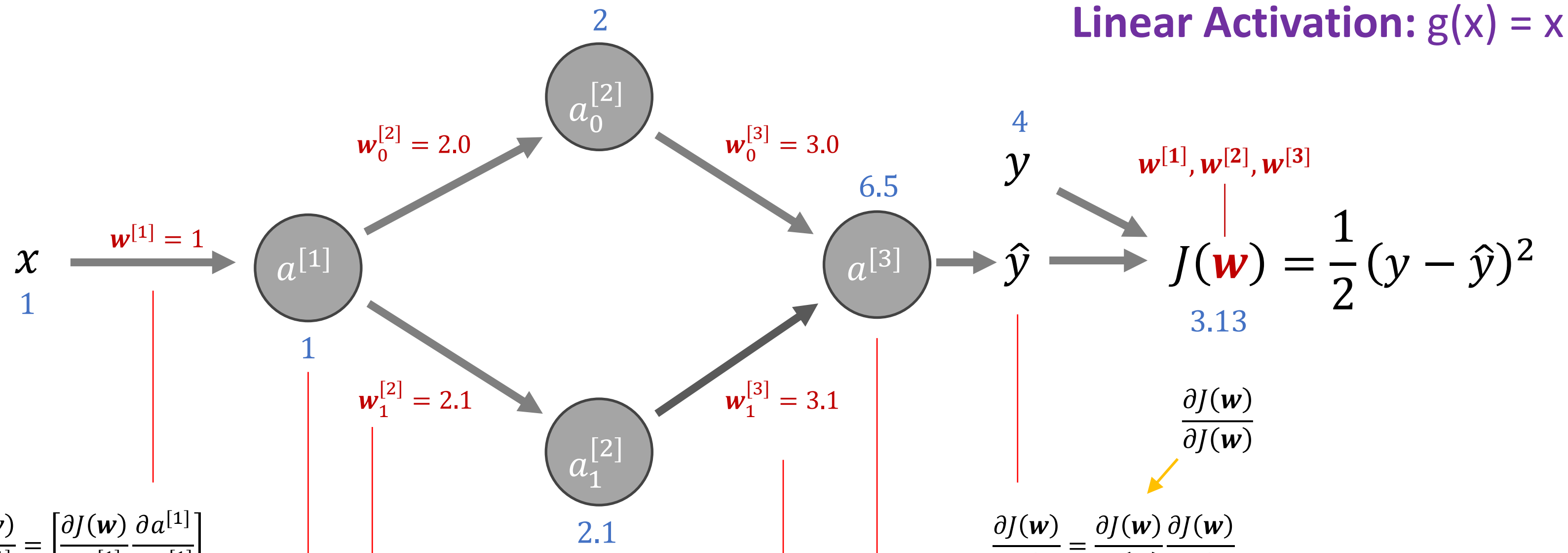
For i in $N-1, \dots, 1$

$$\frac{\partial J(w)}{\partial v_i} = \sum_{v_j \in Ch\,(v_i)} \frac{\partial J(w)}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

Children

Backpropagation with Branches

Linear Activation: g(x) = x



$$\frac{\partial J(\mathbf{w})}{\partial J(\mathbf{w})}$$

$$\frac{\partial J(\mathbf{w})}{\partial \hat{y}} = \frac{\partial J(\mathbf{w})}{\partial J(\mathbf{w})} \frac{\partial J(\mathbf{w})}{\partial \hat{y}}$$

$$\frac{\partial J(\mathbf{w})}{\partial a^{[3]}} = \left[\frac{\partial J(\mathbf{w})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a^{[3]}} \right]$$

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}^{[3]}} = \begin{bmatrix} \frac{\partial J(\mathbf{w})}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial \mathbf{w}_0^{[3]}} \\ \frac{\partial J(\mathbf{w})}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial \mathbf{w}_1^{[3]}} \end{bmatrix}$$

Vector of same size as $\mathbf{w}^{[3]}$

$$\frac{\partial J(\mathbf{w})}{\partial a^{[2]}} = \begin{bmatrix} \frac{\partial J(\mathbf{w})}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial a_0^{[2]}} \\ \frac{\partial J(\mathbf{w})}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial a_1^{[2]}} \end{bmatrix} \in \mathbb{R}^{2\times 1}$$

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}^{[2]}} = \begin{bmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{a}^{[2]}} \frac{\partial \mathbf{a}^{[2]}}{\partial w_0^{[2]}} \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{a}^{[2]}} \frac{\partial \mathbf{a}^{[2]}}{\partial w_1^{[2]}} \end{bmatrix}$$

$\mathbf{w}^{[2]} \in \mathbb{R}^{1\times 2}$

$$\frac{\partial J(\mathbf{w})}{\partial a^{[1]}} = \frac{\partial \mathbf{a}^{[2]}}{\partial a^{[1]}} \frac{\partial J(\mathbf{w})}{\partial \mathbf{a}^{[2]}} = \left[\frac{\partial J(\mathbf{w})}{\partial a_0^{[2]}} \frac{\partial a_0^{[2]}}{\partial a^{[1]}} + \frac{\partial J(\mathbf{w})}{\partial a_1^{[2]}} \frac{\partial a_1^{[2]}}{\partial a^{[1]}} \right]$$

Sum over children!

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}^{[1]}} = \left[\frac{\partial J(\mathbf{w})}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial \mathbf{w}^{[1]}} \right]$$

Let $v_1, \dots, v_N$ be a *topological ordering* of the computation graph (i.e., parents comes before children).

$$v_N = J(\mathbf{w})$$

Forward propagation

For i in $1, \dots, N$

Compute v_i

Backward propagation

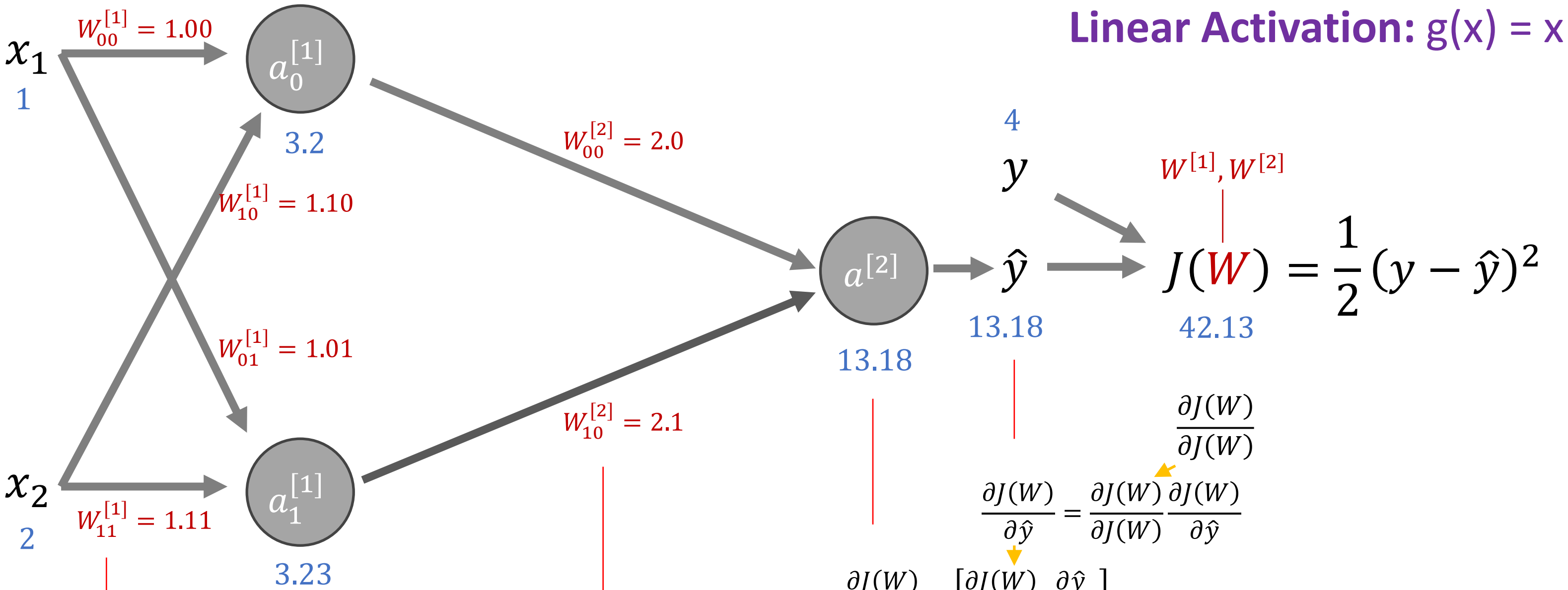
For i in $N-1, \dots, 1$

$$\frac{\partial J(\mathbf{w})}{\partial v_i} = \sum_{v_j \in Ch\,(v_i)} \frac{\partial J(\mathbf{w})}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

Children

Backpropagation with Many Features

Linear Activation: g(x) = x



$$\frac{\partial J(W)}{\partial J(W)}$$

$$\frac{\partial J(W)}{\partial \hat{y}} = \frac{\partial J(W)}{\partial J(W)} \frac{\partial J(W)}{\partial \hat{y}}$$

$$\frac{\partial J(W)}{\partial a^{[2]}} = \left[\frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a^{[2]}}\right]$$

$$\frac{\partial J(W)}{\partial W^{[2]}} = \begin{bmatrix} \frac{\partial J(W)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial W_{00}^{[2]}} \\ \frac{\partial J(W)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial W_{10}^{[2]}} \end{bmatrix}$$

$$\frac{\partial J(W)}{\partial \boldsymbol{a}^{[1]}} = \begin{bmatrix} \frac{\partial J(W)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial a_0^{[1]}} \\ \frac{\partial J(W)}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial a_1^{[1]}} \end{bmatrix} \in \mathbb{R}^{\mathbf{2\times1}}$$

$$\frac{\partial J(w)}{\partial W^{[1]}} = \frac{\partial \boldsymbol{a}^{[1]}}{\partial W^{[1]}} \left(\frac{\partial J(w)}{\partial \boldsymbol{a}^{[1]}}\right)^T = \begin{bmatrix} \frac{\partial J(W)}{\partial a_0^{[1]}} \frac{\partial a_0^{[1]}}{\partial W_{00}^{[1]}} & \frac{\partial J(W)}{\partial a_0^{[1]}} \frac{\partial a_0^{[1]}}{\partial W_{10}^{[1]}} \\ \frac{\partial J(W)}{\partial a_1^{[1]}} \frac{\partial a_1^{[1]}}{\partial W_{01}^{[1]}} & \frac{\partial J(W)}{\partial a_1^{[1]}} \frac{\partial a_1^{[1]}}{\partial W_{11}^{[1]}} \end{bmatrix}$$

$\boldsymbol{x} \in \mathbb{R}^{\mathbf{2\times1}}$

Matrix of same size as $W^{[1]}$

Let $v_1, \dots, v_N$ be a *topological ordering* of the computation graph (i.e., parents comes before children).

$$v_N = J(W)$$

Forward propagation

For i in $1, \dots, N$

Compute v_i

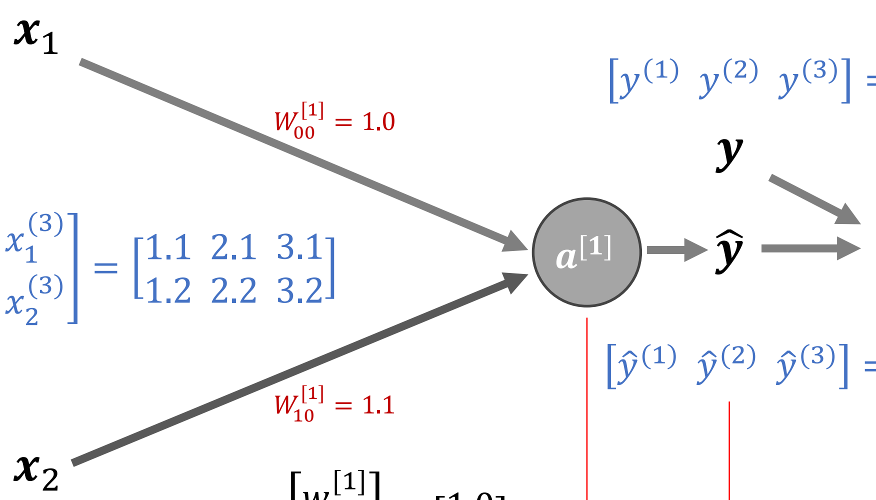
Backward propagation

For i in $N-1, \dots, 1$

$$\frac{\partial J(W)}{\partial v_i} = \sum_{v_j \in Ch\,(v_i)} \frac{\partial J(W)}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

Children

Backpropagation with Many Samples



Linear Activation: g(x) = x

$$\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} \end{bmatrix} = \begin{bmatrix} 1.1 & 2.1 & 3.1 \\ 1.2 & 2.2 & 3.2 \end{bmatrix}$$

$$\begin{bmatrix} w_{00}^{[1]} \\ w_{10}^{[1]} \end{bmatrix} = \begin{bmatrix} 1.0 \\ 1.1 \end{bmatrix}$$

$$\begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$W^{[1]}, W^{[2]}$

$$J\left(W^{[1]}\right) = \frac{1}{2 \times 3} \sum_{i=1}^{3} \left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

$$\begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \hat{y}^{(3)} \end{bmatrix} = \begin{bmatrix} 2.42 & 4.52 & 6.62 \end{bmatrix}$$

3.58

$$\frac{\partial J(W)}{\partial J(W)}$$

$$\frac{\partial J(W)}{\partial \hat{\boldsymbol{y}}} = \begin{bmatrix} \frac{\partial J(W)}{\partial J(W)} \frac{\partial J(W)}{\partial \hat{y}^{(1)}} & \frac{\partial J(W)}{\partial J(W)} \frac{\partial J(W)}{\partial \hat{y}^{(2)}} & \frac{\partial J(W)}{\partial J(W)} \frac{\partial J(W)}{\partial \hat{y}^{(3)}} \end{bmatrix}$$

$$\frac{\partial J(W)}{\partial \boldsymbol{a}^{[1]}} = \begin{bmatrix} \frac{\partial J(W)}{\partial \hat{y}^{(1)}} \frac{\partial \hat{y}^{(1)}}{\partial a^{[1](1)}} & \frac{\partial J(W)}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial a^{[1](2)}} & \frac{\partial J(W)}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial a^{[1](3)}} \end{bmatrix} \in \mathbb{R}^{1\times 3}$$

$$\frac{\partial J(W)}{\partial W^{[1]}} = \frac{\partial \boldsymbol{a}^{[1]}}{\partial W^{[1]}} \left(\frac{\partial J(W)}{\partial \boldsymbol{a}^{[1]}}\right)^T = \begin{bmatrix} \frac{\partial J(W)}{\partial \boldsymbol{a}^{[1](1)}} \frac{\partial \boldsymbol{a}^{[1](1)}}{\partial W_{00}^{[1]}} + \frac{\partial J(W)}{\partial \boldsymbol{a}^{[1](2)}} \frac{\partial \boldsymbol{a}^{[1](2)}}{\partial W_{00}^{[1]}} + \frac{\partial J(W)}{\partial \boldsymbol{a}^{[1](3)}} \frac{\partial \boldsymbol{a}^{[1](3)}}{\partial W_{00}^{[1]}} \\ \frac{\partial J(W)}{\partial \boldsymbol{a}^{[1](1)}} \frac{\partial \boldsymbol{a}^{[1](1)}}{\partial W_{10}^{[1]}} + \frac{\partial J(W)}{\partial \boldsymbol{a}^{[1](2)}} \frac{\partial \boldsymbol{a}^{[1](2)}}{\partial W_{10}^{[1]}} + \frac{\partial J(W)}{\partial \boldsymbol{a}^{[1](3)}} \frac{\partial \boldsymbol{a}^{[1](3)}}{\partial W_{10}^{[1]}} \end{bmatrix}$$

$\boldsymbol{x} \in \mathbb{R}^{2\times 3}$

Sum over all samples' gradient

Matrix of same size as $W^{[1]}$

Let $v_1, \dots, v_N$ be a *topological ordering* of the computation graph (i.e., parents comes before children).

$$v_N = J(W)$$

Forward propagation

For i in $1, \dots, N$

Compute v_i

Backward propagation

For i in $N-1, \dots, 1$

$$\frac{\partial J(W)}{\partial v_i} = \sum_{v_j \in Ch\,(v_i)} \frac{\partial J(W)}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

Children

Backpropagation in (Generalized) Matrix Form

Activation function

Weighted sum

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

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

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

$$\frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[L]}} = \frac{\partial f^{[L]}}{\partial \boldsymbol{W}^{[L]}}\left(\boxed{\frac{\partial g^{[L]}}{\partial f^{[L]}}}\right)^{T} \qquad \boldsymbol{\delta}^{[L]}$$

$$\frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[l+1]}} = \frac{\partial f^{[l+1]}}{\partial \boldsymbol{W}^{[l+1]}}\left(\boxed{\frac{\partial g^{[l+1]}}{\partial f^{[l+1]}}\cdots\frac{\partial f^{[L]}}{\partial g^{[L-1]}}\frac{\partial g^{[L]}}{\partial f^{[L]}}}\right)^{T} \qquad \boldsymbol{\delta}^{[l+1]}$$

$$\frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[l]}} = \frac{\partial f^{[l]}}{\partial \boldsymbol{W}^{[l]}}\left(\frac{\partial g^{[l]}}{\partial f^{[l]}}\frac{\partial f^{[l+1]}}{\partial g^{[l]}}\boxed{\frac{\partial g^{[l+1]}}{\partial f^{[l+1]}}\cdots\frac{\partial f^{[L]}}{\partial g^{[L-1]}}\frac{\partial g^{[L]}}{\partial f^{[L]}}}\right)^{T}$$

$$= \frac{\partial f^{[l]}}{\partial \boldsymbol{W}^{[l]}}\left(\boxed{\frac{\partial g^{[l]}}{\partial f^{[l]}}\frac{\partial f^{[l+1]}}{\partial g^{[l]}}\boldsymbol{\delta}^{[l+1]}}\right)^{T} \qquad \boldsymbol{\delta}^{[l]}$$

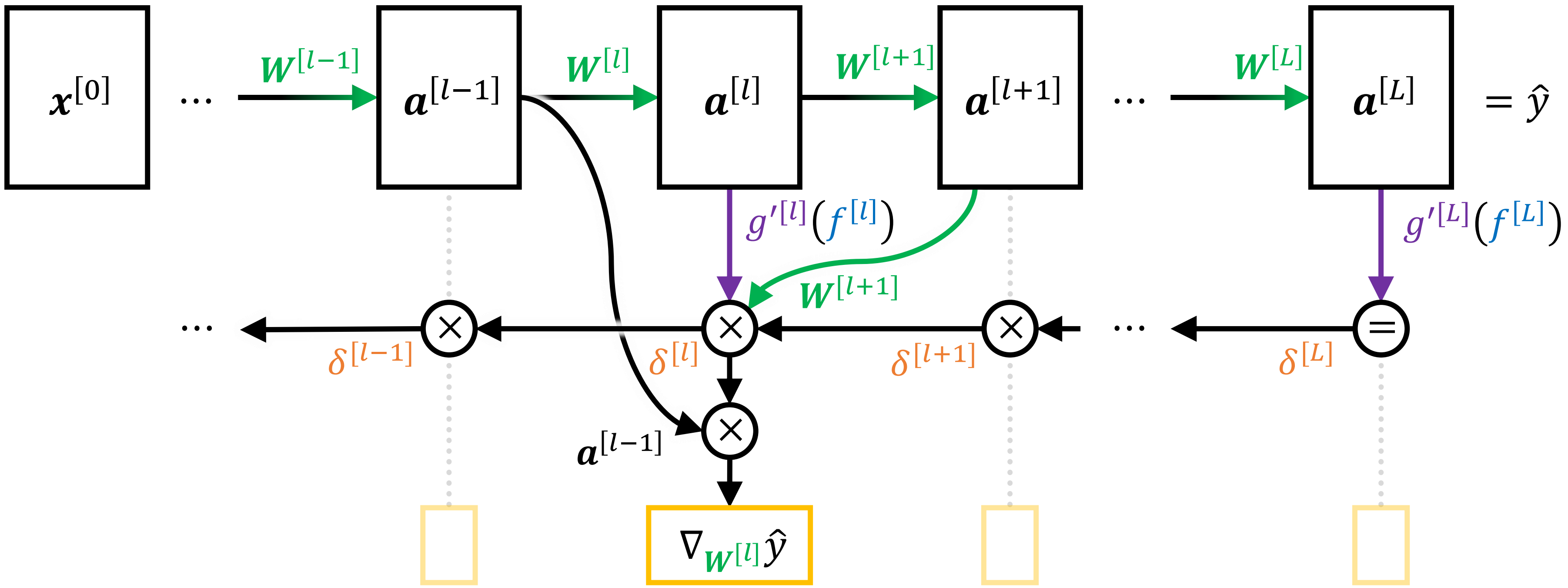
$$\boxed{\boldsymbol{\delta}^{[l]} = g'^{[l]}\left(f^{[l]}\right) W^{[l+1]}\boldsymbol{\delta}^{[l+1]} = g'^{[l]}\left(f^{[l]}\right) \circ \left(W^{[l+1]}\boldsymbol{\delta}^{[l+1]}\right)}$$

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

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

$$\boxed{\frac{\partial g^{[L]}}{\partial \boldsymbol{W}^{[l]}} = \boldsymbol{a}^{[l-1]}\left(\boldsymbol{\delta}^{[l]}\right)^{T}}$$

Backpropagation in (Generalized) Matrix Form



$$\nabla_{W^{[l]}} \hat{y} = a^{[l-1]} \left(\delta^{[l]}\right)^{\top}$$

$$\delta^{[l]} = \left[g'^{[l]}\left(f^{[l]}\right)\right] \circ \left(W^{[l+1]} \delta^{[l+1]}\right)$$

Matrix form is only for **notational convenience.** When in doubts, refer to the element wise derivatives

Notes on Backpropation

- Backpropagation is used to train most neural networks
- Despite its success, backprop is believed to be biologically <u>implausible</u>
  - **Synapses are unidirectional**, backprop runs two passes: forward and backward
  - How **derivative** is calculated in each **neuron**, like backprop, is unclear
  - Backpropagation path is **linear**, neurons aren't linear
  - Backpropagation signal is **instantaneous**, brain is not

Outline

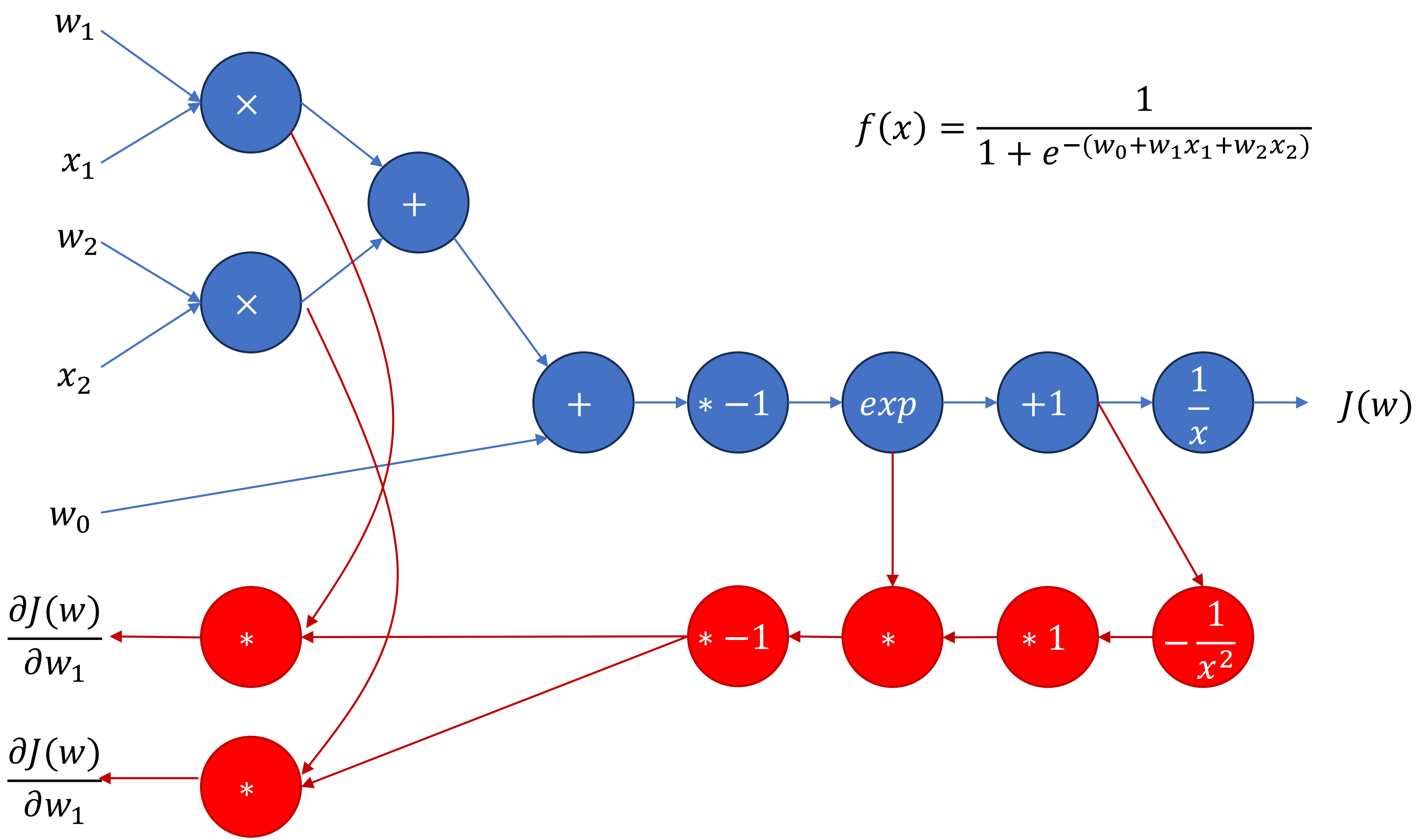
- Notation and Math Refresher
- Backpropagation
  - Backpropagation on different scenarios
  - Biological plausibility of backpropagation
- **Automatic Differentiation**
  - Reverse mode automatic differentiation
  - Comparison with other methods
- Introduction to PyTorch
  - Tensors
  - Modules & Functions
  - Loss function & Optimizers

Automatic Differentiation (AD)

- Backpropagation is a **special case** of automatic differentiation
  - Applied to neural networks, i.e., $\mathbb{R}^N \rightarrow \mathbb{R}$ (many features to one loss)
  - In practice, we use AD for neural networks
- AD has two modes: **forward** mode and **reverse** mode
  - Backpropagation is a special case of reverse mode AD
- Example library:
  - **PyTorch autograd**
    - Just need to implement the forward pass (e.g., layers), the backward pass is done <u>automatically</u>
  - **Jax grad**
  - **Tensorflow autodiff**

Reverse Mode Differentiation

$$f(x) = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2)}}$$



AD vs **Symbolic Differentiation**

What we have done in the previous lectures!

n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

- Disadvantages?
  - Symbolic differentiation results in **complex** and redundant expressions
  - Can **only handle math**, not procedure
- The goal is to **compute derivatives**, <u>not</u> finding a derivative formula

Credit: https://www.semanticscholar.org/paper/Automatic-differentiation-in-machine-learning%3A-a-Baydin-Pearlmutter/643da4c4de1954daeac571a82367241db012a8bf

AD vs **Numerical Differentiation**

$$\frac{\partial f(x_1, \dots, x_N)}{\partial \boldsymbol{x_i}} \approx \frac{f(x_1, \dots, \boldsymbol{x_i} + \epsilon, \dots, x_N) - f(x_1, \dots, \boldsymbol{x_i} - \epsilon, \dots, x_N)}{2\epsilon}$$

<u>Numerical Differentiation</u>

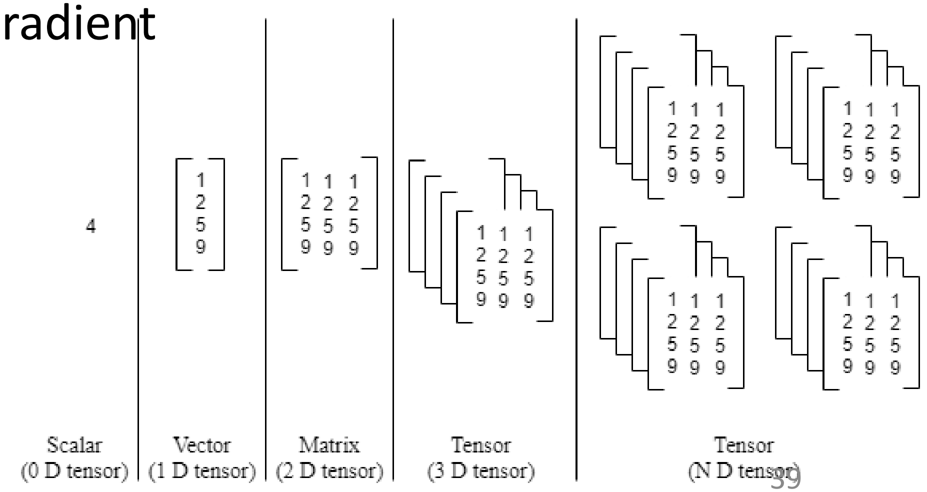
- **Expensive**, need to do forward pass for each derivative
- Introduce **numerical errors**
- Normally, only used for testing

Outline

- Notation and Math Refresher
- Backpropagation
  - Backpropagation on different scenarios
  - Biological plausibility of backpropagation
- Automatic Differentiation
  - Reverse mode automatic differentiation
  - Comparison with other methods
- **Introduction to PyTorch**
  - Tensors
  - Modules & Functions
  - Loss function & Optimizers

Tensors

- N-dimensional (e.g., 1D, 2D, 3D, …) array representation
- **Similar to Numpy** arrays, with GPU support
- Contain information about computational graph
- Important functions:
  - `torch.tensor(…, requires_grad=True)`
    - Build tensor, and possibly set the tensor to require gradient
  - `backward()`
    - Performs backpropagation



Credit: analyticsvidhya.com

Tensors: Example

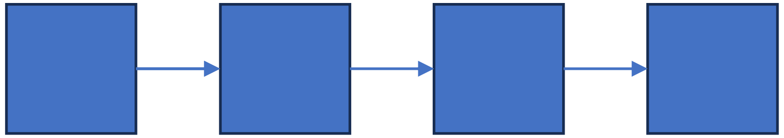
```
import torch  # Import PyTorch

x = torch.tensor([0.])
w1 = torch.tensor([0.], requires_grad=True)
w2 = torch.tensor([0.], requires_grad=True)

y = w2* torch.sigmoid(w1*x)
y.backward()

print(x,w1,w2)   tensor([0.]) tensor([0.], requires_grad=True) tensor([0.], requires_grad=True)
print(w1.grad)   tensor([0.])
print(w2.grad)   tensor([0.5000])
print(x.grad)    None
```

Modules and Functions API

```
def __init__(self, ...):
    ...
def forward(self, ...):
    ...
```

Neural Networks Module (`torch.nn`)

- Containers
  - Module (`torch.nn.Module`)
  - Sequential (`torch.nn.Sequential`)

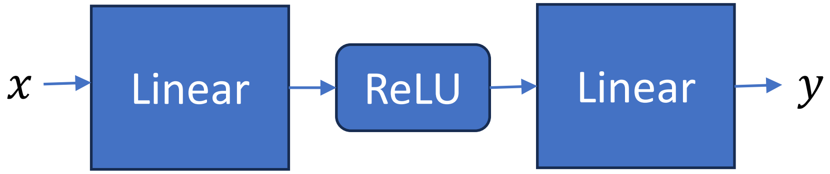


- Linear Layers
  - Linear: Single Layer NN without activation (`torch.nn.Linear`)
- Non-linear activation functions
  - ReLU (`torch.nn.ReLU`)
  - Sigmoid (`torch.nn.Sigmoid`)
  - Softmax (`torch.nn.Softmax`)

Functional version (`torch.nn.functional`)

Modules and Functions API: Example

```python
class NeuralNetRegressor(torch.nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.linear1 = torch.nn.Linear(input_size, hidden_size)
        self.linear2 = torch.nn.Linear(hidden_size, 1)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        f1 = self.linear1(x)
        a1 = self.relu(f1)
        f2 = self.linear2(x)
        return f2

model1 = NeuralNetClassifier(2,8) # 2 features, 8 hidden neurons
model2 = torch.nn.Sequential(torch.nn.Linear(2,8), torch.nn.ReLU(), torch.nn.Linear(8,1)) # same
```




```python
w1 = torch.tensor(8, 2, requires_grad=True)
w2 = torch.tensor(2, 1, requires_grad=True)

def neural_net_regressor(x): # also the same
    f1 = torch.nn.functional.linear(x,w1)
    a1 = torch.nn.functional.relu(f1)
    return torch.nn.functional.linear(a1,w2)
```

Loss Functions

- Mean Squared Error (torch.nn.MSELoss)
- Binary Cross Entropy (torch.nn.BCELoss)
- Cross Entropy (torch.nn.CrossEntropyLoss)

```
loss_function = torch.nn.MSELoss()
loss_function = torch.nn.BCELoss()
loss_function = torch.nn.CrossEntropyLoss()
```

Optimizers

Optimizers (`torch.optim`)

- Stochastic Gradient Descent (`torch.optim.SGD`)
- Adam (`torch.optim.Adam`)

Important functions:

- `optimizer.zero_grad()`
  - Set all gradients to zero, before computing gradient
- `optimizer.step()`
  - Update the weights, and let the optimizer know that one step of optimization is done

```
optimizer = torch.optim.SGD([w_1, w_2], lr=0.01)
optimizer = torch.optim.Adam([w_1, w_2], lr=0.01)
```

Example 1: Training NN using functional API

```
model = neural_net_regressor // w1 and w2 are declared before

optimizer = torch.optim.Adam([w1,w2], lr=0.01)


for epoch in range(num_epochs):
    optimizer.zero_grad()          Zero the gradients in the weight tensor
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()                Do backpropagation
    optimizer.step()               Update the weights
```

Example 2: Training NN using modular API

```
model = NeuralNetRegressor()

optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
loss_function = torch.nn.MSELoss()

for epoch in range(num_epochs):
    optimizer.zero_grad()
    y_pred = model(x)
    loss = loss_function(y_pred, y)
    loss.backward()
    optimizer.step()
```

Retrieve all the weights in the model

Zero the gradients in the weight tensor

Do backpropagation

Update the weights

Example 3: Building and Training a NN

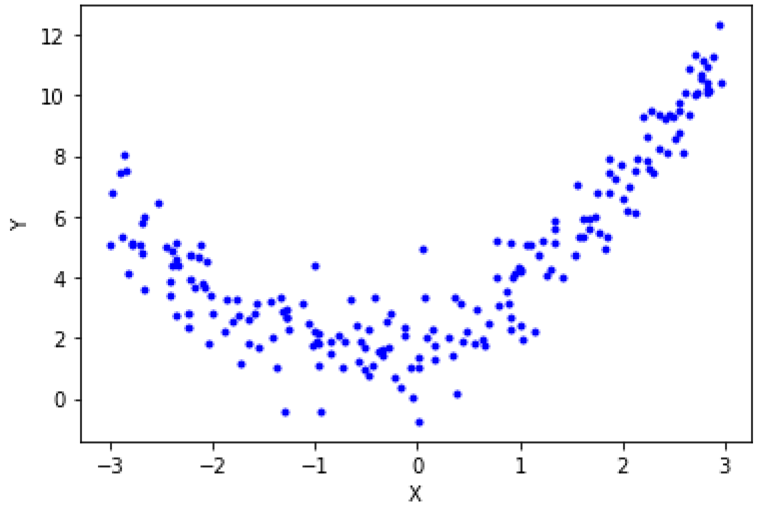
Step 0: Import the package

```
import torch
```

Example 3: Building and Training a NN

Step 1: Create a Dataset

```
# Generate 100 random data points
# Normally distributed with 0 mean and 1 variance
x = 5*torch.randn(100, 1)

# Set true output y = f(x) = x^2
y = torch.square(x)

# Add noise to the true output y, noise is of the same shape as y
# Noise is normally distributed with 0 mean and 1 variance
y += torch.randn_like(y)
```



Credit: analyticsvidhya.com

Example 3: Building and Training a NN

Step 2: Prepare the Model and Optimizer

```
# Instantiate the model
model = NeuralNetRegressor(1,8)


# Define the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=1e-1)


# Define loss using a predefined loss function
loss_function = torch.nn.MSELoss()
```

Example 3: Building and Training a NN

Step 3: Train the Model

```
n_epoch = 10000
for epoch in range(n_epoch):
    optimizer.zero_grad() # Set the gradients to 0

    y_pred = model(x) # Get the model predictions

    loss = loss_function(y_pred, y) # Get the loss

    if epoch%1000==0:
        print(f"Epoch {epoch}: traing loss: {loss}") # Print stats

    loss.backward() # Compute the gradients

    optimizer.step() # Take a step to optimize the weights
```

Example 3: Building and Training a NN

Step 4: Test the model

```
x2 = 5*torch.randn(5, 1) # Generate new data points
print(x2.tolist())
# [[5.367788791656494], [0.8833026885986328], [-0.6179562211036682], [4.864236831665039], [5.316009998321533]]

y2 = torch.square(x2) # Ground truth (true output) with no noise
print(y2.tolist())
# [[28.813156127929688], [0.7802236676216125], [0.38186988234519996], [23.660799026489258], [28.25996208190918]]

y_pred = model(x2)
print(y_pred.tolist())
# [[29.595108032226562], [0.8340979814529419], [0.5545129179954529], [23.755483627319336], [28.972129821777344]]
```

Notes on Implementation

- If you want to implement custom layers, loss functions, etc, you have to make sure that they are **differentiable**.
- Non-differentiable functions can also be used, but it requires a gradient estimator to estimate the gradient for backpropagation.

PyTorch Resources

Books on Deep Learning (with PyTorch implementation)

- https://d2l.ai

Good online tutorial:

- https://pytorch.org/tutorials/
- https://web.stanford.edu/class/cs224n/materials/CS224N_PyTorch_Tutorial.html

Feel free to try out the examples in the slides!

Summary

- Backpropagation
  - Backpropagation on different scenarios:
    - Path, branches, many features, and many samples (sum the gradients)
  - Biological plausibility of backpropagation – believed to be **not feasible**
- Automatic Differentiation
  - Reverse mode automatic differentiation – **backprop is a special case**: $\mathbb{R}^N \rightarrow \mathbb{R}$
  - Comparison with other methods: **symbolic** and **numerical** differentiation
- Introduction to PyTorch
  - Tensors
    - **n-dimensional array representation** with GPU support
    - Maintain **computational graph**
  - Modules & Functions: Linear (linear), ReLU (relu), etc – they are equivalent
  - Loss function & Optimizers

Coming Up Next Week

- **Neural Networks (Deep Learning) Architectures**
  - Convolutional Neural Networks
  - Recurrent Neural Networks
  - Attention Neural Networks (the basis of **ChatGPT**) – maybe…
- **Applications of Deep learning**
  - Vision
  - Speech and Language
- **Issues with Deep Learning**
  - Overfitting
  - Exploding/vanishing gradients

To Do

- **Lecture Training 9**
  - +100 Free EXP
  - +50 Early bird bonus