# Introduction to Neural Networks

**Phạm Quang Nhật Minh**

minhpham0902@gmail.com

December 29, 2024

# Lecture outline



- Neural units
- The XOR problem
- Feed-Forward Neural Networks
- Training Neural Nets

# Lecture outline

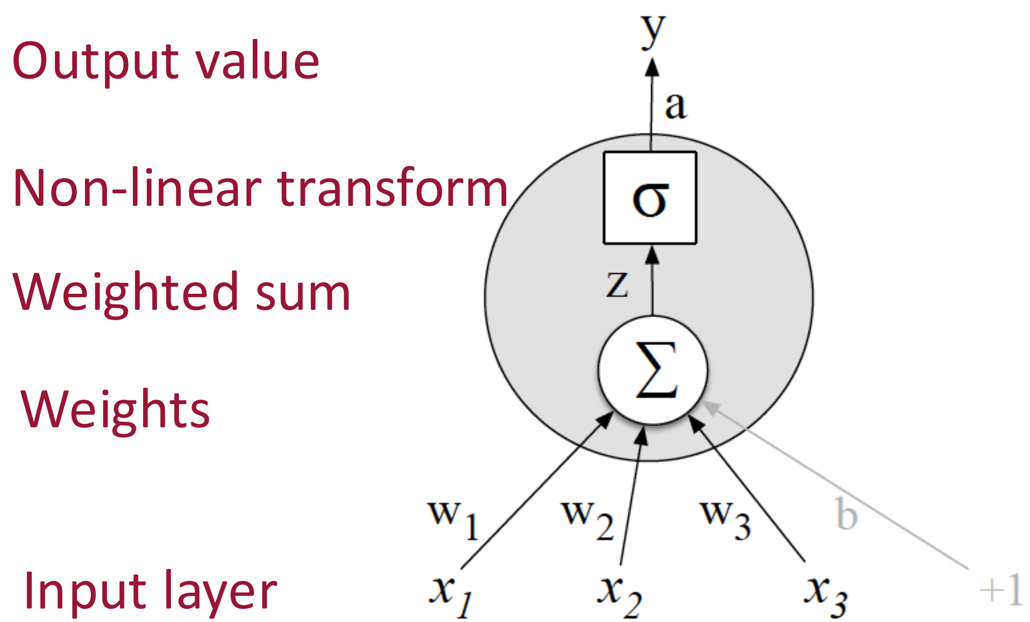


- Neural units
- The XOR problem
- Feed-Forward Neural Networks
- Training Neural Nets

# Neural Network unit



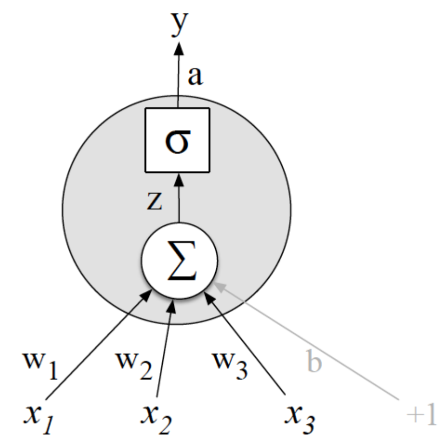
- The building block of a neural network
  - Weight vector $w = w_1 \ldots w_n$
  - Bias term $b$
  - Activation function $f$



# Neural unit



- The building block of a neural network
  - Weight vector $w = w_1 \dots w_n$
  - Bias term $b$
  - Activation function $f$ (non-linear)




- Output of a neural unit

$$y = a = f(z)$$

Here:

- $z$ is the weighted sum

$$z = \sum_i w_i x_i + b$$

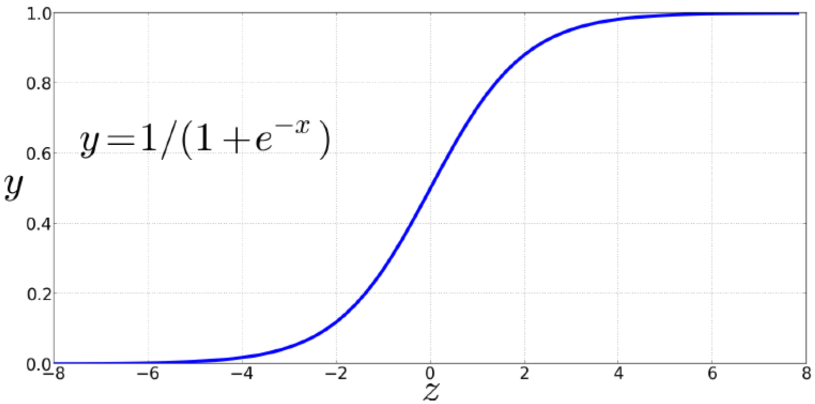
$$z = w \cdot x + b$$

# Non-Linear Activation functions



- There are many non-linear activation functions
  - Sigmoid

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




  - Tanh

$$y = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

  - Rectified Linear (ReLU)

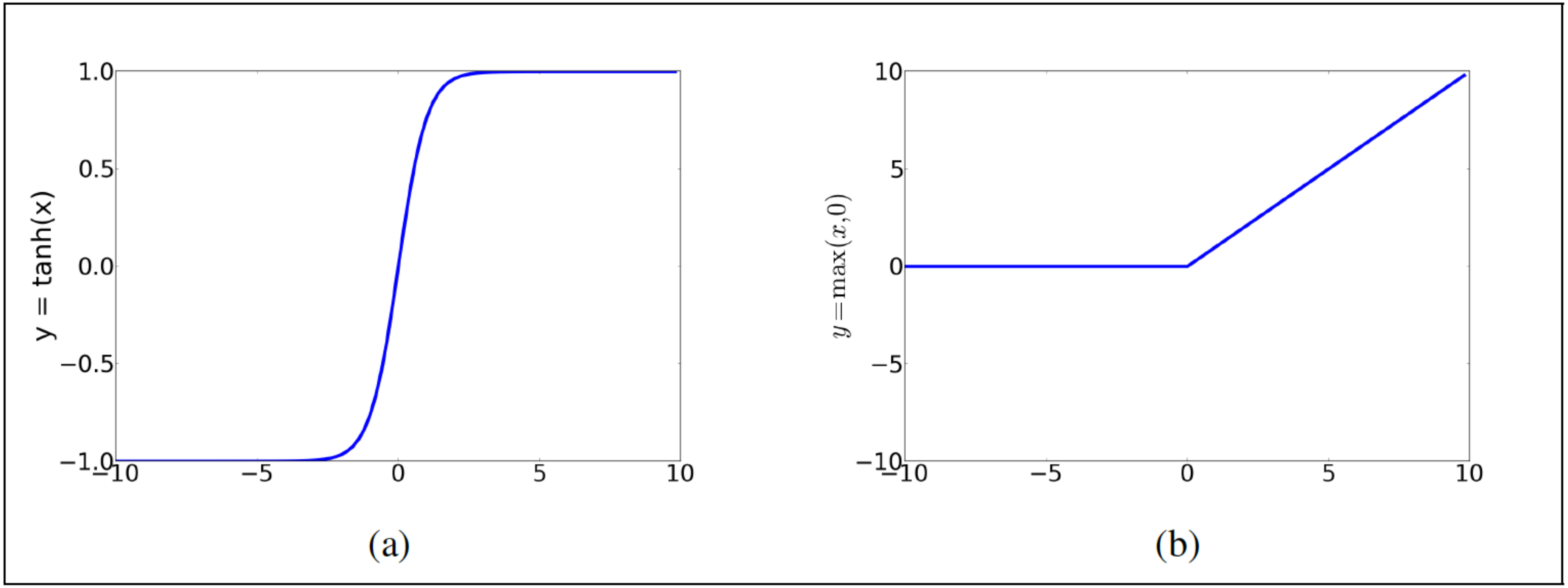
$$y = \max(x, 0)$$

  - PReLU
  - …

# Activation functions



- Tanh and ReLU functions



# An example

*Suppose a unit has:*

- $w = [0.2, 0.3, 0.9]$
- $b = 0.5$

What happens with input x:

- $x = [0.5, 0.6, 0.1]$

$$y = s(w \cdot x + b) =$$

# An example

*Suppose a unit has:*

- $w = [0.2, 0.3, 0.9]$
- $b = 0.5$

What happens with the following input x?

- $x = [0.5, 0.6, 0.1]$

$$y = s(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

# An example

*Suppose a unit has:*

- $w = [0.2, 0.3, 0.9]$
- $b = 0.5$

What happens with input x:

- $x = [0.5, 0.6, 0.1]$

$$y = s(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5 * .2 + .6 * .3 + .1 * .9 + .5)}} =$$

# An example

*Suppose a unit has:*

- $w = [0.2, 0.3, 0.9]$
- $b = 0.5$

What happens with input x:

- $x = [0.5, 0.6, 0.1]$

In Python:

```
import numpy as np
y = 1/(1+np.exp(-(np.dot(w,x) + b)))
```

$$y = s(w \cdot x + b) = \frac{1}{1+e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1+e^{-(.5 \times .2 + .6 \times .3 + .1 \times .9 + .5)}} = \frac{1}{1+e^{-0.87}} = .70$$

# Lecture outline



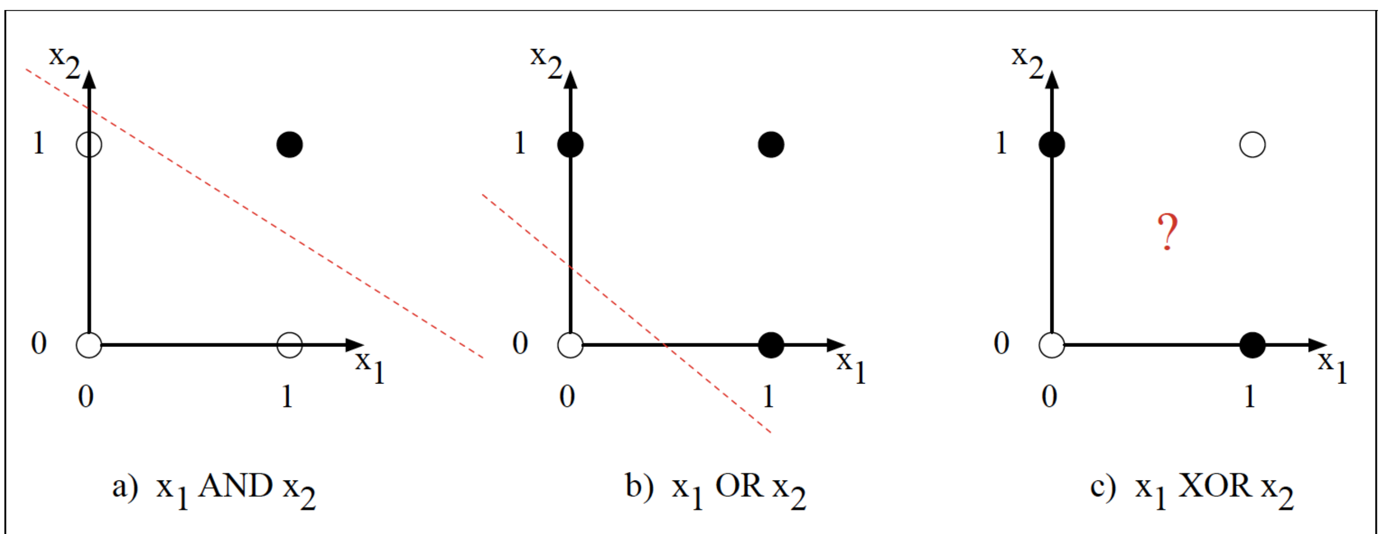
- Neural units
- The XOR problem
- Feed-Forward Neural Networks
- Training Neural Nets

# Boolean functions



- AND, OR, XOR functions

| AND | | | OR | | | XOR | | |
|---|---|---|---|---|---|---|---|---|
| x1 | x2 | y | x1 | x2 | y | x1 | x2 | y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |




a) $x_1$ AND $x_2$ b) $x_1$ OR $x_2$ c) $x_1$ XOR $x_2$

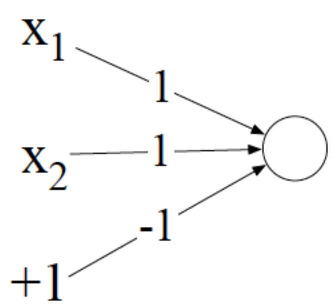
# Boolean functions using Perceptron

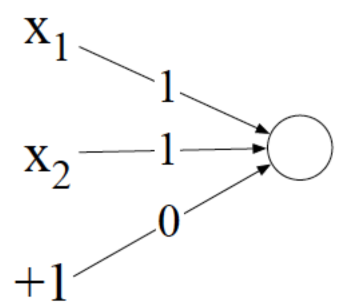


- Using Perceptron to compute above functions

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

- We can use Perceptron (a) for AND and (b) for OR



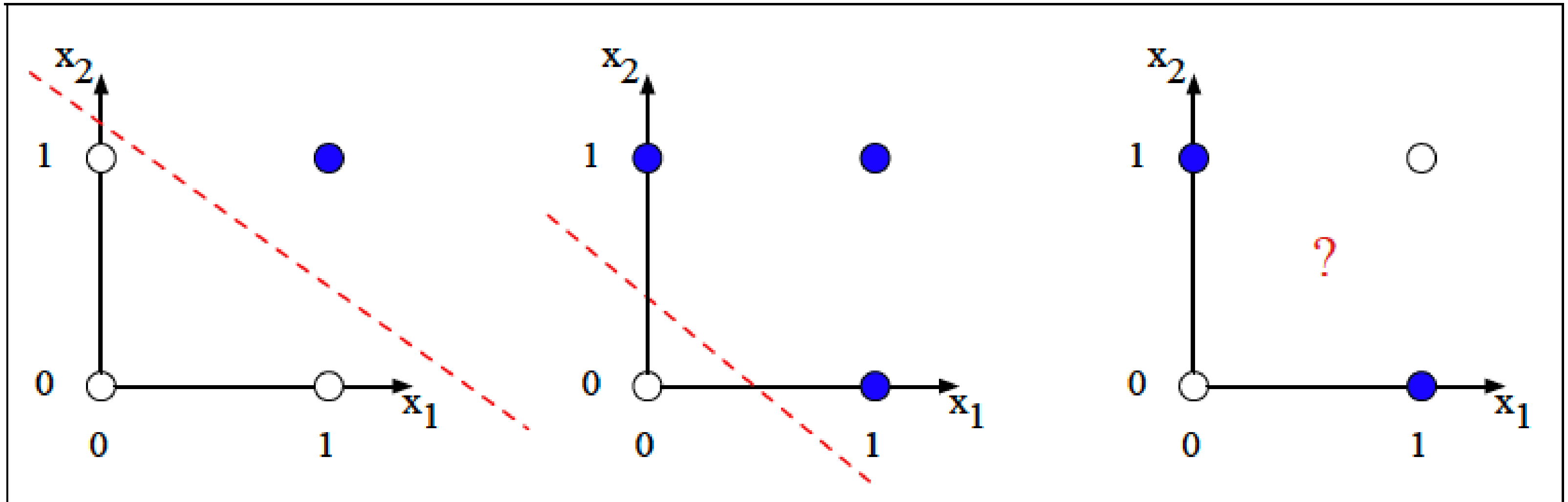




| AND | | | OR | | | XOR | | |
|---|---|---|---|---|---|---|---|---|
| x1 | x2 | y | x1 | x2 | y | x1 | x2 | y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# The XOR problem



- It's not possible to build a perceptron to compute logical XOR!
- The solution: neural networks!

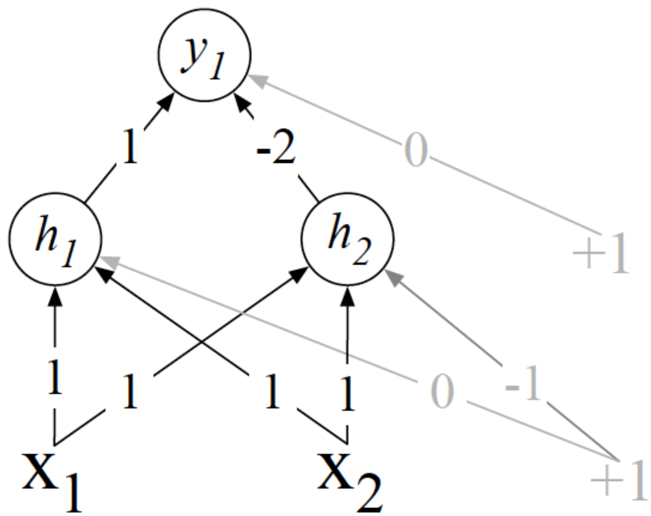



a) $x_1$ AND $x_2$ b) $x_1$ OR $x_2$ c) $x_1$ XOR $x_2$

# The solution: neural networks



- XOR solution with two-layer neural network and ReLU activation functions

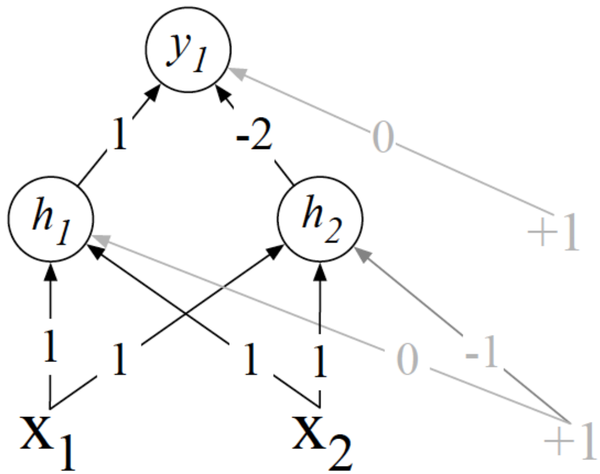



XOR

| x1 | x2 | y |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# The solution: neural networks






XOR

| x1 | x2 | y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| x1 | x2 | h1 | h2 | y1 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 2 | 1 | 0 |

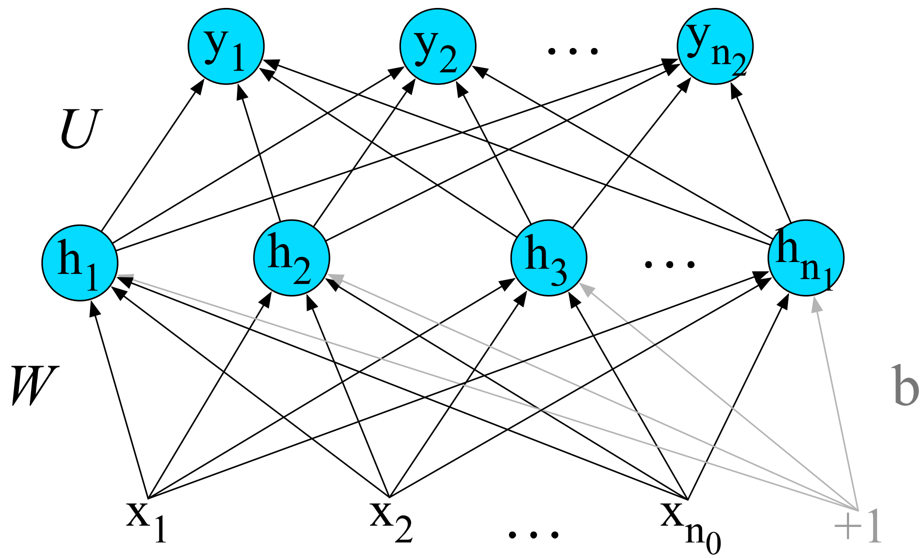
# Lecture outline



- Neural units
- The XOR problem
- Feed-Forward Neural Networks
- Training Neural Nets

# Feedforward Neural Networks

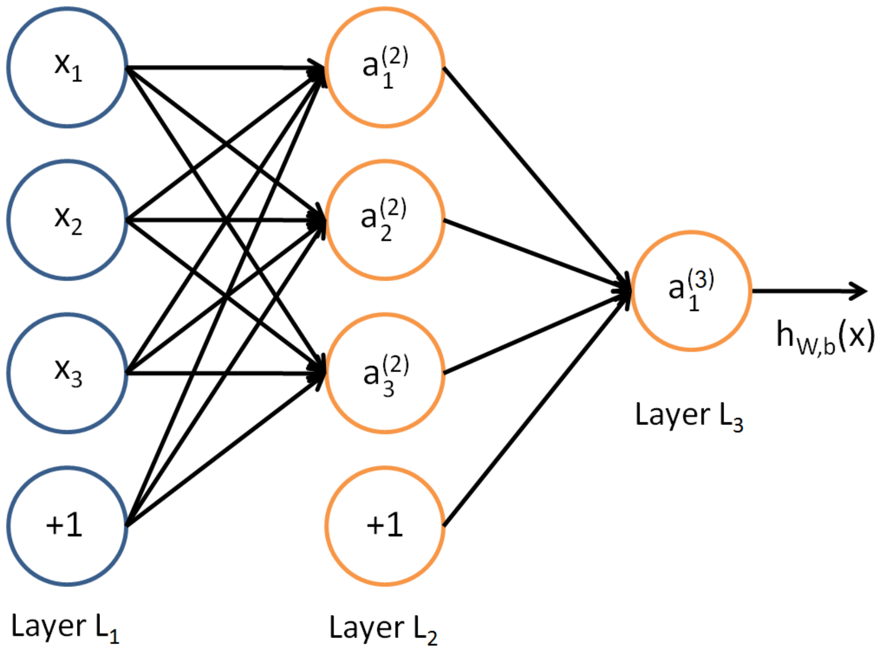
- Can also be called **multi-layer perceptrons** (or **MLPs**) for historical reasons



# Feed-forward neural networks



- Simple feed-forward neural networks inclue:
  - Input units
  - Hidden units
  - Output units



# Feed-forward neural networks



- A single hidden unit has:
  - parameters $w$ (the weight vector) and
  - Bias term $b$ (scalar)
- Combine weight vectors and bias terms of units into matrix $W$ and vector $\mathbf{b}$

# Feed-forward neural networks



- A single hidden unit has:
  - parameters $w$ (the weight vector) and
  - Bias term $b$ (scalar)
- Combine weight vectors and bias terms of units into matrix $W$ and vector $\mathbf{b}$
- Output of the hidden layer, the vector $h$ with sigmoid as the activation function

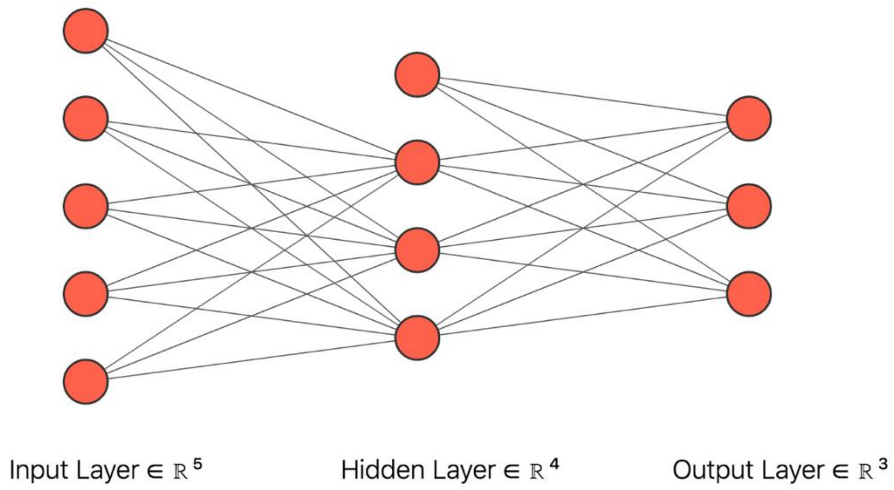
$$h = \sigma(Wx + \mathbf{b})$$

  - The activation function is applied to vector element-wise
    - $g([z_1, z_2, z_3]) = [g(z_1), g(z_2), g(z_3)]$

# Dimensions of vectors and matrices



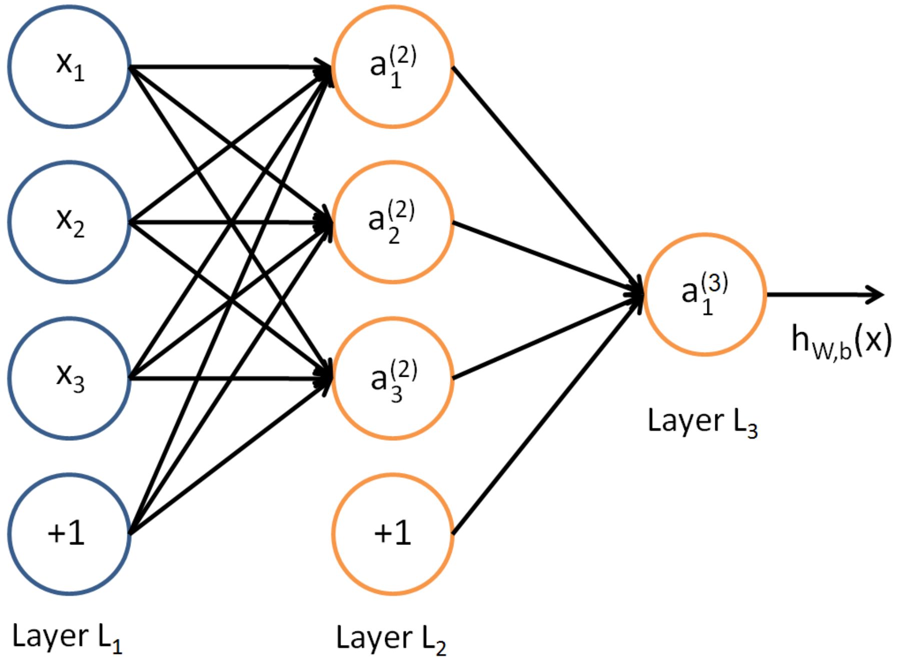
- Input layer (layer 0): $x \in \mathbb{R}^{n_0}$
- Hidden layer (layer 1): $h \in \mathbb{R}^{n_1}$, $b \in \mathbb{R}^{n_1}$
- Weight matrix: $W \in \mathbb{R}^{n_1 \times n_0}$



# Output layer



- If we do binary classification and use sigmoid function at the output layer, we use a single output unit

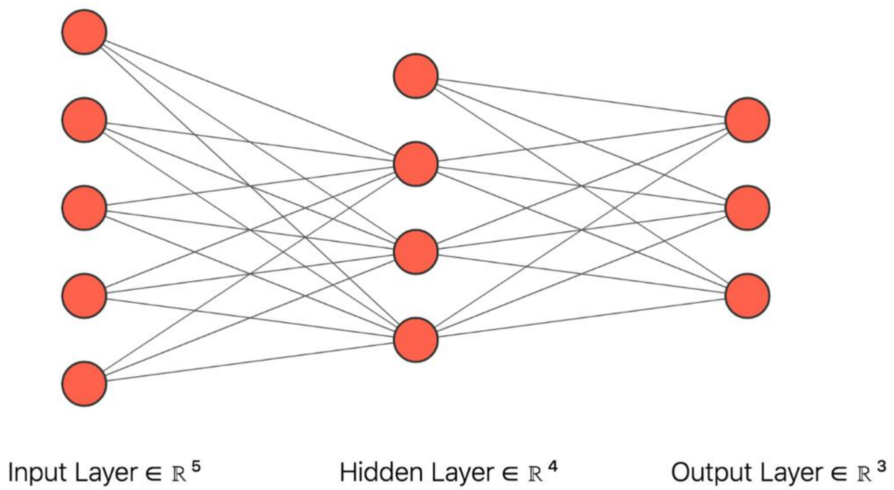


# Output layer



- For multi-class classification, we use *K* units in output layer and softmax function
  - *K* is the number of classes

$$p(y=c|x) = \frac{e^{w_c \cdot x + b_c}}{\sum_{j=1}^{k} e^{w_j \cdot x + b_j}}$$



# Binary Logistic Regression as a 1-layer Network

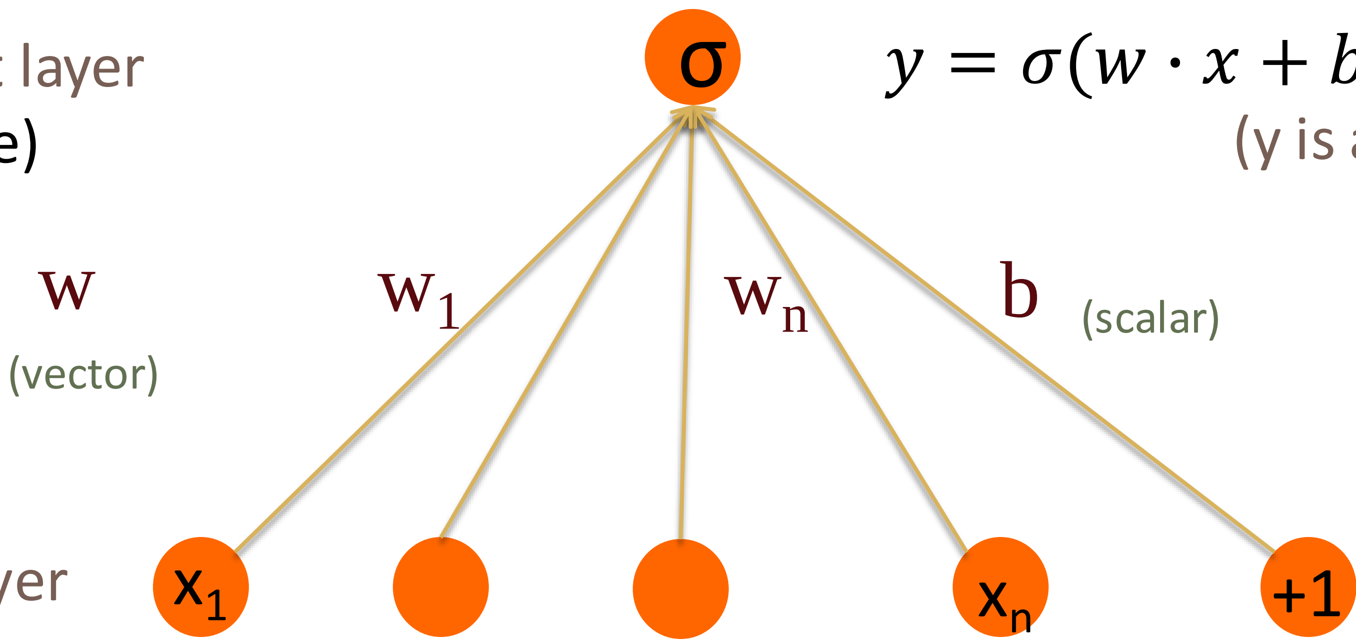


(we don't count the input layer in counting layers!)

Output layer
(σ node)

$$y = \sigma(w \cdot x + b)$$

(y is a scalar)



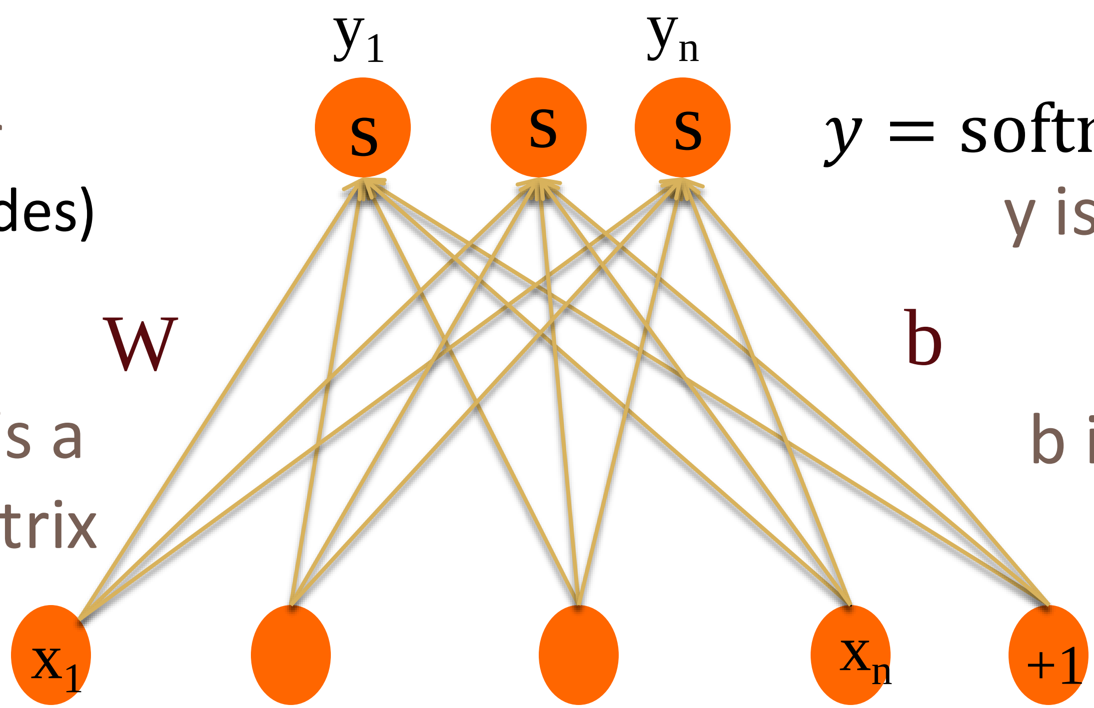

Input layer
vector x

# Multinomial Logistic Regression as a 1-layer Network



Fully connected single layer network

Output layer
(softmax nodes)




$y = \text{softmax}(Wx + b)$

y is a vector

W is a matrix

b is a vector

Input layer
scalars

# Reminder: softmax: a generalization of sigmoid

- For a vector *z* of dimensionality *k*, the softmax is:

$$\text{softmax}(z) = \left[ \frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

- Example:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \le i \le k$$

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

# Two-Layer Network with scalar output

Output layer
(σ node)

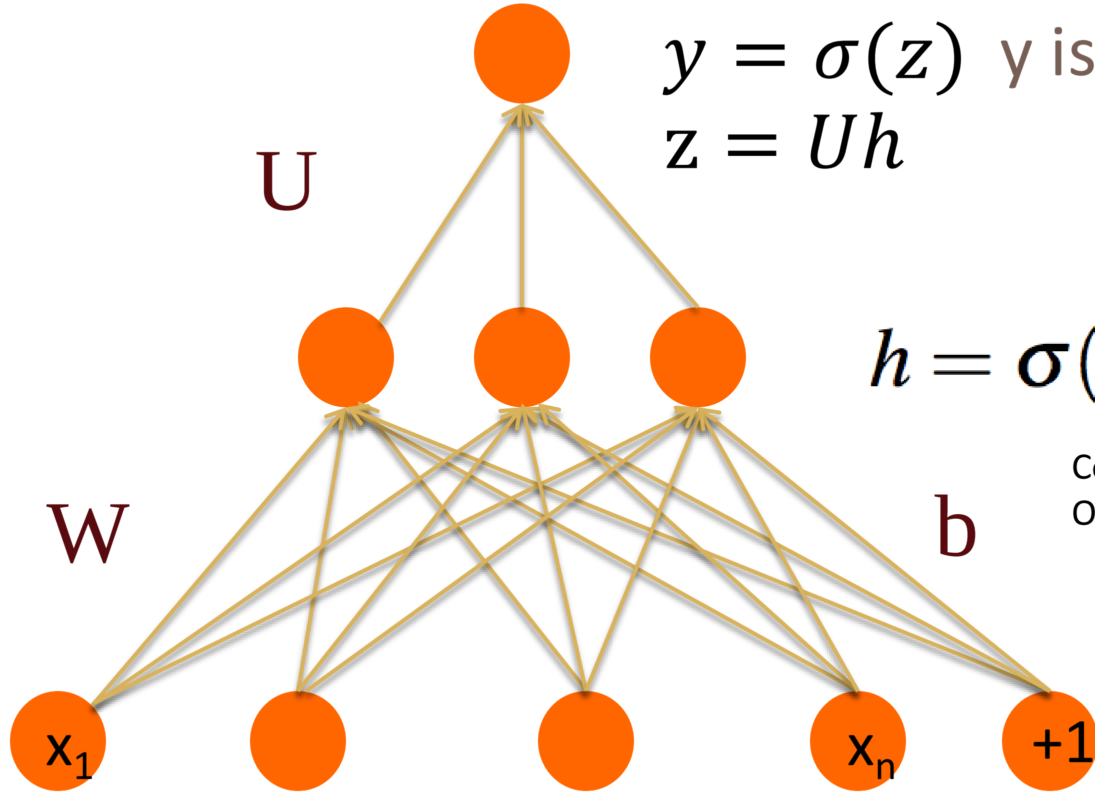
$y = \sigma(z)$ y is a scalar
$z = Uh$

hidden units
(σ node)

$h = \sigma(Wx + b)$

Could be ReLU
Or tanh

Input layer
(vector)



# Two-Layer Network with scalar output

Output layer
(σ node)

$y = \sigma(z)$ y is a scalar

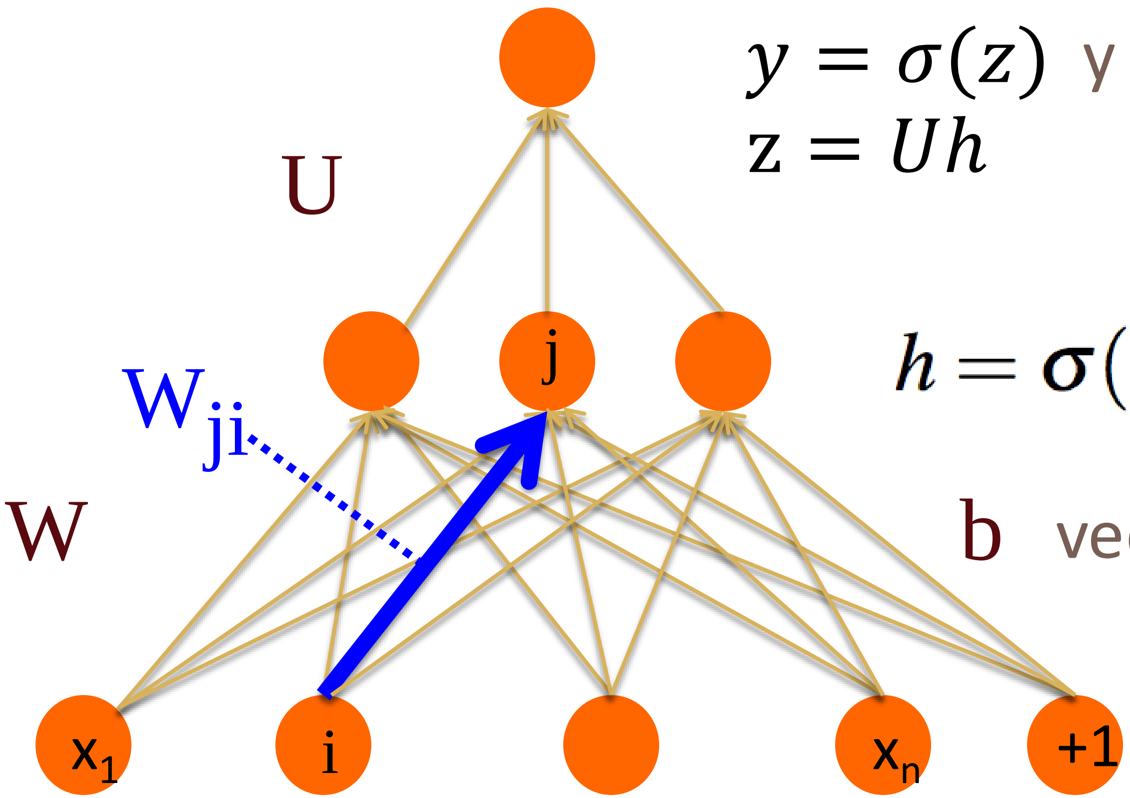
$z = Uh$

hidden units
(σ node)

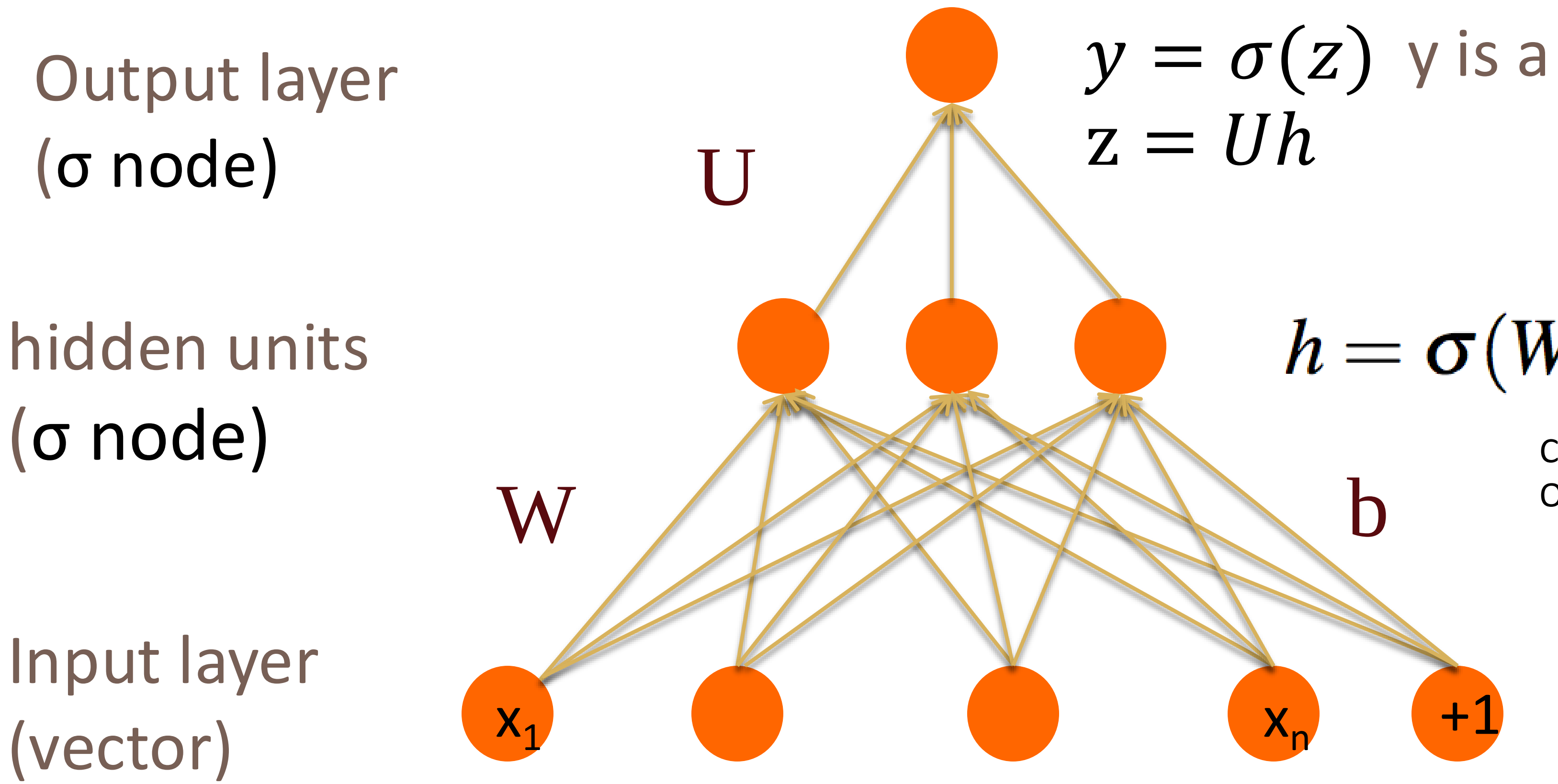
$h = \sigma(Wx + b)$

b vector

Input layer
(vector)



# Two-Layer Network with scalar output




$y = \sigma(z)$ y is a scalar

$z = Uh$

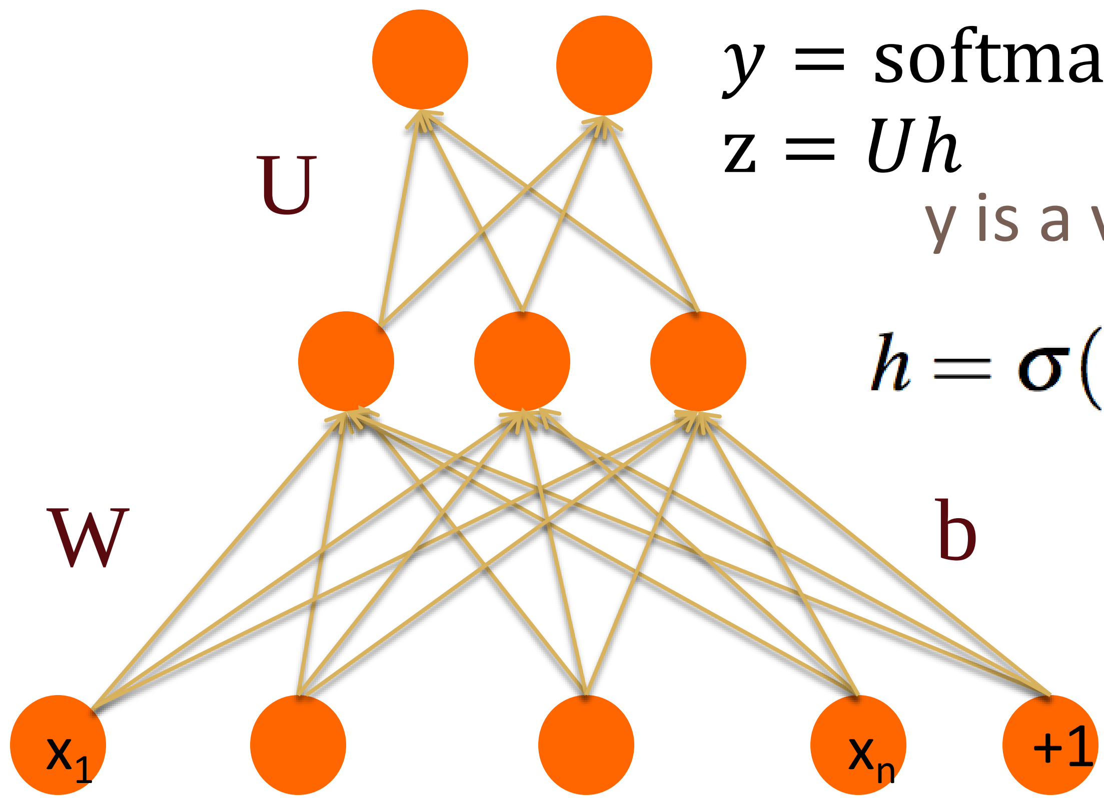
$h = \sigma(Wx + b)$

# Two-Layer Network with softmax output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)




$$y = \text{softmax}(z)$$

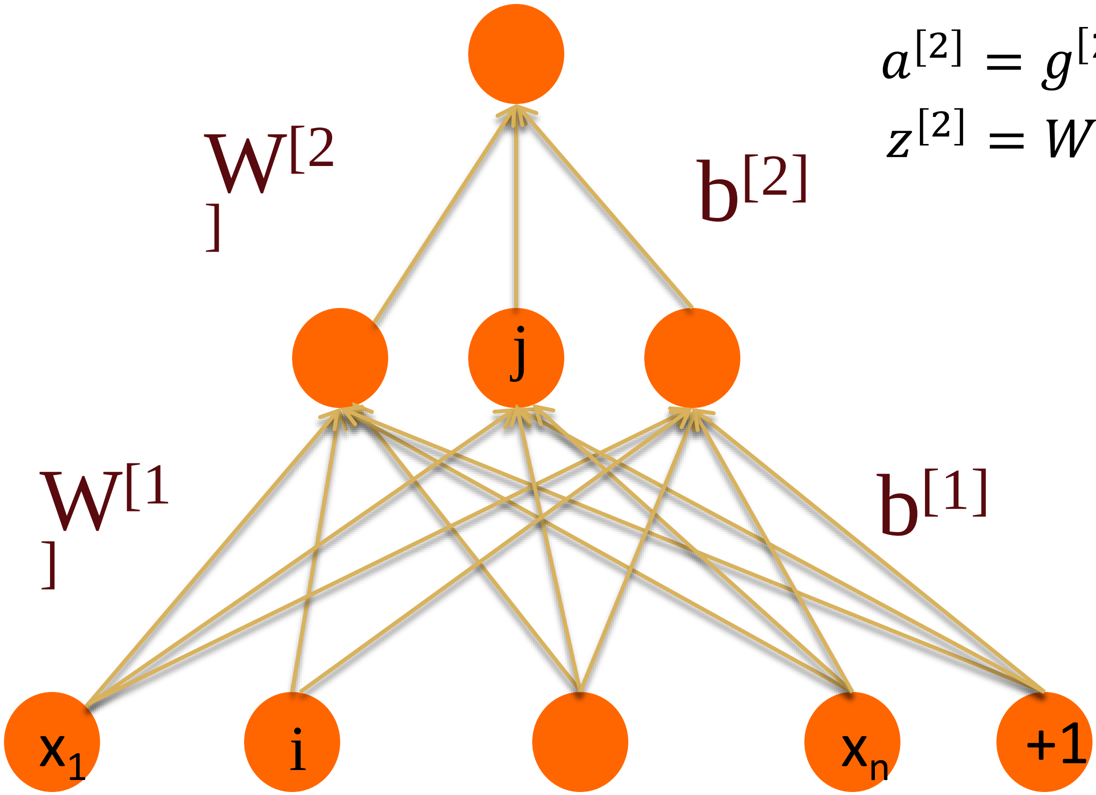
$$z = Uh$$

y is a vector

$$h = \sigma(Wx + b)$$

Could be ReLU
Or tanh

# Multi-layer Notation




$y = a^{[2]}$

$a^{[2]} = g^{[2]}(z^{[2]})$ sigmoid or softmax

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

$a^{[1]} = g^{[1]}(z^{[1]})$ ReLU

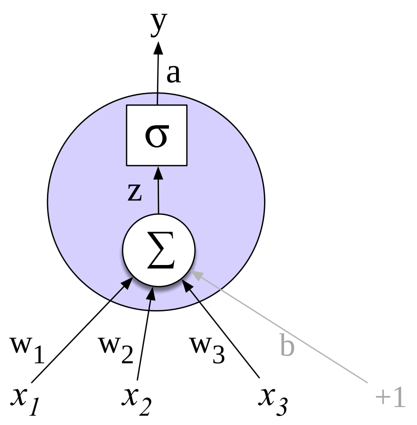
$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$

$a^{[0]}$

# Multi Layer Notation



$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$




**for** $i$ **in** 1..n

$$
\begin{aligned}
z^{[i]} &= W^{[i]} \, a^{[i-1]} + b^{[i]} \\
a^{[i]} &= g^{[i]}(z^{[i]})
\end{aligned}
$$

$$\hat{y} = a^{[n]}$$

# Replacing the bias unit

- Let's switch to a notation without the bias unit
- Just a notational change

1. Add a dummy node $a_0$=1 to each layer
2. Its weight $w_0$ will be the bias
3. So input layer $a^{[0]}_0$=1,
   - And $a^{[1]}_0$=1 , $a^{[2]}_0$=1,…

# Replacing the bias unit

- Instead of:

$$x = x_1, x_2, \ldots, x_{n0}$$

$$h = \sigma(Wx + b)$$

$$h_j = \sigma\left(\sum_{i=1}^{n_0} W_{ji} x_i + b_j\right)$$

We'll do this:

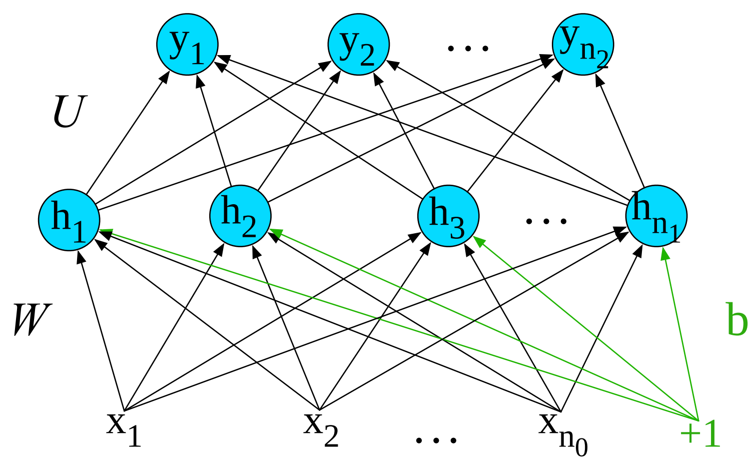
$$x = x_0, x_1, x_2, \ldots, x_{n0}$$

$$h = \sigma(Wx)$$

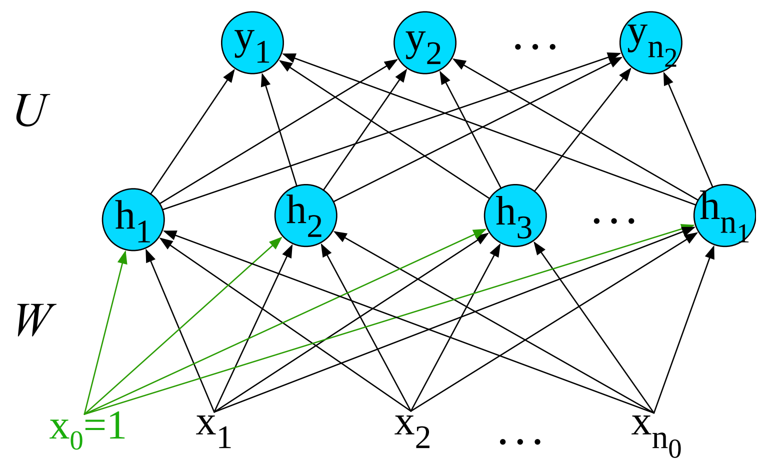
$$\sigma\left(\sum_{i=0}^{n_0} W_{ji} x_i\right)$$

# Replacing the bias unit

Instead of:




We'll do this:



# Lecture outline



- Neural units
- The XOR problem
- Feed-Forward Neural Networks
- Training Neural Nets

# Loss function



- Binary classiffication with sigmoid function at the output layer
  - Cross entropy loss (same as logistic regression)

$$L_{CE}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y} + (1-y)\log(1-\hat{y})]$$

# Loss function



- Multinomial classification with softmax function

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

- Representing $y$ as **one-hot vector**, where true class is $i$

$$y_i = 1 \text{ and } y_j = 0 \ \forall\, j \neq i$$

- Loss function becomes

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i = -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

# Computing the Gradient

- Calculate partial derivative of the loss function with respect to each parameter
- In neural networks, computing gradients for weights in layers is complicated!
- Solution: **error backpropagation**, or **backprop** (Rumelhart et al., 1986) .

# Computation graphs

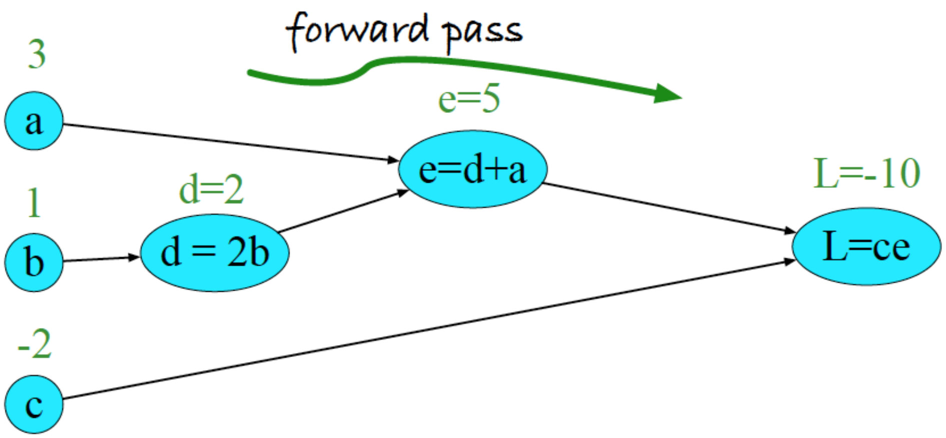


- **Backpropagation** is the same as **backward differentiation**
- Backward differentiation depends on **computation graphs**

# Computation graphs



- The computation is broken down into separate operations, each of which is modeled as a node in a graph
- Consider: $L(a,b,c) = c(a + 2b)$
  - series of computation
    - $d = 2 * b$
    - $e = a + d$
    - $L = c * e$



# Backward differentiation on compution graphs



- We would like to compute $\frac{\partial L}{\partial a} \; \frac{\partial L}{\partial b} \; \frac{\partial L}{\partial c}$
- **Chain rule**

$$\frac{du}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

  - We can apply the chain rule to more than two functions

On compution graph

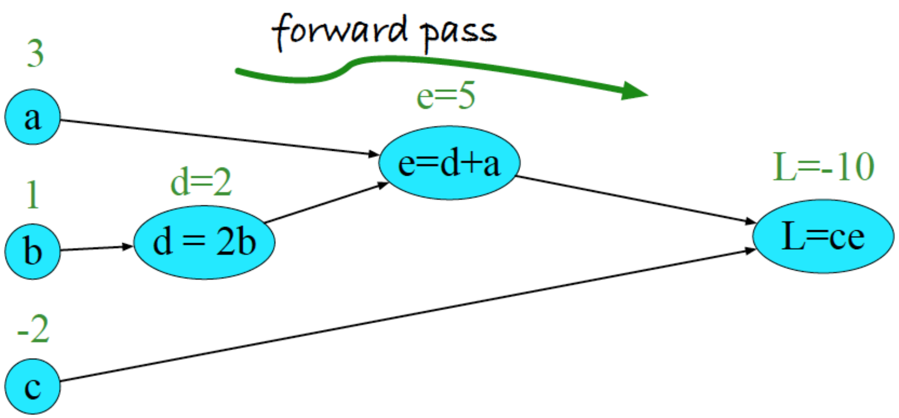
$$L = ce$$

So:

$$\frac{\partial L}{\partial c} = e$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$



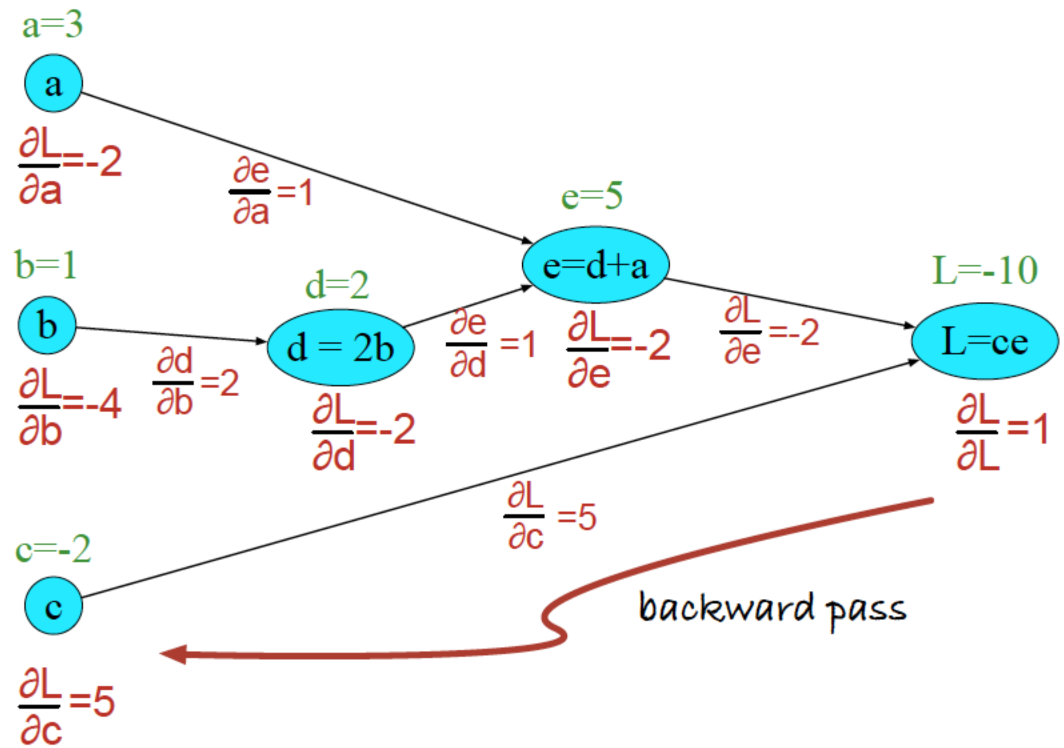
# Backward differentiation on compution graphs



$$L = ce \;:\; \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

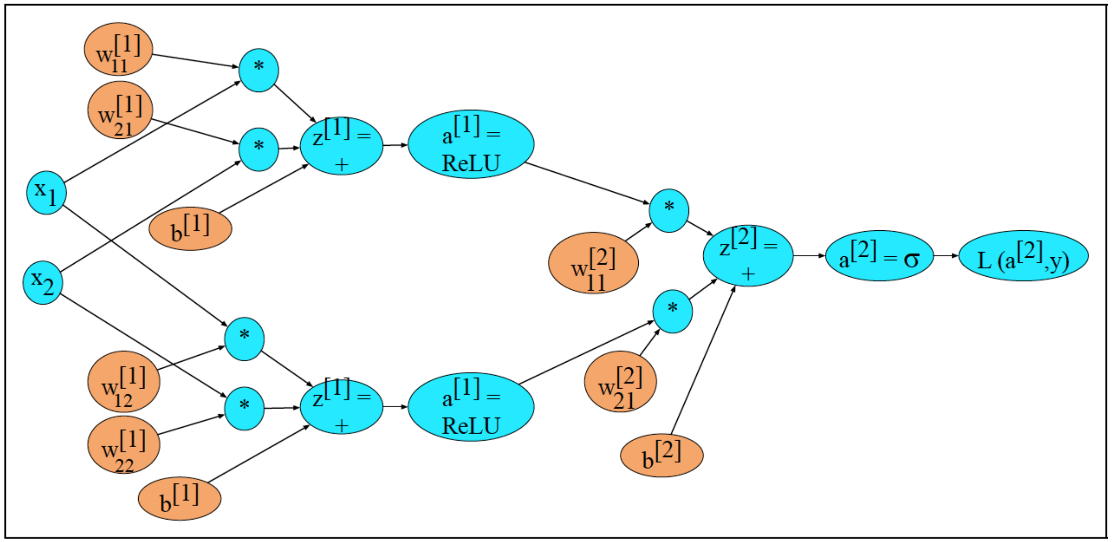
$$e = a + d \;:\; \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b \;:\; \frac{\partial d}{\partial b} = 2$$



# Backward differentiation for a neural network






- Derivatives of activation functions
  - Sigmoid: $\frac{d\sigma}{dz} = \sigma(z)(1 - \sigma(z))$
  - Tanh: $\frac{d\tanh(z)}{d(z)} = 1 - \tanh^2(z)$
  - ReLU: $\frac{d\text{ReLU}(z)}{d(z)} = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$

# Training neural networks



- We apply gradient-based optimization algorithms
  - SGD
  - Adam
  - …
- Aspects we need to care when training
  - Weight initialization
  - Regularization: dropout,…
  - Hyperparameter tuning
    - Learning rate
    - Mini-batch size
    - Model architecture
- Some libraries that support differentiation on compution graphs: Pytorch, Tensorflow, Jax