Lecture slides for this course
have been prepared by Dr. Le Minh Huy,
EEE, Phenikaa University

# Deep Learning

## Chapter 2 Building Neural Network from Scratch

Dr. Van-Toi NGUYEN

*EEE, Phenikaa University*

# Chapter 2: Building Neural Network from Scratch

1. Shallow neural network
2. Deep neural network
3. Building neural network: step-by-step (modulation)
4. Regularization
5. Dropout
6. Batch Normalization
7. Optimizers
8. Hyper-parameters
9. Practice

# Previous Lecture Overview

**Chapter 1: Course Infor & Programming review - week 1**

1. Course introduction and grades
2. History of Deep learning
3. Deep learning applications

**Chapter 2: Building Neural Network from Scratch – week 2-7**

1. Shallow neural network - week 2
2. Deep neural network - week 3
3. Building neural network: step-by-step (modulation) - week 3
4. Regularization - week 4
5. Dropout - week 4
6. Batch Normalization - week 5
7. Optimizers - week 6
8. Hyper-parameters - week 7
9. Practice- week

**Midterm**

**Chapter 3: Convolutional Neural Network - week 8-10**

1. Convolutional operator
2. History of CNN
3. Deep Convolutional Models
4. Layers in CNN
5. Applications of CNN
6. Practice

**Midterm summary**

**Chapter 4: TensorFlow Library- week 11-13**

1. Introduction to TensorFlow
2. Building a deep neural network with TensorFlow
3. Applications
4. Practice

**Chapter 5: Recurrent Neural Network week 14-15**

1. Unfolding Computational Graphs
2. Building a Recurrent Neural Networks
3. Long Short-Term Memory
4. Vision with Language Processing
5. Application of RNN
6. Practice

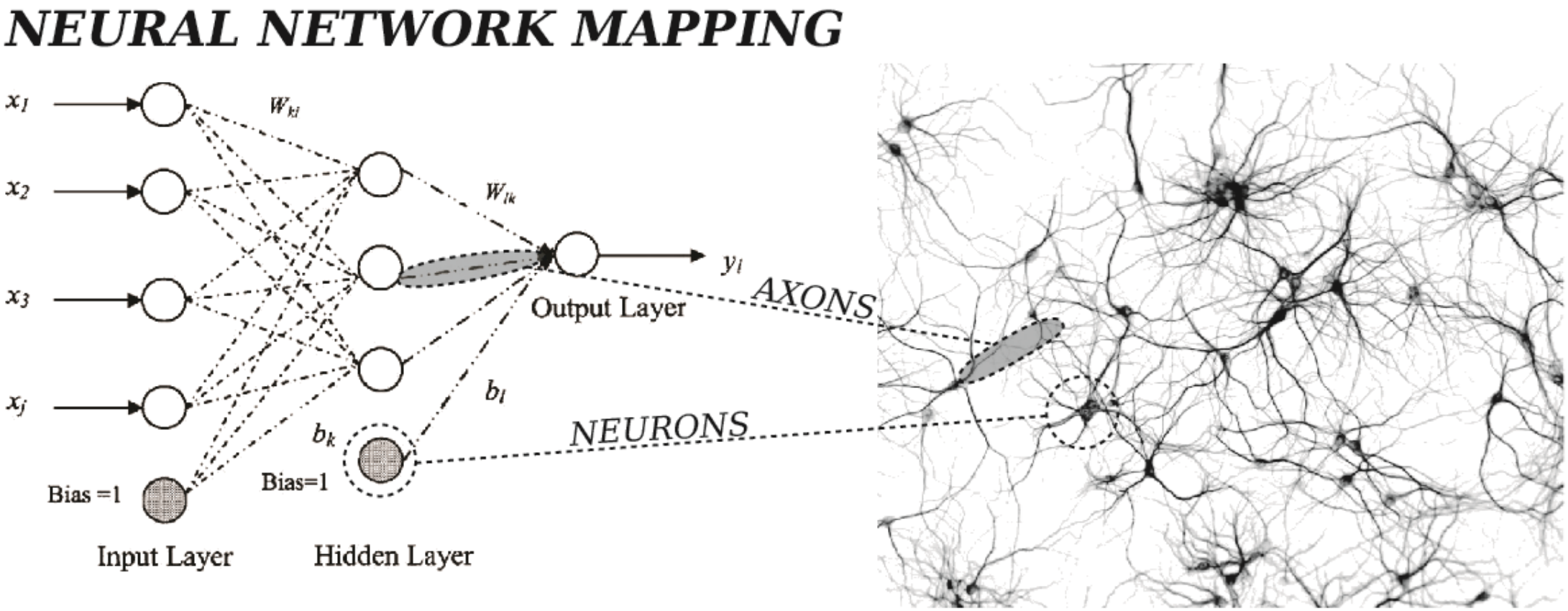
# Previous Lecture Overview

## Basic of Neural Network

- The Perceptron and its Learning Rule (Frank Rosenblatt, 1957)
- Adaptive Linear Neuron and Delta Rule (Widrow & Hoff, 1960)
- Logistic Regression and Gradient Descent

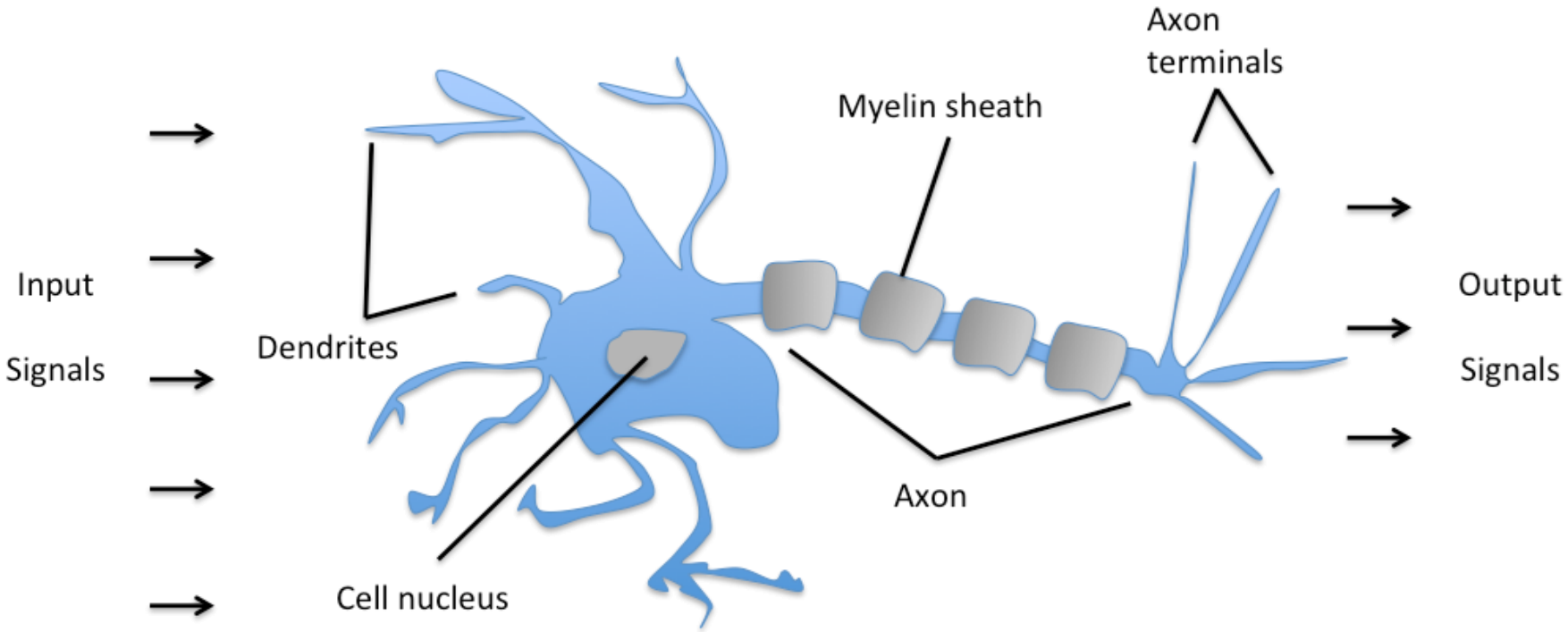
# Previous Lecture Overview

## Biologically inspired (akin to the neurons in a brain)



# Previous Lecture Overview

## Artificial Neurons and the McCulloch-Pitts Model (1943)

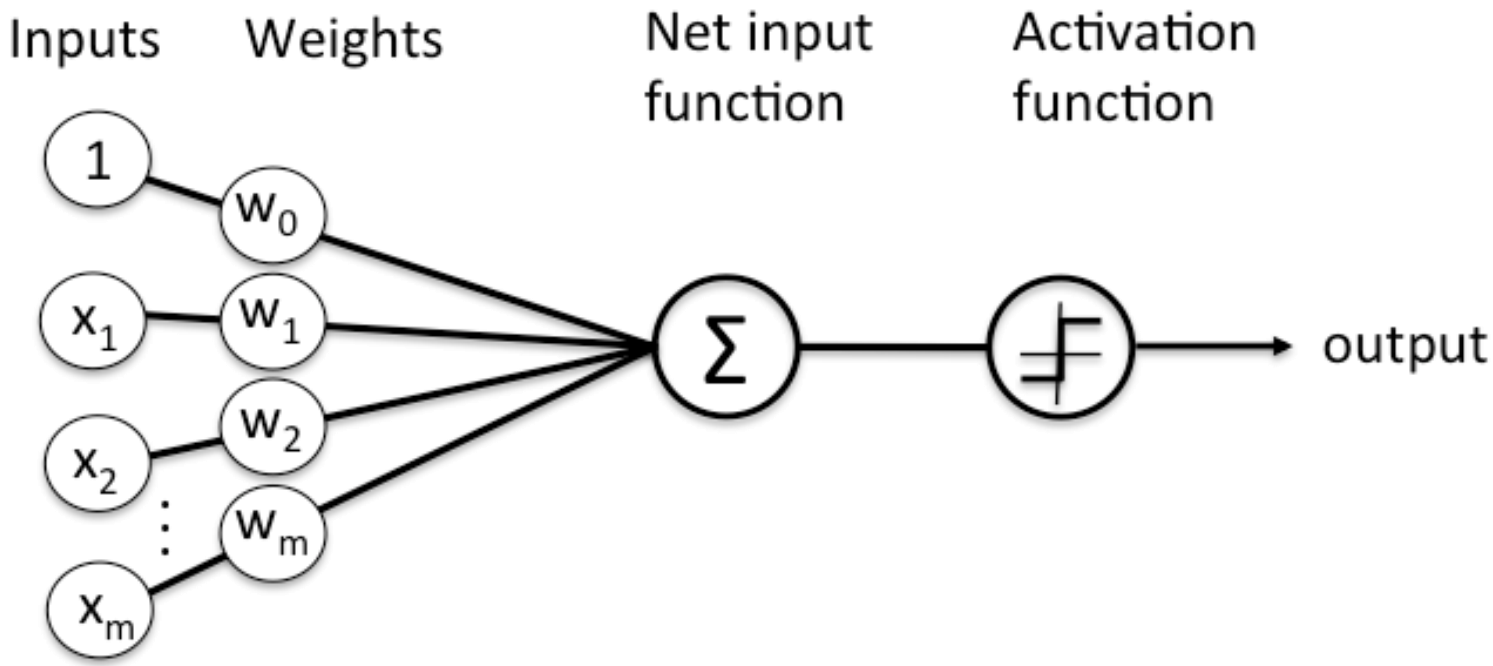



**Schematic of a biological neuron.**

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

# Previous Lecture Overview

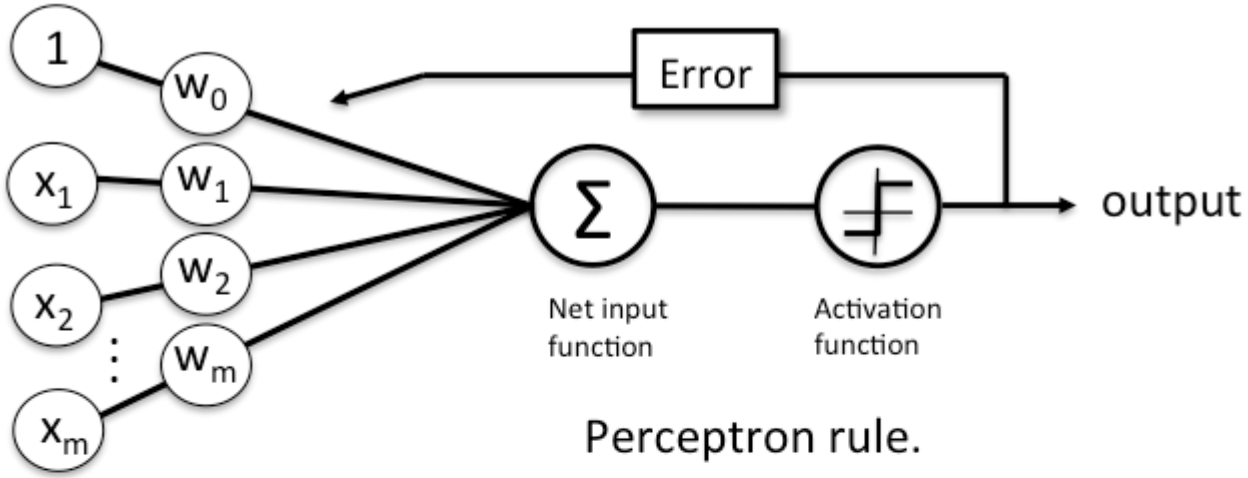
## Frank Rosenblatt's Perceptron (1957)



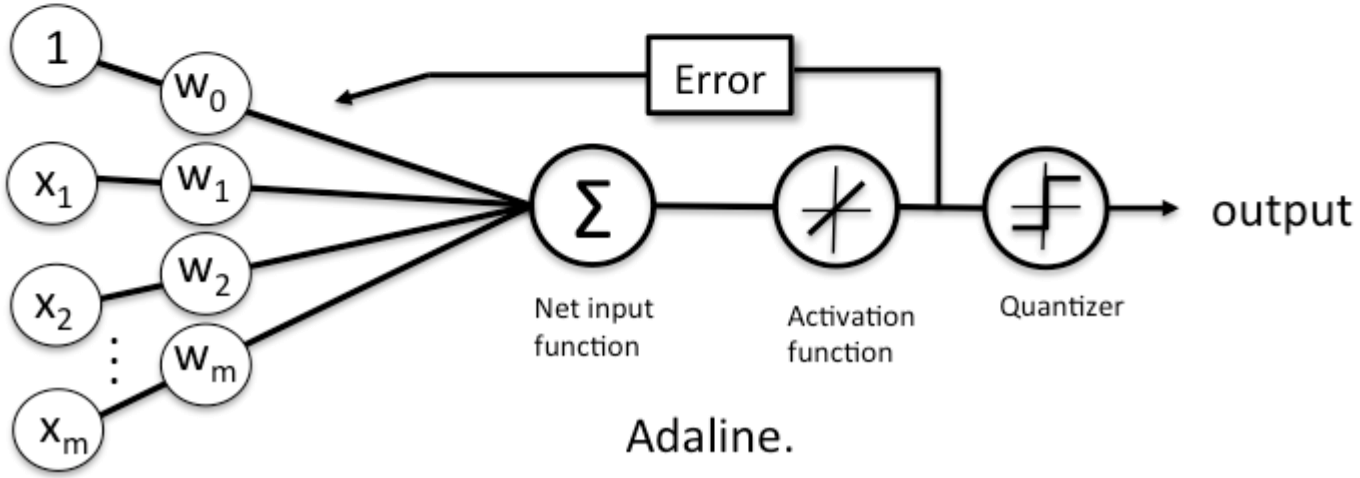

**Schematic of Rosenblatt's perceptron.**

# Previous Lecture Overview

## Adaptive Linear Neurons and the Delta Rule (1960)




Perceptron rule.

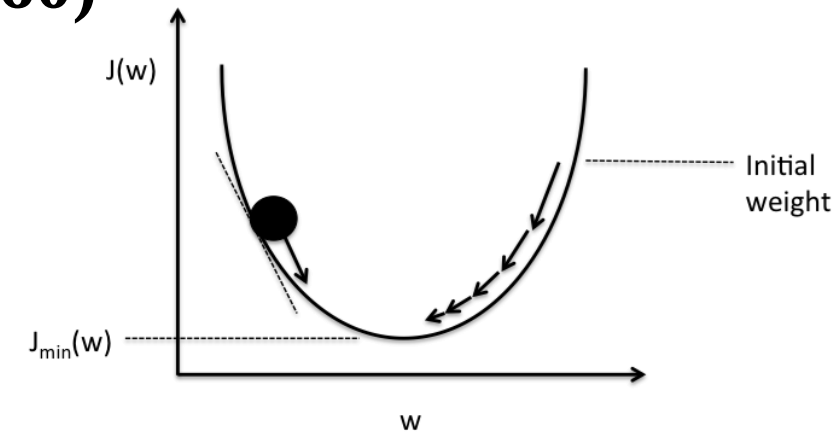



Adaline.

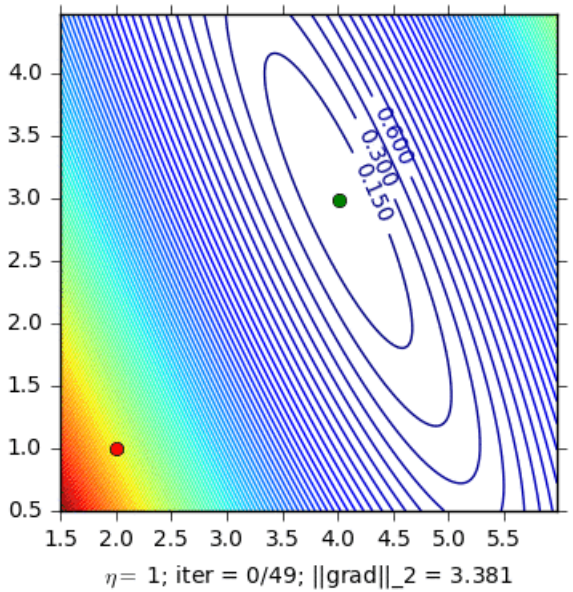
PHENIKAA UNIVERSITY

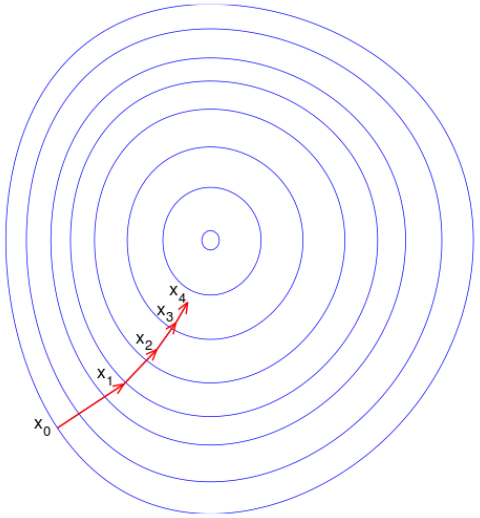
# Adaptive Linear Neurons and the Delta Rule (1960)

- **Gradient Descent**
  - A first-order iterative optimization algorithm for finding the minimum of a function
  - Take steps proportional to the **negative of the gradient** of the function at the current point




Schematic of gradient descent.



# Previous Lecture Overview

## Adaptive Linear Neurons and the Delta Rule (1960)

- Cost function: sum of squared errors (SSE)
  - $J(w) = \frac{1}{2}\sum_i (y'^{(i)} - y^{(i)})^2$
- To minimize SSE, we can use "gradient descent"
- A step in the opposite direction of gradient

$$\Delta w = -\alpha \nabla J(w)$$

where $\alpha$ is the learning rate, $0 < \alpha < 1$

- Thus, we need to compute the partial derivative of the cost function for each weight in the weight vector,

$$\Delta w_j = -\alpha \frac{\partial J}{\partial w_j}$$

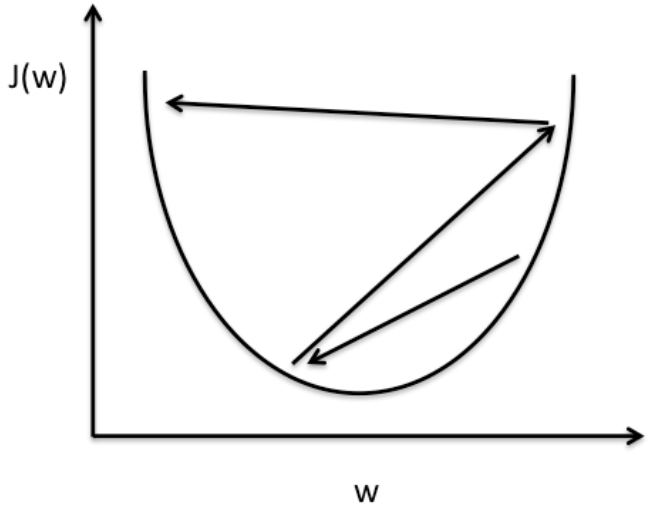
# Previous Lecture Overview

## Adaptive Linear Neurons and the Delta Rule (1960)

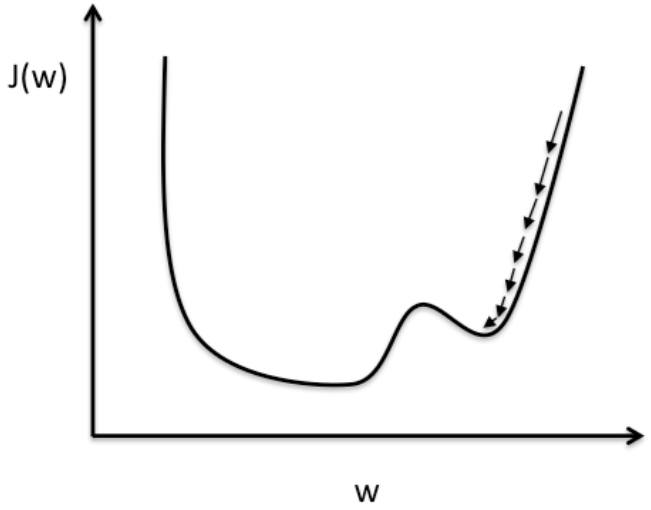
- A step in gradient descent:
  - $\Delta w_j = -\alpha \frac{\partial J}{\partial w_j} = -\alpha \sum_i \left(y'^{(i)} - y^{(i)}\right)\left(-x_j^{(i)}\right) = \alpha \sum_i (y'^{(i)} - y^{(i)}) x_j^{(i)}$
- Update weight vector:
  - $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$
- Differences with the perceptron rule
  - The output $y^{(i)}$ is a real number, not a class label as in perceptron learning rule.
  - Weight update is based on "all samples in the training set" (Batch GD)

## Adaptive Linear Neurons and the Delta Rule (1960)

- If the learning rate is TOO LARGE, gradient descent will overshoot the minima and diverge.
- If the learning rate is too small, gradient descent will require too many epochs to converge and can become trapped in local minima more easily.




Large learning rate: Overshooting.




Small learning rate: Many iterations until convergence and trapping in local minima.

## Adaptive Linear Neurons and the Delta Rule (1960)

- If features are scaled on the same scale, gradient descent converges faster and prevents weights from becoming too small (weight decay).

- Common way for feature scaling

$$x_{j,std} = \frac{x_j - \mu_j}{\sigma_j}$$

where $\mu_j$ is the sample mean of the feature $x_j$ and $\sigma_j$ the standard deviation.

- After standardization, the features will have unit variance and centered around mean zero.

# Previous Lecture Overview

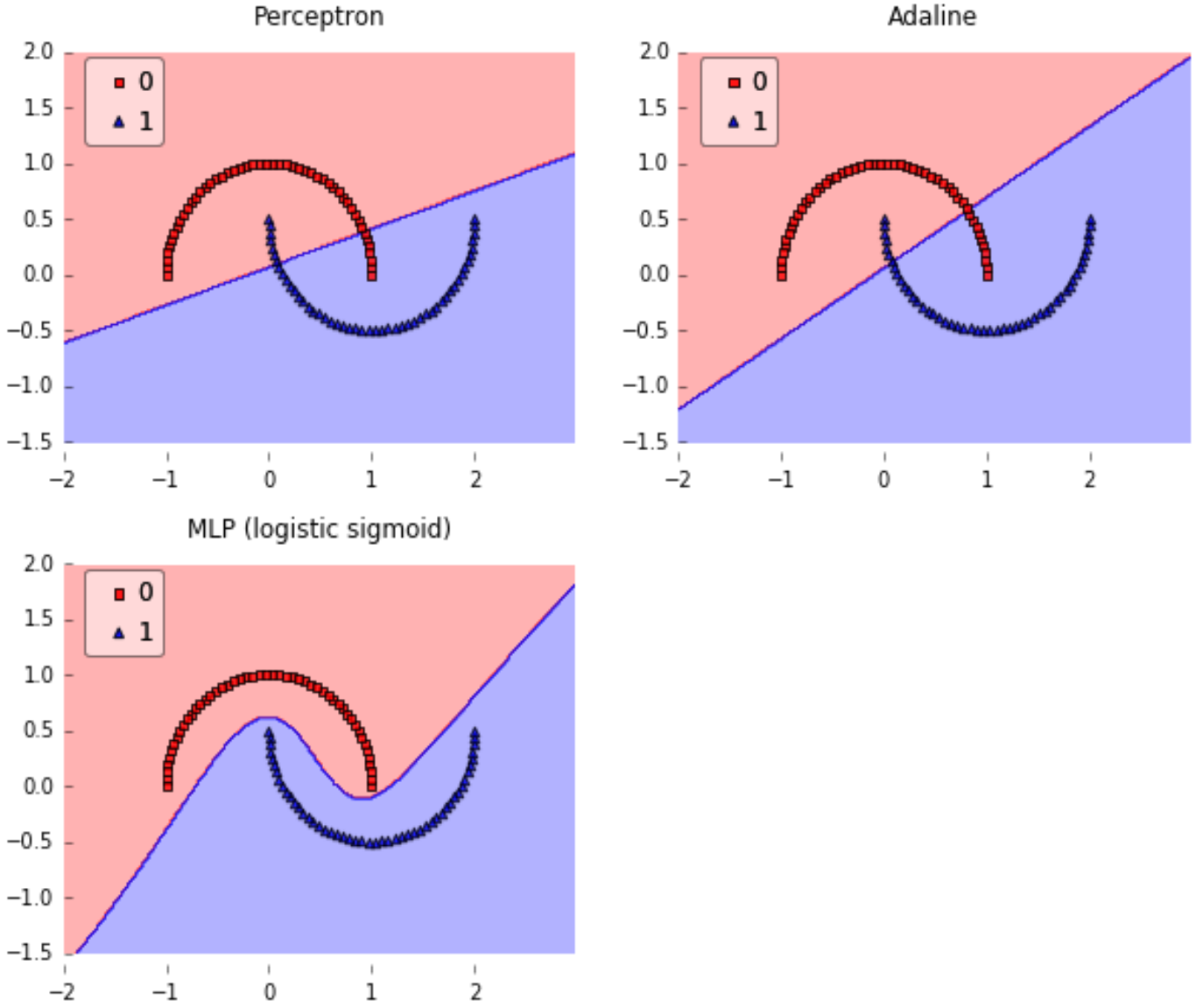
## Adaptive Linear Neurons and the Delta Rule (1960)

- Batch Gradient Descent (BGD)
  - Cost function is minimized based on the complete training dataset (all samples)
- Stochastic Gradient Descent (SGD)
  - Weights are incrementally updated after each individual training sample
  - Converges faster than BGD since weights are updated immediately after each training sample
  - Computationally more efficient, especially for large datasets
- Mini-batch Gradient Descent (MGD)
  - Compromise between BGD and SGD, dataset is divided into mini-batches
  - Smoother convergence than SGD

# Previous Lecture Overview

## Logistic Regression

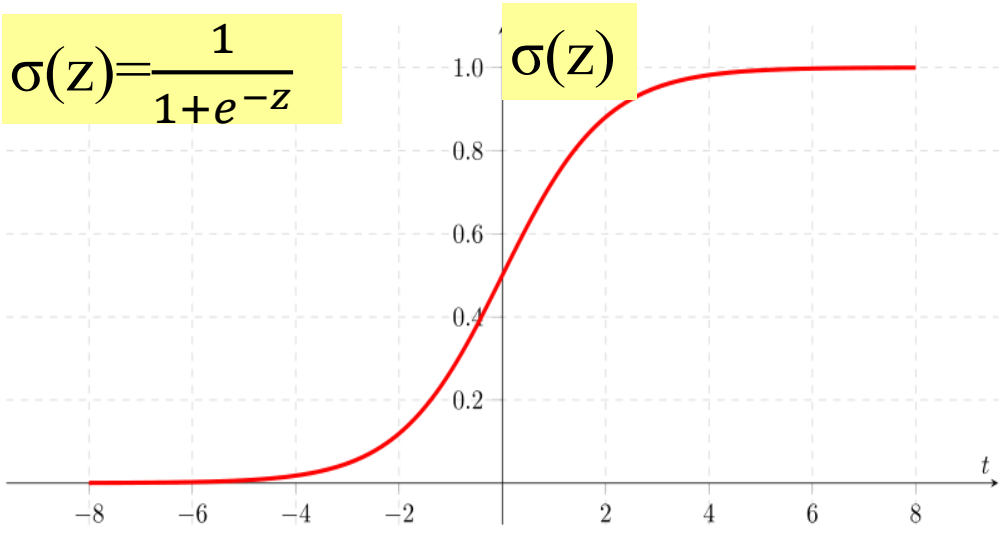
Perceptron vs. Adaline vs. Multi-Layer Perceptrons (Logistic Regression)



## Logistic Regression

- Definition:
  - Given input $x \in \mathcal{R}^{n_x}$, calculate the probability $\hat{y} = P(y = 1|x)$, $0 \leq \hat{y} \leq 1$.
- Parameters:
  - Weights: $w \in \mathcal{R}^{n_x}$
  - Bias: $b \in \mathcal{R}$
- Output:
  - $\hat{y} = \sigma(z) = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the **sigmoid activation function**

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




If z is large positive number, σ(z) → 1

If z is small negative number, σ(z) → 0

# Previous Lecture Overview

## Logistic Regression

- Cost function is the average of all cross-entropy losses

$$\mathcal{J}(w, b)_m = \frac{1}{m}\sum_{i=1_m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$
$$= -\frac{1}{m}\sum_{i=1}^{} \left[y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)\right]$$

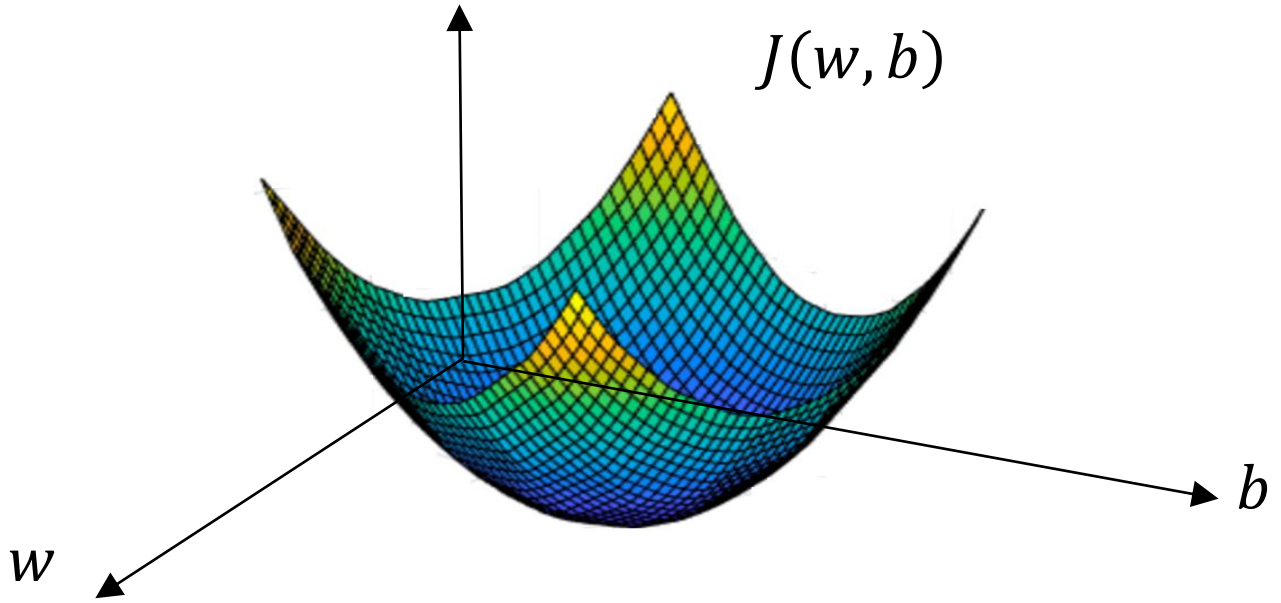
- Goal:
  - Find vectors **w** and **b** that minimize the cost function (total loss)
- Logistic regression can be viewed as a small neural network!

# Previous Lecture Overview

## Logistic Regression

- $\hat{y}^{(i)} = \sigma\left(z^{(i)}\right) = \sigma\left(w^T x^{(i)} + b\right)$, where $\sigma\left(z^{(i)}\right) = \frac{1}{1+e^{-z^{(i)}}}$
- $\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log \hat{y}^{(i)} + \left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right]$
- Find w, b that minimize $\mathcal{J}(w,b)$

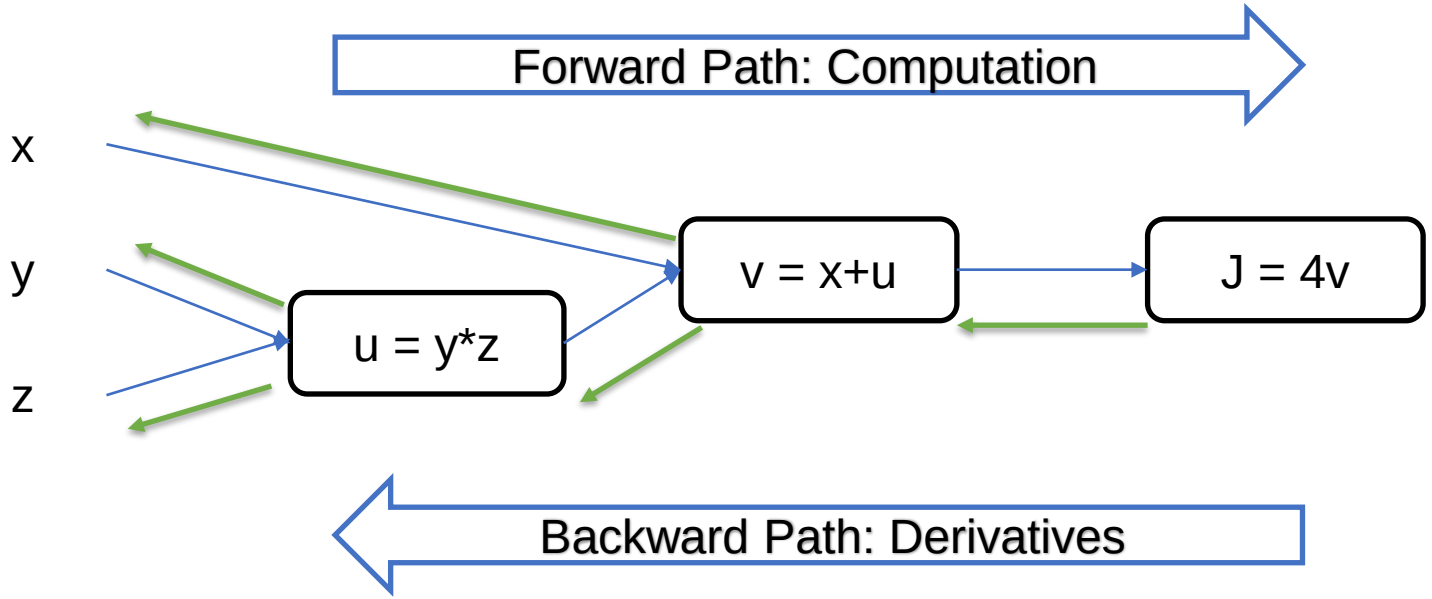
## Convergence



# Previous Lecture Overview

## Logistic Regression Computation Graph

- A graph that depicts all the computations required for a function in a forward path
- For example: J(x, y, z) = 4(x + yz)



# Previous Lecture Overview

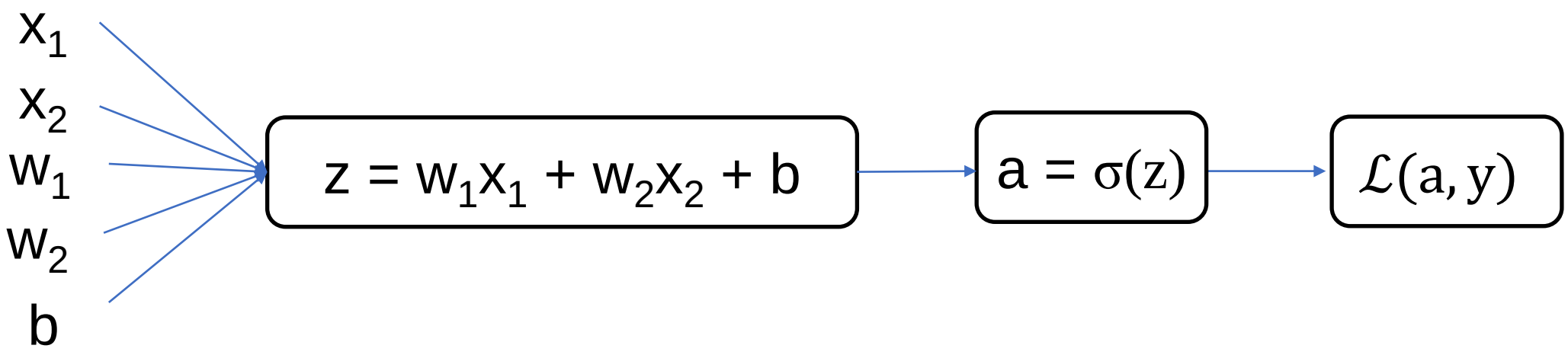
## Logistic Regression

$$z = w^T x + b$$

$$\hat{y} = a = \sigma(z)$$

$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$

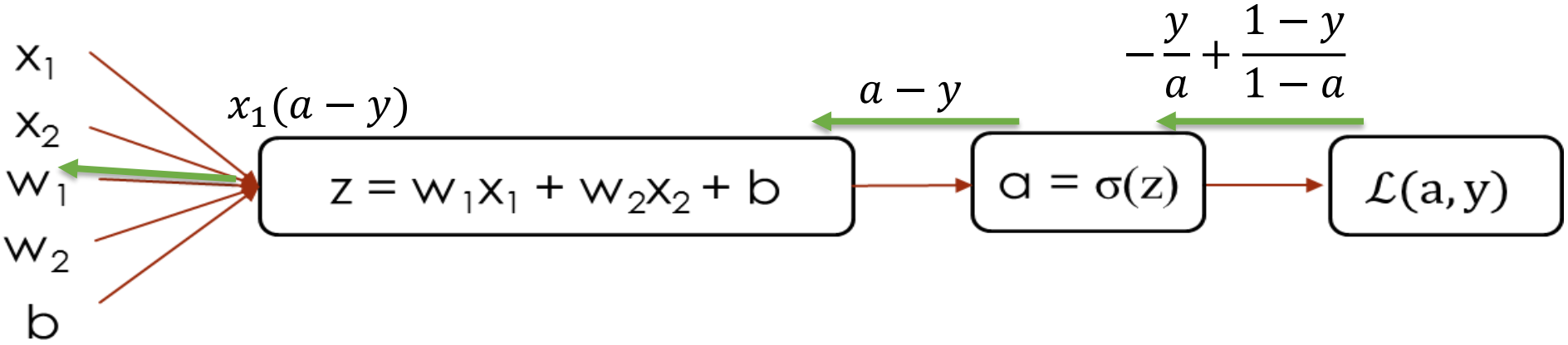
## Computation Graph



# Previous Lecture Overview

## Logistic Regression

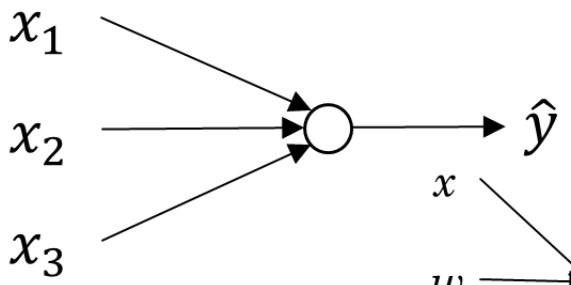
## Computation Graph

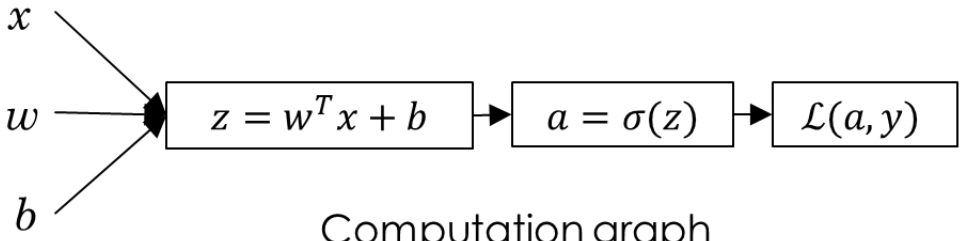



- $\frac{\delta L(a,y)}{\delta a} = \left(-\frac{y}{a}+\frac{1-y}{1-a}\right)$
- $\frac{\delta L(a,y)}{\delta z} = \frac{\delta L(a,y)}{\delta a}\frac{\delta a}{\delta z} = \left(-\frac{y}{a}+\frac{1-y}{1-a}\right)\left(a(1-a)\right) = a-y$
- $\frac{\delta L(a,y)}{\delta w_1} = \frac{\delta L(a,y)}{\delta a}\frac{\delta a}{\delta z}\frac{\delta z}{\delta w_1} = \left(-\frac{y}{a}+\frac{1-y}{1-a}\right)a(1-a)x_1 = x_1(a-y) = x_1\frac{\delta L(a,y)}{\delta z}$

# What is a Shallow Neural Network?

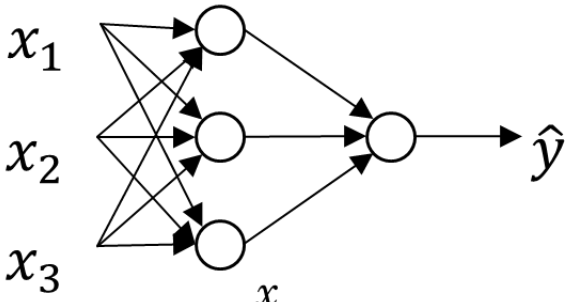
Single neuron model

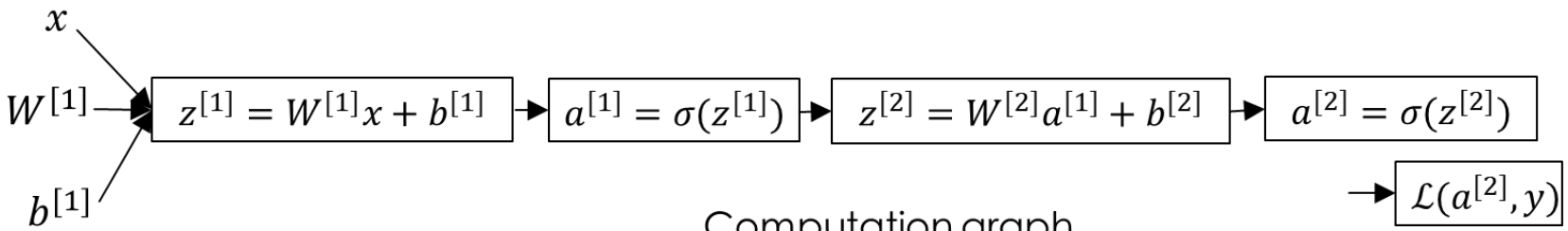



Computation graph

Multiple layer neural model



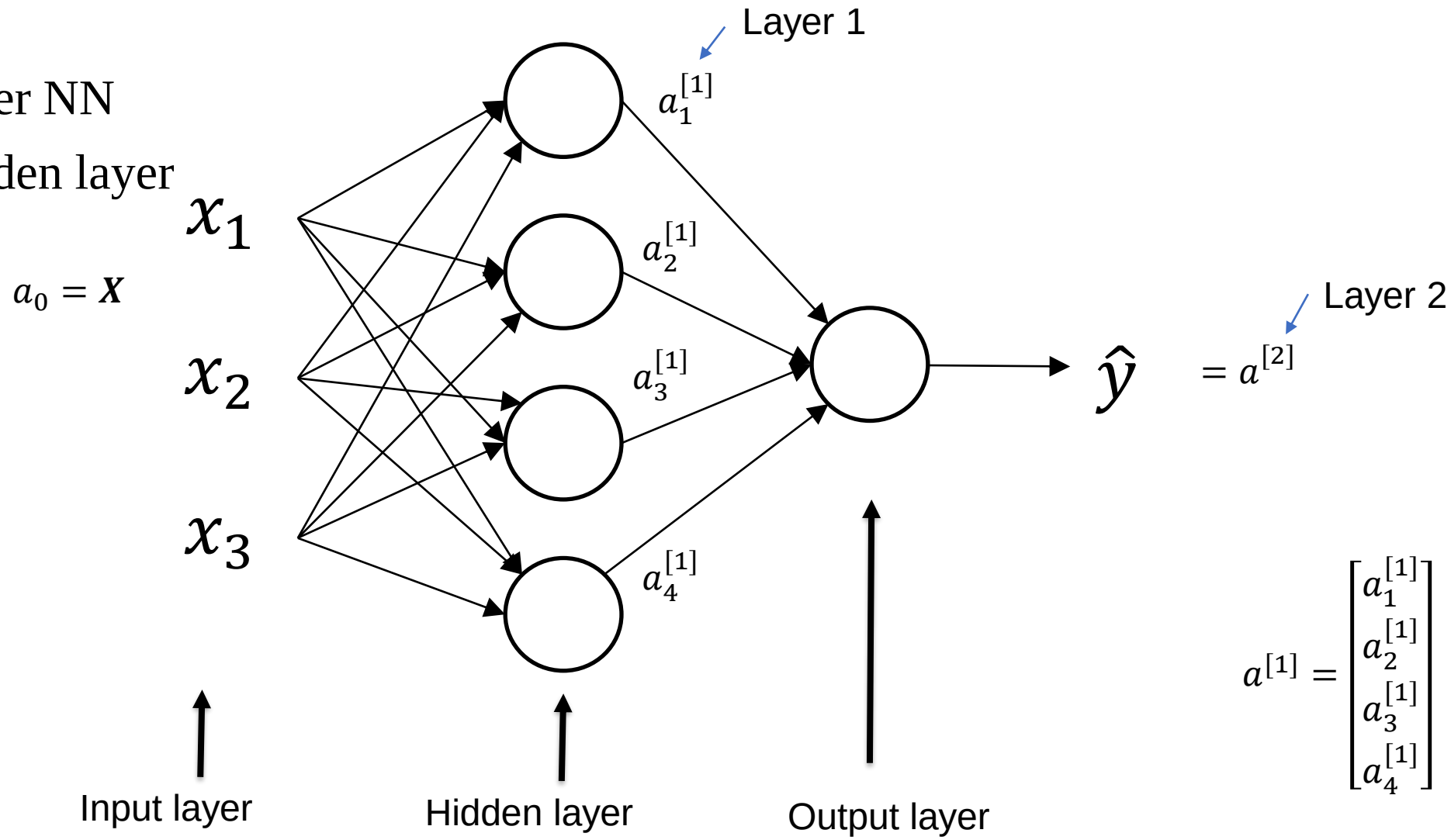




Computation graph

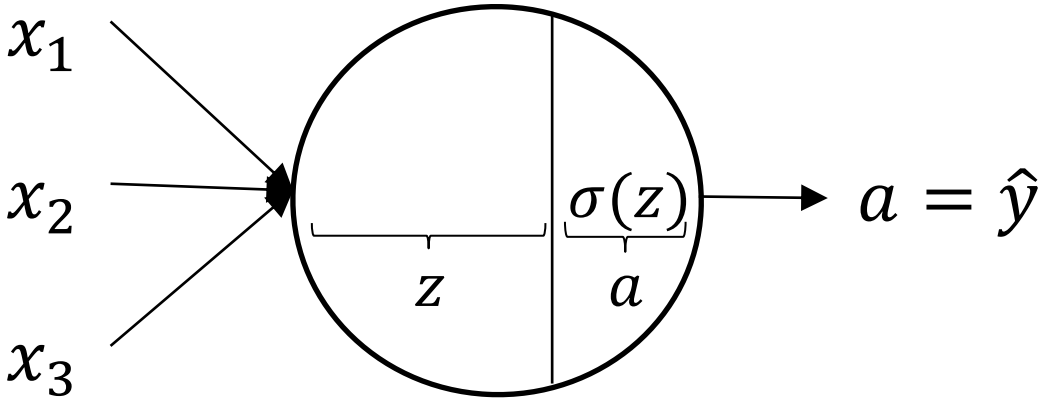
# 1. Shallow neural network

## One hidden layer Neural Network

- 2-layer NN
- 1 hidden layer



## Computing NN's Output

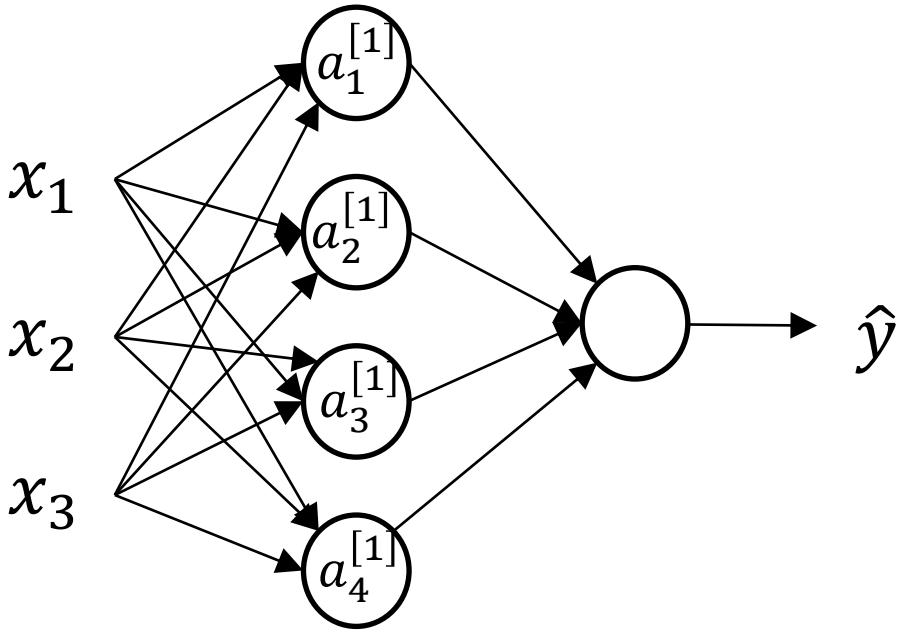



$$z = w^T x + b$$

$$a = \sigma(z)$$

# 1. Shallow neural network

## Computing NN's Output




$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

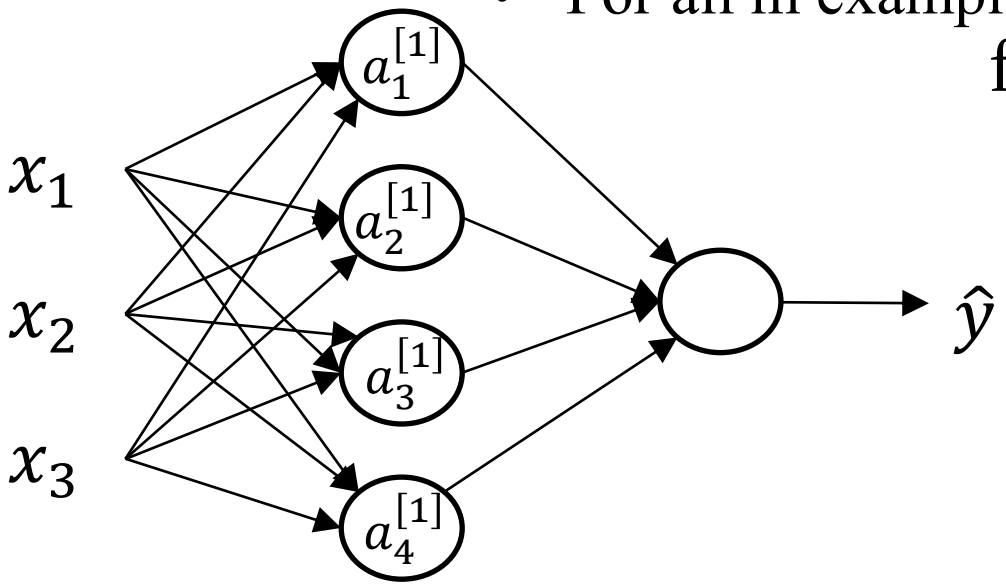
Given input x:

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

$$a^{[1]} = \sigma(z^{[1]})$$

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

$$a^{[2]} = \sigma(z^{[2]})$$




- For all m examples …

for i = 1 to m:

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

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

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

# 1. Shallow neural network

## Vectorizing across multiple examples

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \dots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

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

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

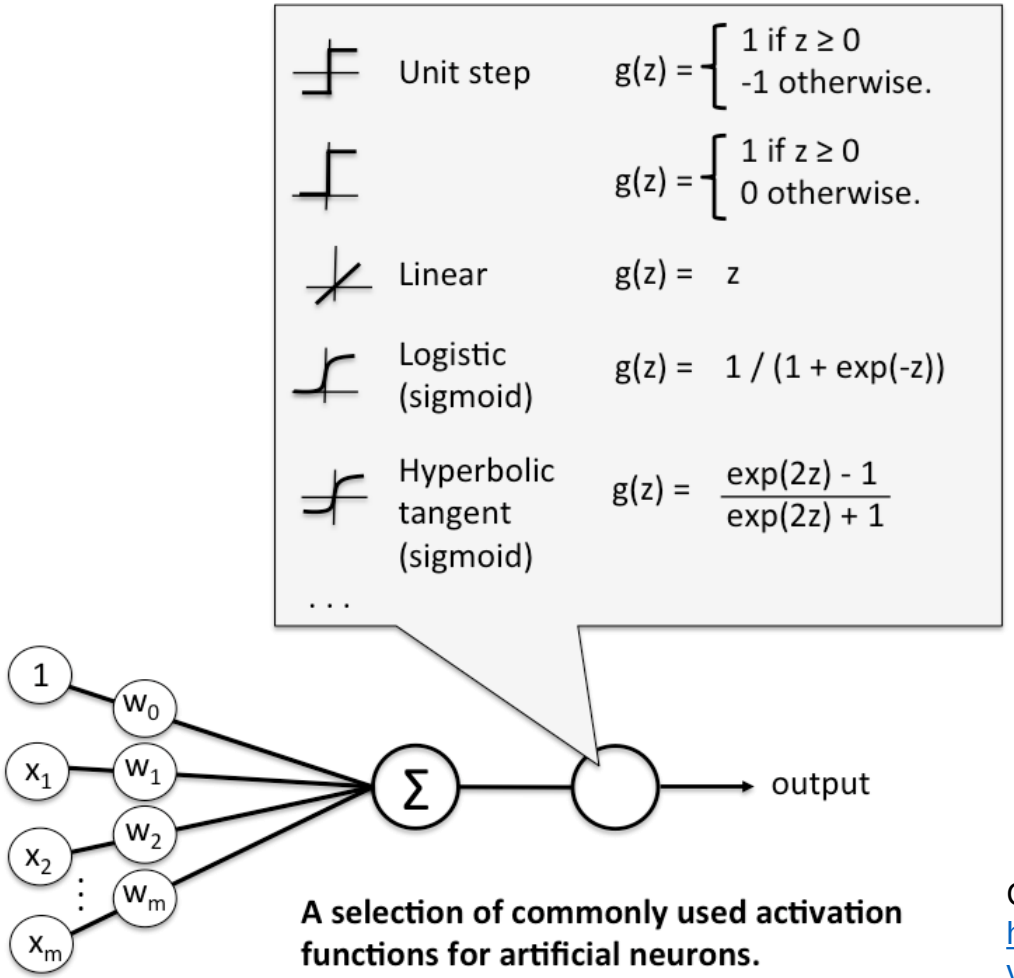
$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

# 1. Shallow neural network

## Activation functions



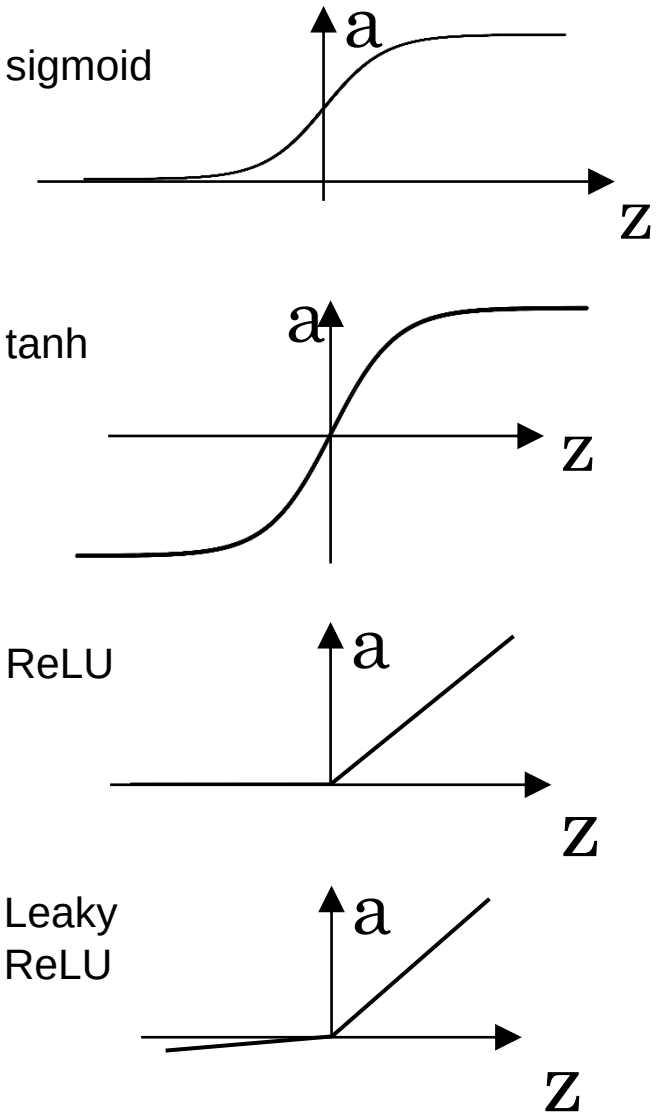

**A selection of commonly used activation functions for artificial neurons.**

Comprehensive List of Activation Functions:
https://stats.stackexchange.com/questions/115258/comprehensive-list-of-activation-functions-in-neural-networks-with-pros-cons

# 1. Shallow neural network

## Activation functions

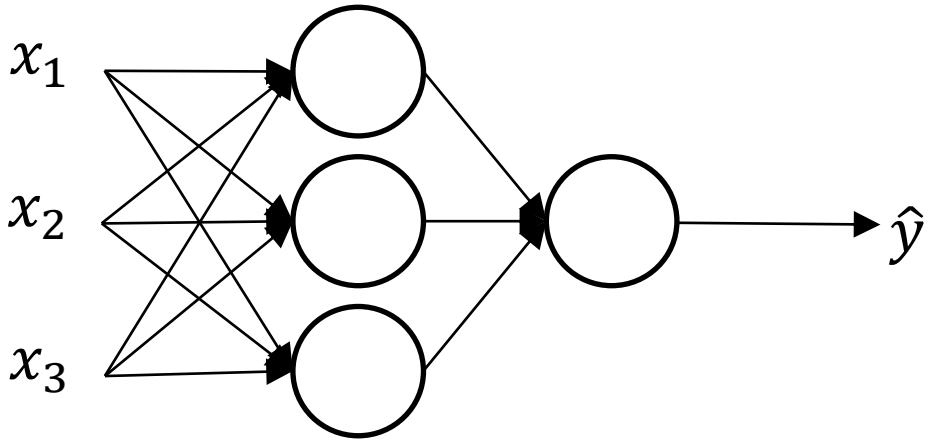
| Activation Function | Formula (g(z)) | Derivative (g'z)) |
|---|---|---|
| sigmoid | $a = \frac{1}{1+e^{-z}}$ | $a(1-a)$ |
| tanh | $a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ | $1-a^2$ |
| ReLU | $\max(0,z)$ | $0 \text{ if } z<0$<br>$1 \text{ if } z \geq 0$ |
| Leaky ReLU | $\max(0.01z,z)$ | $0.01 \text{ if } z<0$<br>$1 \text{ if } z \geq 0$ |



# 1. Shallow neural network

## Why non-linear activation function?

- What not linear?
- Suppose $g^{[1]}$, $g^{[2]}$ are all linear
  - $a^{[1]} = z^{[1]}$
  - $a^{[2]} = z^{[2]}$
  - $a^{[2]} = W^{[2]}a^{[1]}+b^{[2]}$
  - $= W^{[2]}(W^{[1]}X+b^{[1]})+b^{[2]}$
  - $= W^{[2]}W^{[1]}X + W^{[2]}b^{[1]}+b^{[2]}$
  - $= W'X + b'$
- All LINEAR!!!




Given x:

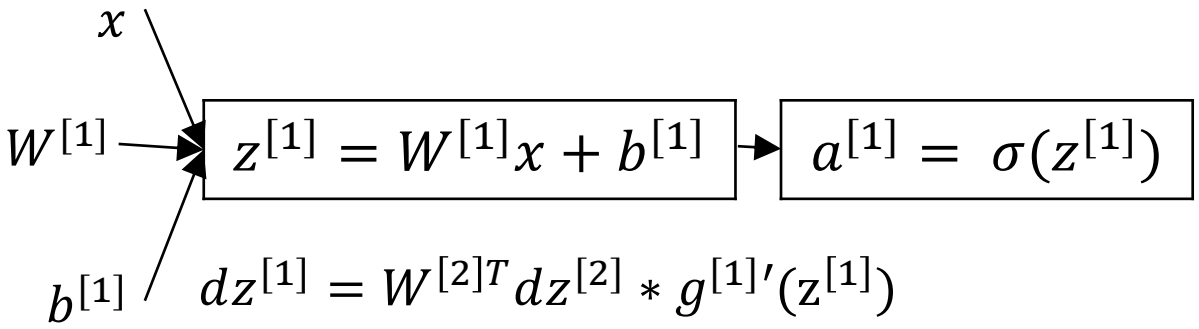
$$z^{[1]} = W^{[1]}x + b^{[1]}$$

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

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

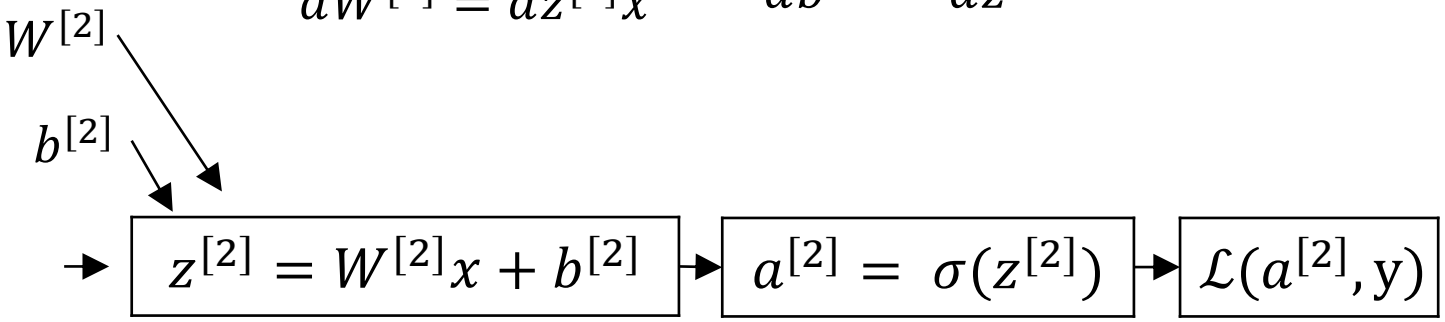
$$a^{[2]} = g^{[2]}(z^{[2]})$$

## Gradient descent for one hidden layer




$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = dz^{[1]} x^T \qquad db^{[1]} = dz^{[1]}$$




$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]^T}$$

$$db^{[2]} = dz^{[2]}$$

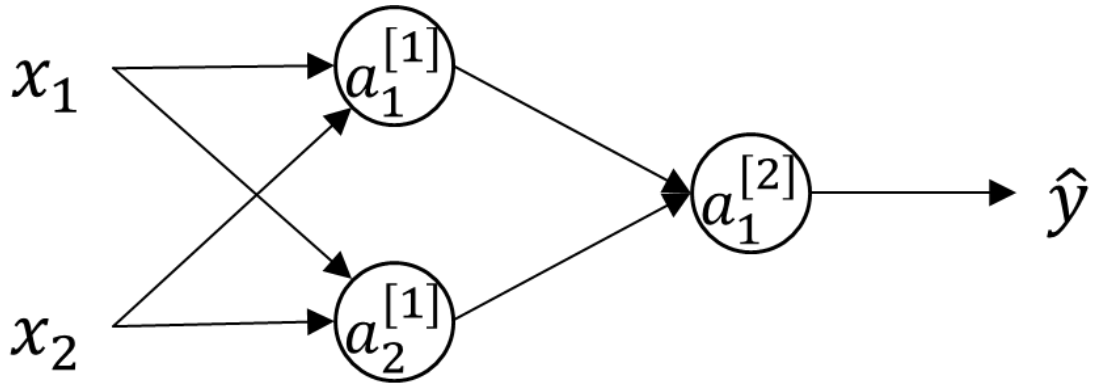
## Vectorizing Gradient Descent

| | |
|---|---|
| $dz^{[2]} = a^{[2]} - y$ | $dZ^{[2]} = A^{[2]} - Y$ |
| $dW^{[2]} = dz^{[2]} a^{[1]^T}$ | $dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$ |
| $db^{[2]} = dz^{[2]}$ | $db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$ |
| $dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$ | $dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$ |
| $dW^{[1]} = dz^{[1]} x^T$ | $dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$ |
| $db^{[1]} = dz^{[1]}$ | $db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$ |

$$J(..) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

# 1. Shallow neural network

## Initializing weights




- What is weights are initialized to **zero**?
- Suppose all weights are zero:
  - $W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
  - $b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
  - $a_1^{[1]} = a_2^{[1]}$
  - $dz_1^{[1]} = dz_2^{[1]}$
  - $dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ (i.e., symmetric rows)
  - $W^{[1]} = W^{[1]} - \alpha dW$
  - $W^{[1]} = \begin{bmatrix} f & g \\ f & g \end{bmatrix}$ (i.e., symmetric rows)
- No need of TWO or more neurons … because all computations are same!
  - Do NOT initialize all weights are ZERO!!

# 1. Shallow neural network

**Initialize weights RANDOMLY!**

- W[1] = np.random.randn((2,2))***0.01**
  - **Small** random values are suggested!
  - If too large, $Z^{[1]} = W^{[1]}X + b^{[1]}$ will also be very large and $a^{[1]} = g^{[1]}(z^{[1]})$ will be in the flat areas and gradient descent will be very, very slooooooooow….

- b[1] = np.zero((2,1))  (b can be zero, no problem!)