Báo cáo tuần 3

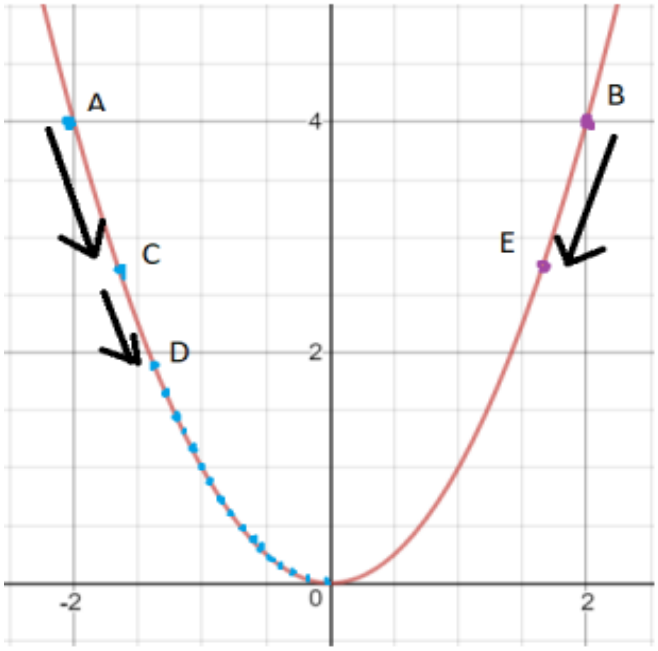
1. Gradient descent

1. Gradient descent :

Gradient descent là thuật toán tìm giá trị nhỏ nhất của hàm số f(x) dựa trên đạo hàm. Thuật toán:

- 1. Khởi tạo giá trị x = x0 tùy ý
- 2. Gán x = x - learning_rate * f'(x) (learning_rate là hằng số dương ví dụ learning_rate = 0.001)
- 3. Tính lại f(x): Nếu f(x) đủ nhỏ thì dừng lại, ngược lại tiếp tục bước 2

Thuật toán sẽ lặp lại bước 2 một số lần đủ lớn (100 hoặc 1000 lần tùy vào bài toán và hệ số learning_rate) cho đến khi f(x) đạt giá trị đủ nhỏ.



Hình 3.6: Ví dụ về thuật toán gradient descent

Việc chọn hệ số learning_rate cực kì quan trọng, có 3 trường hợp:

• Nếu learning_rate nhỏ: mỗi lần hàm số giảm rất ít nên cần rất nhiều lần thực hiện bước 2 để hàm số đạt giá trị nhỏ nhất.

• Nếu learning_rate hợp lý: sau một số lần lặp bước 2 vừa phải thì hàm sẽ đạt giá trị đủ nhỏ.

• Nếu learning_rate quá lớn: sẽ gây hiện tượng overshoot (như trong hình 3.8) và không bao giờ đạt được giá trị nhỏ nhất của hàm.

2. Các biến thể của Gradient Descent :

Dựa vào số lượng dữ liệu cho mỗi lần thực hiện bước 2 trong gradient descent là người ta chia ra làm 3 loại:

• Batch gradient descent: Dùng tất cả dữ liệu trong training set cho mỗi lần thực hiện bước tính đạo hàm.

• Stochastic gradient descent: Chỉ dùng một dữ liệu trong training set cho mỗi lần thực hiện bước tính đạo hàm.

• Mini-batch gradient descent: Dùng một phần dữ liệu trong training set cho mỗi lần thực hiện bước tính đạo hàm.

2.1. Batch Gradient Descent

Ở mỗi lần cập nhật tham số, ta dùng **toàn bộ dữ liệu huấn luyện** (tất cả các điểm từ x1->xm).

Vì vậy, **chỉ có 1 gradient duy nhất** được tính cho toàn bộ dataset, rồi dùng nó để cập nhật tham số.

Với hàm mất mát trung bình (loss function):

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)}, h_\theta(x^{(i)}))$$

Gradient:

$$\nabla J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L(y^{(i)}, h_\theta(x^{(i)}))$$

Cập nhật tham số:

$$\theta := \theta - \eta \cdot \nabla J(\theta)$$

 Kết quả: Rất chính xác nhưng **tốn nhiều tài nguyên**, đặc biệt với dữ liệu lớn.

2.2. Stochastic Gradient Descent

Ở mỗi lần cập nhật tham số, ta chỉ lấy **1 điểm dữ liệu duy nhất** (ví dụ x1), tính gradient, cập nhật tham số.

Sau đó lấy x2, x3, … cho đến hết tập dữ liệu.

Với mỗi mẫu $(x^{(i)}, y^{(i)})$:

$$\theta := \theta - \eta \cdot \nabla_\theta L(y^{(i)}, h_\theta(x^{(i)}))$$

 Vậy nên:

- Mỗi vòng lặp qua hết tập dữ liệu (epoch) sẽ có **m lần cập nhật** (nếu có mmm dữ liệu).

- Gradient tính trên 1 điểm dữ liệu thì **rất nhiễu (noise)** → đường đi của SGD lên xuống zig-zag quanh cực tiểu.

Nhưng chính "noise" này lại giúp SGD **thoát khỏi local minima** trong những bài toán phức tạp.

2.3. Mini-batch Gradient Descent

Đây là sự kết hợp giữa Batch GD và SGD.

Ta chia tập dữ liệu thành nhiều **mini-batch** (ví dụ mỗi batch 32 hoặc 64 điểm).

Với mỗi batch:

1. Lấy dữ liệu trong batch đó
2. Tính gradient trung bình trên batch
3. Cập nhật tham số

Sau đó chuyển sang batch tiếp theo cho đến khi hết dữ liệu (hết 1 epoch).

Với mini-batch có kích thước m:

$$\nabla J_{mini}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(y^{(i)}, h_\theta(x^{(i)}))$$

Cập nhật:

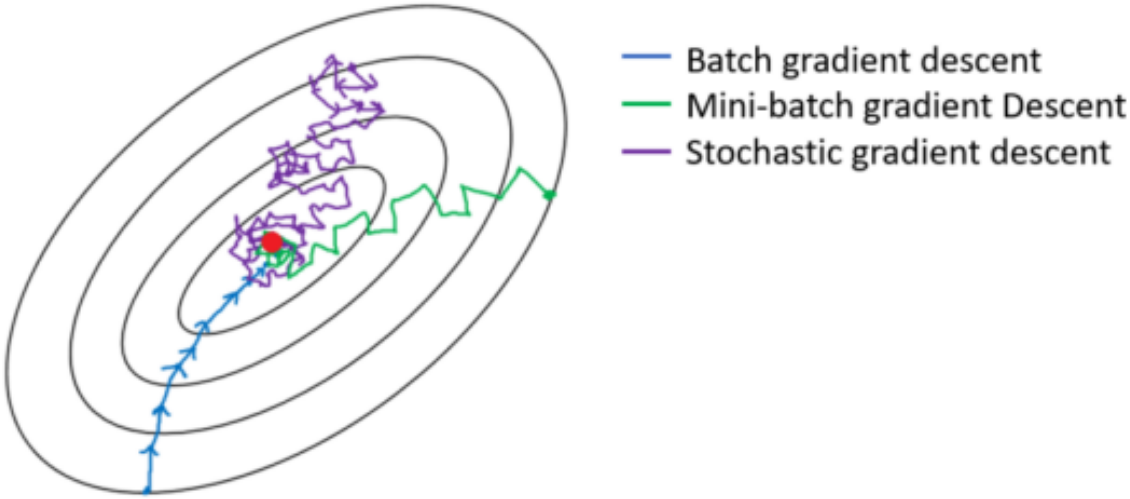
$$\theta := \theta - \eta \cdot \nabla J_{mini}(\theta)$$

 Tốc độ nhanh hơn BGD, ổn định hơn SGD.

Hưởng lợi từ **parallel computing** (GPU xử lý song song các batch).

Thường là lựa chọn **chuẩn trong Deep Learning**.

Lưu ý: Cần chọn batch size phù hợp (quá nhỏ → giống SGD, quá lớn → giống BGD).



2. Neural network

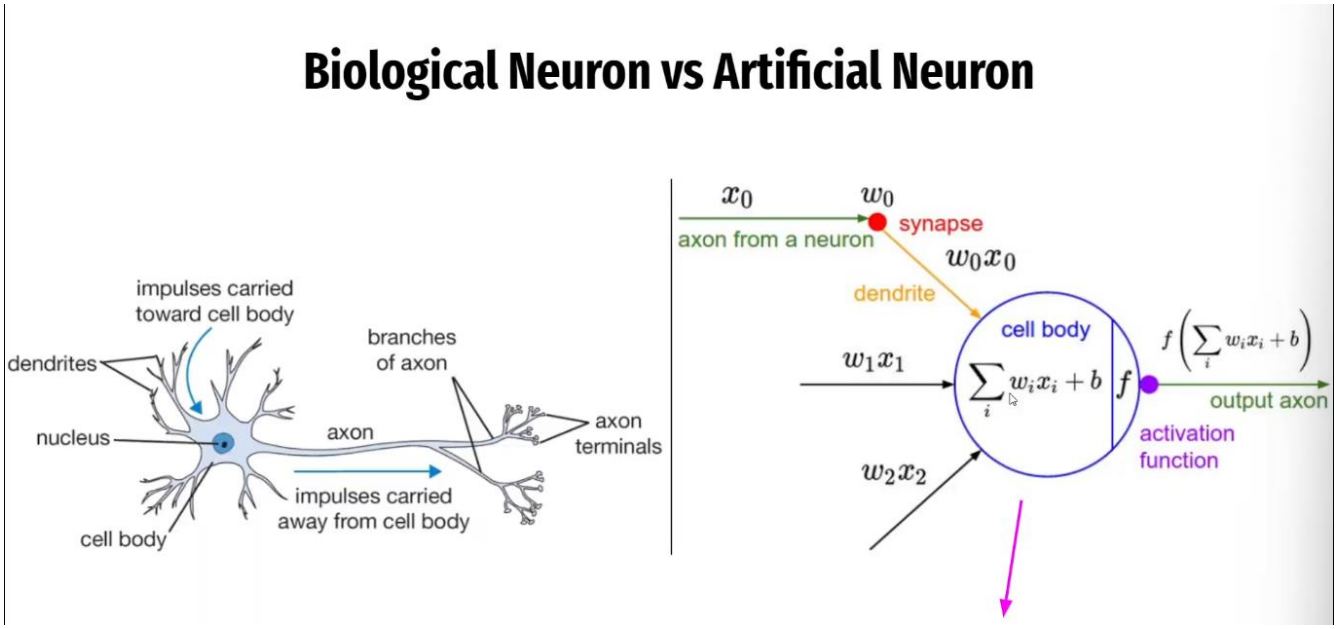
1.Neural network?

1.1 Neural network là gì?

Neural Network (Mạng nơ-ron nhân tạo) là một mô hình tính toán được lấy cảm hứng từ cấu trúc và hoạt động của não người, dùng trong lĩnh vực trí tuệ nhân tạo và học máy.

Định nghĩa đơn giản:

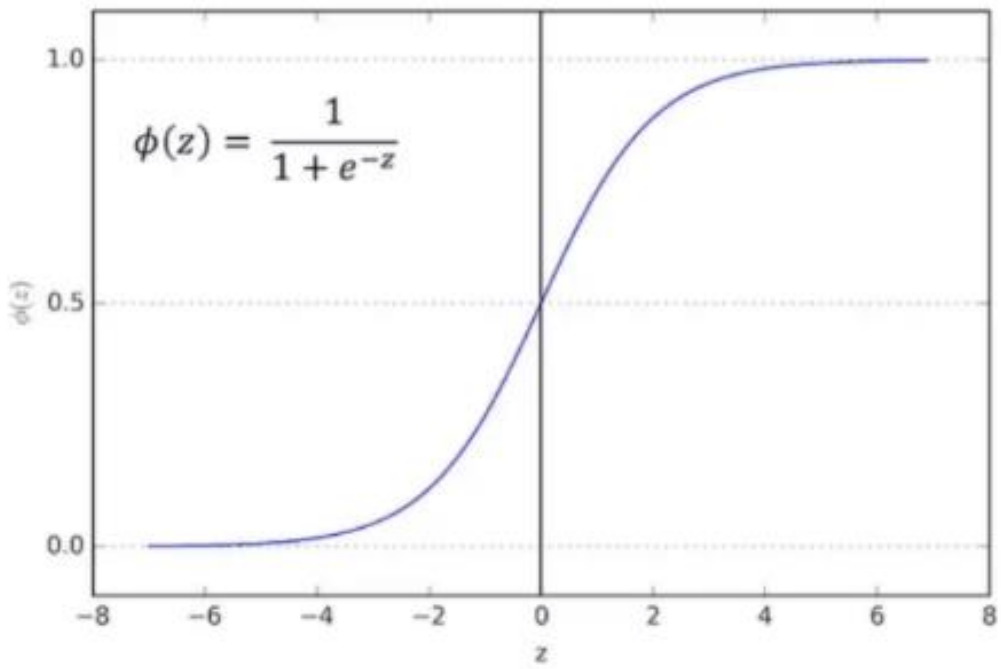
Mạng nơ-ron là một hệ thống gồm nhiều **lớp (layers)** và **nút (neurons)** được kết nối với nhau, dùng để xử lý và học từ dữ liệu đầu vào nhằm đưa ra dự đoán hoặc quyết định.



Neuron = linear classifier + activation function

1.2.Activation function

1.2.1. Sigmoid activation function

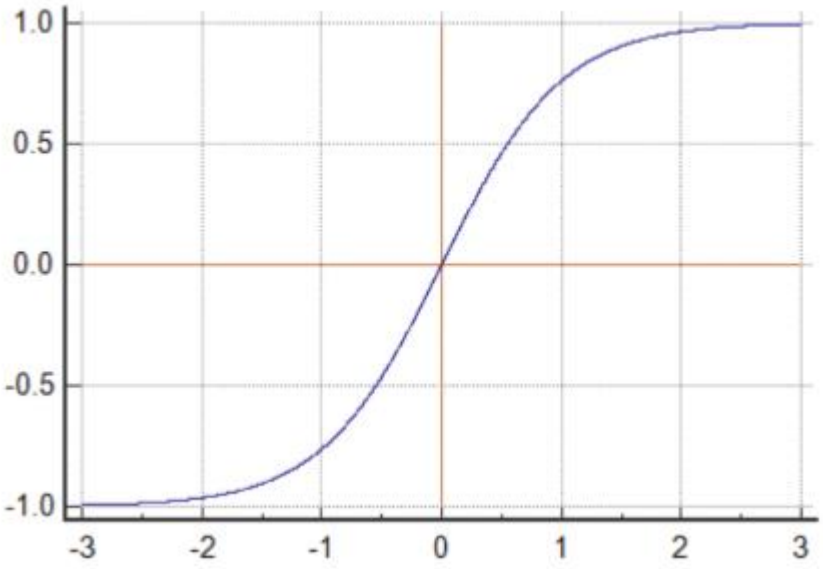


-Đạo hàm: σ(x)(1−σ(x)) → **tiệm cận 0** khi |x| lớn ⇒ ***vanishing gradient*.**

-Ưu: diễn giải như xác suất; dùng tốt ở **output nhị phân**.

-Nhược: chậm học ở mạng sâu; gradient nhỏ; vanishing.

1.2.2. Tanh activation function



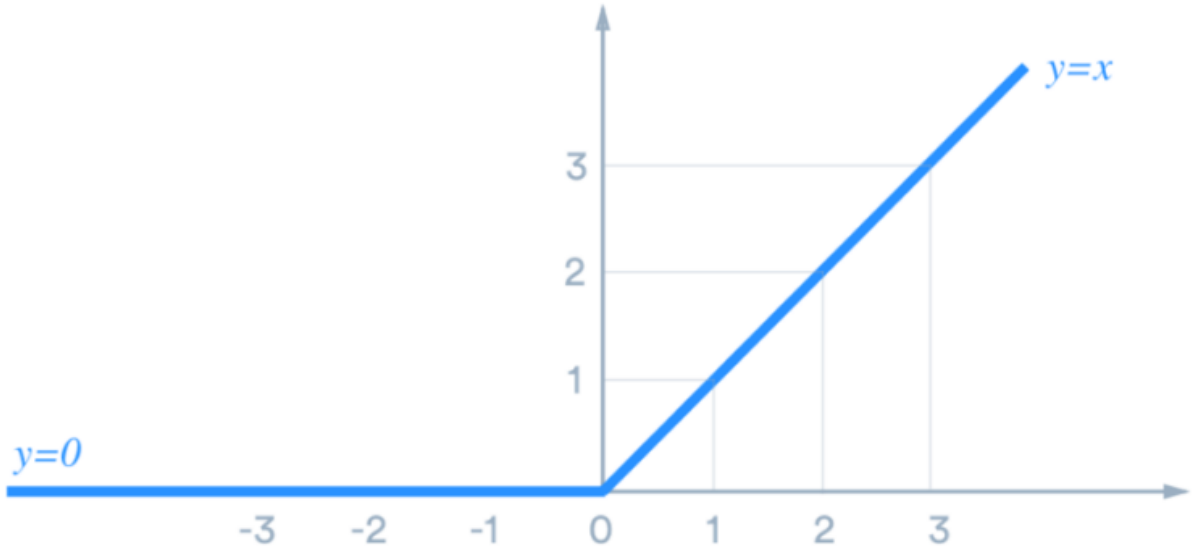
-Công thức: tanh(x); miền: (−1,1);

-Đạo hàm: 1−tanh²(x) → cũng ***vanishing gradient***khi |x| lớn.

-Ưu: tối ưu dễ hơn sigmoid vì trung tâm tại 0

-Nhược: vẫn vanishing; ít dùng cho mạng sâu không có chuẩn hóa.

1.2.3.ReLU activation function

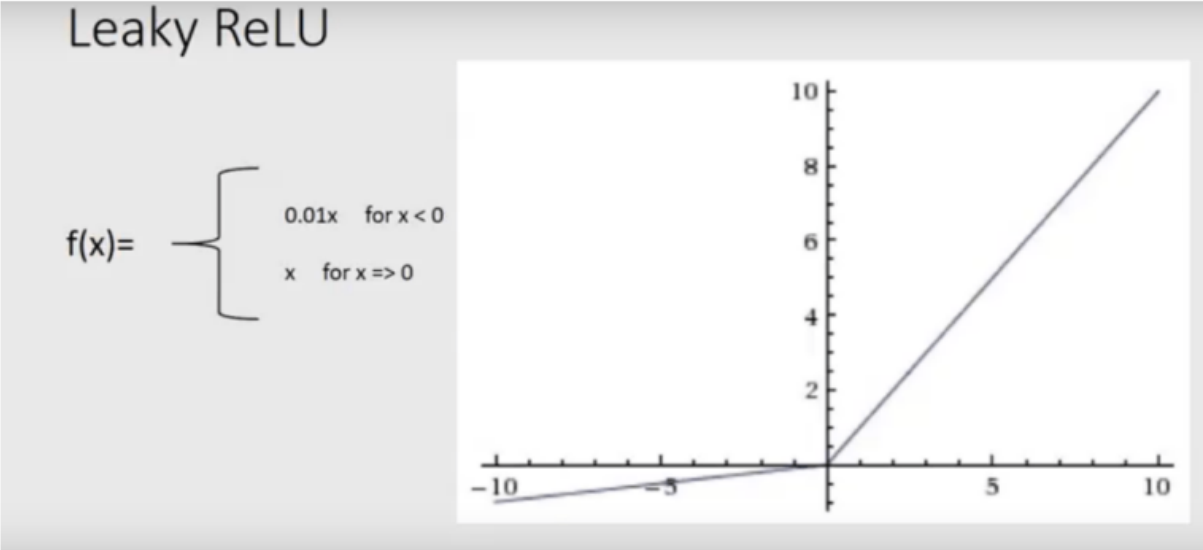


-Hàm relu (rectified linear unit): y = max(0, x)

-Nhận xét:
• Hàm ReLU activation đơn giản để tính => thời gian train model nhanh hơn.
• Đạo hàm là 1 với x >= 0 nên không bị vanishing gradient.

-Tuy nhiên với các node có giá trị nhỏ hơn 0, qua ReLU activation sẽ thành 0, hiện tượng đấy gọi là "Dying ReLU". Nếu các node bị chuyển thành 0 thì sẽ không có ý nghĩa với bước linear activation ở lớp tiếp theo và các hệ số tương ứng từ node đấy cũng không được cập nhật với gradient descent.
=> Leaky ReLU ra đời.

1.2.4.Leaky ReLU



- Hàm Leaky ReLU có các điểm tốt của hàm ReLU và giải quyết được vấn đề Dying ReLU bằng
cách xét một độ dốc nhỏ cho các giá trị âm thay vì để giá trị là 0.

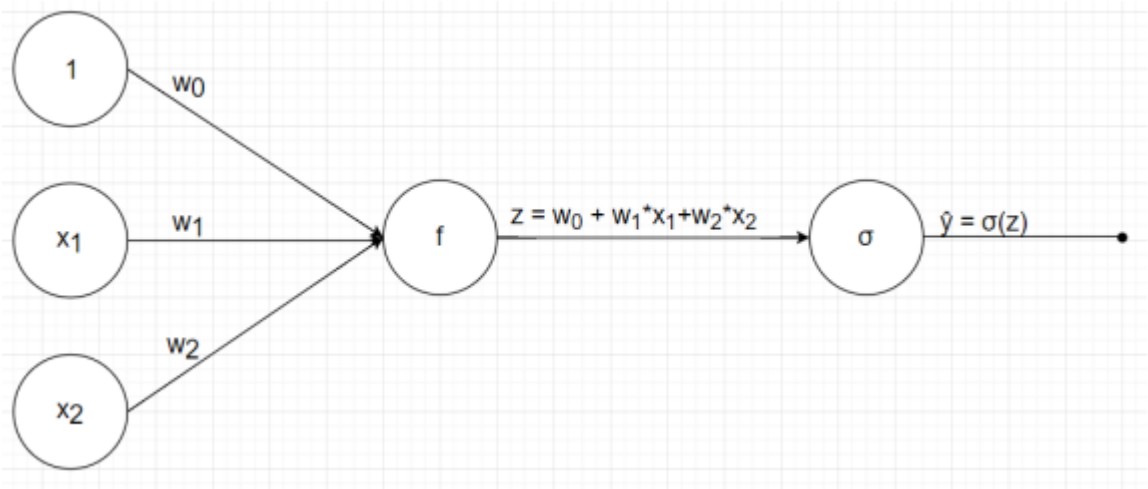
-Lời khuyên: Mặc định nên dùng ReLU làm hàm activation. Không nên dùng hàm sigmoid.

2. Mô hình Neuron Network

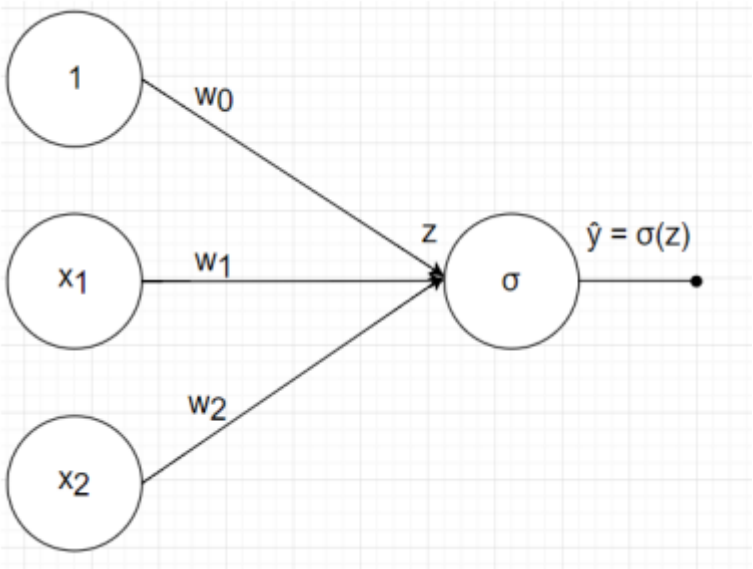
2.1.Logistic regression dưới góc nhìn của Neuron Network

Logistic regression là mô hình neural network đơn giản nhất chỉ với input layer và output layer.
Mô hình của logistic regression từ bài trước là: $\hat{y}$= σ(w0 +w1 ∗ x1 +w2 ∗ x2). Có 2 bước:

- Tính tổng linear: z = 1∗ w0 +x1∗ w1 +x2∗ w2
- Áp dụng sigmoid function: $\hat{y}$ = σ(z)

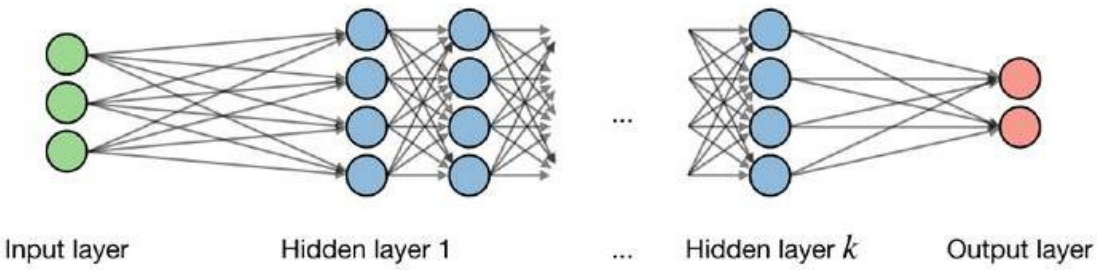


Biểu diễn gọn lại:



Model = architecture + parameters

2.2.Cấu trúc tổng quát



Cấu trúc cơ bản của một mạng nơ-ron:

1. **Input layer (Lớp đầu vào):** Nhận dữ liệu đầu vào (ví dụ: ảnh, văn bản, số liệu…).

2. **Hidden layers (Các lớp ẩn):** Xử lý và trích xuất đặc trưng từ dữ liệu (càng nhiều lớp thì càng "sâu").

3. **Output layer (Lớp đầu ra):** Đưa ra kết quả dự đoán hoặc phân loại.

Các hình tròn được gọi là node. Mỗi node trong hidden layer và output layer :

• Liên kết với tất cả các node ở layer trước đó với các hệ số w riêng.

• Mỗi node có 1 hệ số bias b riêng.

• Diễn ra 2 bước: tính tổng linear và áp dụng activation function.

2.3 Parameters

-Số node trong hidden layer thứ i là $l^{(i)}$

-Ma trận $W^{(k)}$ kích thước $l^{(k-1)} * l^{(k)}$ là ma trận hệ số giữa layer (k-1) và layer k, trong đó $w_{ij}^{(k)}$ là hệ số kết nối từ node thứ i của layer k-1 đến node thứ j của layer k.

-Vector $b^{(k)}$ kích thước $l^k * 1$ là hệ số bias của các node trong layer k, trong đó $b_i^{(k)}$ là bias của node thứ i trong layer k.

-Với node thứ i trong layer l có bias $b_i^{(l)}$ thực hiện 2 bước:

• Tính tổng linear: là tổng tất cả các node trong layer trước nhân với hệ số w tương ứng, rồi cộng với bias b.

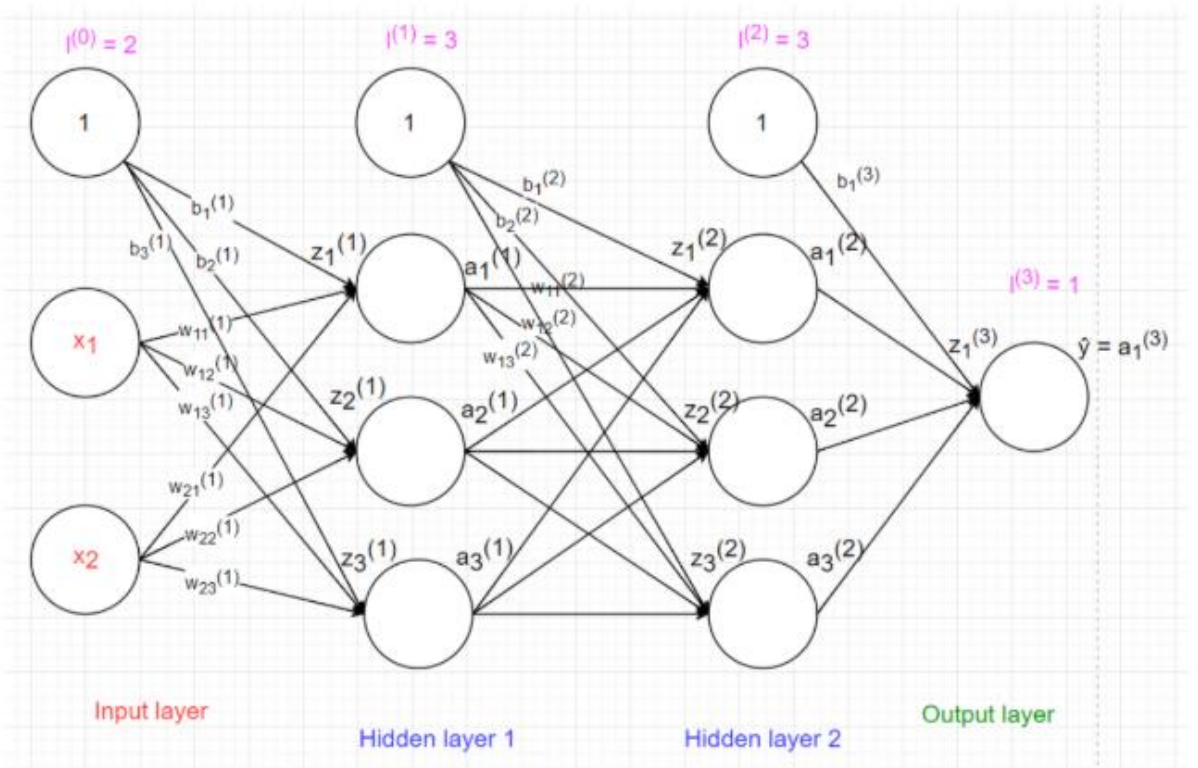
$$z_i^{(l)} = \sum_{j=1}^{l^{(l-1)}} a_j^{(l-1)} * w_{ji}^{(l)} + b_i^{(l)},$$

• Áp dụng activation function: $a_i^{(l)}$= σ($z_i^{(l)}$)

-Vector $z^{(k)}$ kích thước $l^{(k+1)}$ là giá trị các node trong layer k sau bước tính tổng linear.

-Vector $a^{(k)}$ kích thước $l^{(k+1)}$ là giá trị của các node trong layer k sau khi áp dụng hàm activation function.

Ví dụ



Giả sử ta có mô hình neural network trên gồm 3 layer. Input layer có 2 node, hidden layer 1 có 3 node, hidden layer 2 có 3 node và output layer có 1 node.

Do mỗi node trong hidden layer và output layer đều có bias nên trong input layer và hidden layer cần thêm node 1 để tính bias (nhưng không tính vào tổng số node layer có).

Để nhất quán về mặt ký hiệu, gọi input layer là $a^{(0)}$ (= x) kích thước 2*1.

$$z^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix} = \begin{bmatrix} a_1^{(0)} * w_{11}^{(1)} + a_2^{(0)} * w_{21}^{(1)} + a_3^{(0)} * w_{31}^{(1)} + b_1^{(1)} \\ a_1^{(0)} * w_{12}^{(1)} + a_2^{(0)} * w_{22}^{(1)} + a_3^{(0)} * w_{32}^{(1)} + b_2^{(1)} \\ a_1^{(0)} * w_{13}^{(1)} + a_2^{(0)} * w_{23}^{(1)} + a_3^{(0)} * w_{33}^{(1)} + b_3^{(1)} \end{bmatrix}$$

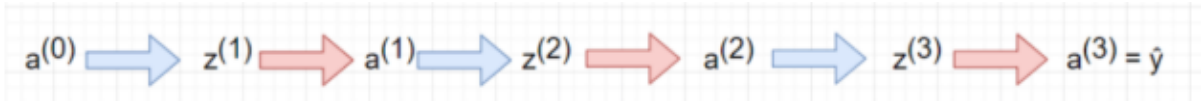
$$= (W^{(1)})^T * a^{(0)} + b^{(1)}$$

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

Tương tự ta có:

$$z^{(2)} = (W^{(2)})^T * a^{(1)} + b^{(2)}$$
$$a^{(2)} = \sigma(z^{(2)})$$
$$z^{(3)} = (W^{(3)})^T * a^{(2)} + b^{(3)}$$
$$\hat{y} = a^{(3)} = \sigma(z^{(3)})$$

Quá trình trên được gọi là **Feedforward:**



3. Backpropagation

1.Backpropagation

Định nghĩa: Backpropagation (BP) là **thuật toán tính đạo hàm của hàm mất mát (loss function) theo từng trọng số trong mạng nơ-ron**, bằng cách **lan truyền lỗi ngược từ output về input**.

Mục tiêu: Tìm gradient cho tất cả các trọng số W và b

Công cụ toán học: Dựa trên **Quy tắc dây chuyền (Chain Rule)** trong đạo hàm:

$$\frac{\partial L(z,y)}{\partial w} = \frac{\partial L(z,y)}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Kết quả, trọng số được cập nhật như sau:

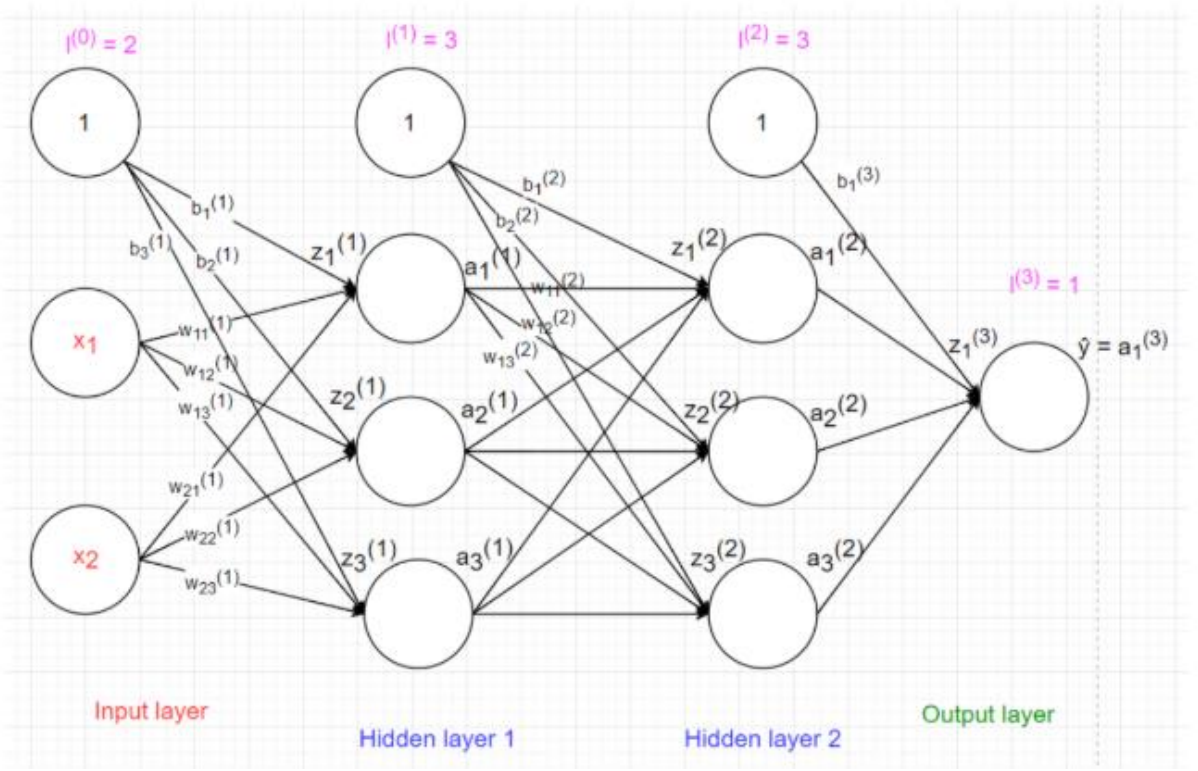
$$w \leftarrow w - \eta \frac{\partial L(z,y)}{\partial w}$$

Cập nhật trọng số (Updating weights) – Trong mạng nơ-ron, việc cập nhật trọng số diễn ra như sau:

1. Lấy một batch dữ liệu huấn luyện.
2. Thực hiện lan truyền xuôi để tính loss.
3. Lan truyền ngược loss để lấy gradient.
4. Dùng gradient để cập nhật trọng số của mạng.

2.Ví dụ

Tiếp tục với mô hình neuron network ở phần 2:



Bước 1: Đạo hàm tại Output layer :

Ta bắt đầu từ loss:

$$\frac{\partial J}{\partial \hat{Y}}, \text{ trong đó } \hat{Y} = A^{(3)}$$

Tính gradient theo công thức;

$$\delta^{(L)} = \frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial A^{(L)}}{\partial Z^{(L)}}$$

Bước 2: Gradient với W và b theo hidden layer 2:

Tính:

$$\frac{\partial J}{\partial W^{(3)}} = (A^{(2)})^T * \left(\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial A^{(3)}}{\partial Z^{(3)}}\right), \quad \frac{\partial J}{\partial b^{(3)}} = \left(\sum \left(\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial A^{(3)}}{\partial Z^{(3)}}\right)\right)^T$$

Và:

$$\frac{\partial J}{\partial A^{(2)}} = \left(\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial A^{(3)}}{\partial Z^{(3)}}\right) * (W^{(3)})^T$$

Bước 3: Gradient với W và b theo hidden layer 1:

Tính:

$$\frac{\partial J}{\partial W^{(2)}} = (A^{(1)})^T * \left(\frac{\partial A^{(2)}}{\partial Z^{(2)}} \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}}\right), \quad \frac{\partial J}{\partial b^{(2)}} = \left(\sum \left(\frac{\partial A^{(2)}}{\partial Z^{(2)}} \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}}\right)\right)^T$$

Và:

$$\frac{\partial J}{\partial A^{(1)}} = \left(\frac{\partial A^{(2)}}{\partial Z^{(2)}} \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}}\right) * (W^{(2)})^T$$

Bước 4: Tiếp tục cho đến input

$$\frac{\partial J}{\partial W^{(1)}} = (A^{(0)})^T * \left(\frac{\partial A^{(1)}}{\partial Z^{(1)}} \otimes \frac{\partial A^{(1)}}{\partial Z^{(1)}}\right),$$

trong đó $A^{(0)} = X$

Nếu network có nhiều layer hơn thì cứ tiếp tục cho đến khi tính được đạo hàm của loss function J với tất cả các hệ số W và bias b.

Ở phần trên với quá trình feedforward thì ở phần này quá trình tính đạo hàm sẽ ngược lại: