

# Introduction au Machine Learning

---

Réseaux de neurones :  
*représentation et apprentissage*

**Emmanuel Dellandréa**  
**[emmanuel.dellandrea@ec-lyon.fr](mailto:emmanuel.dellandrea@ec-lyon.fr)**

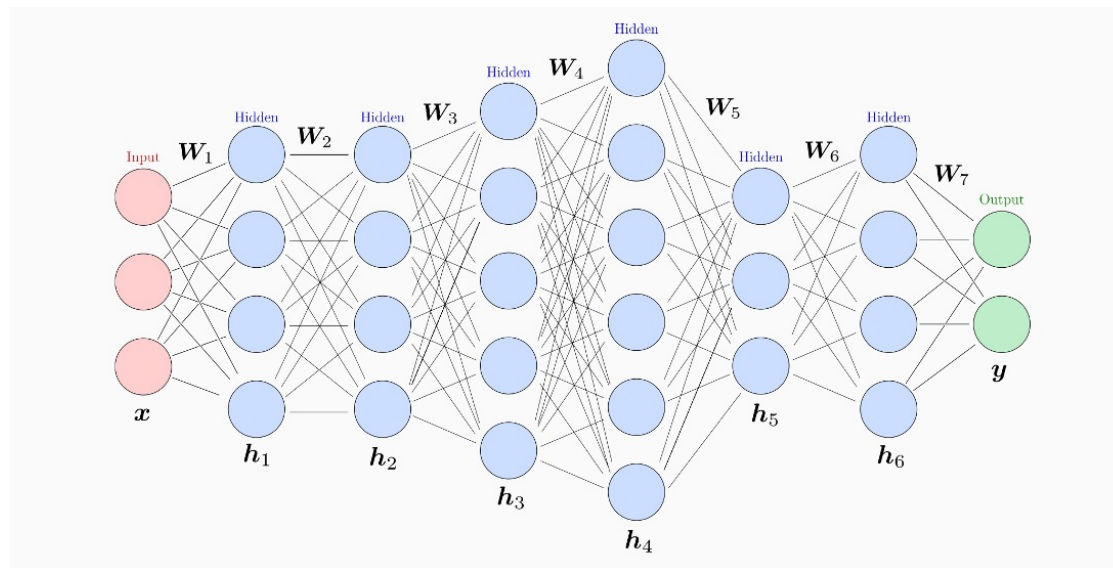
Version du 14/01/2025



# Introduction

- Qu'est-ce qu'un réseau de neurones ?

→ Ce sont des systèmes informatiques inspirés du cerveau humain, conçus pour reconnaître des motifs et apprendre à partir de données.

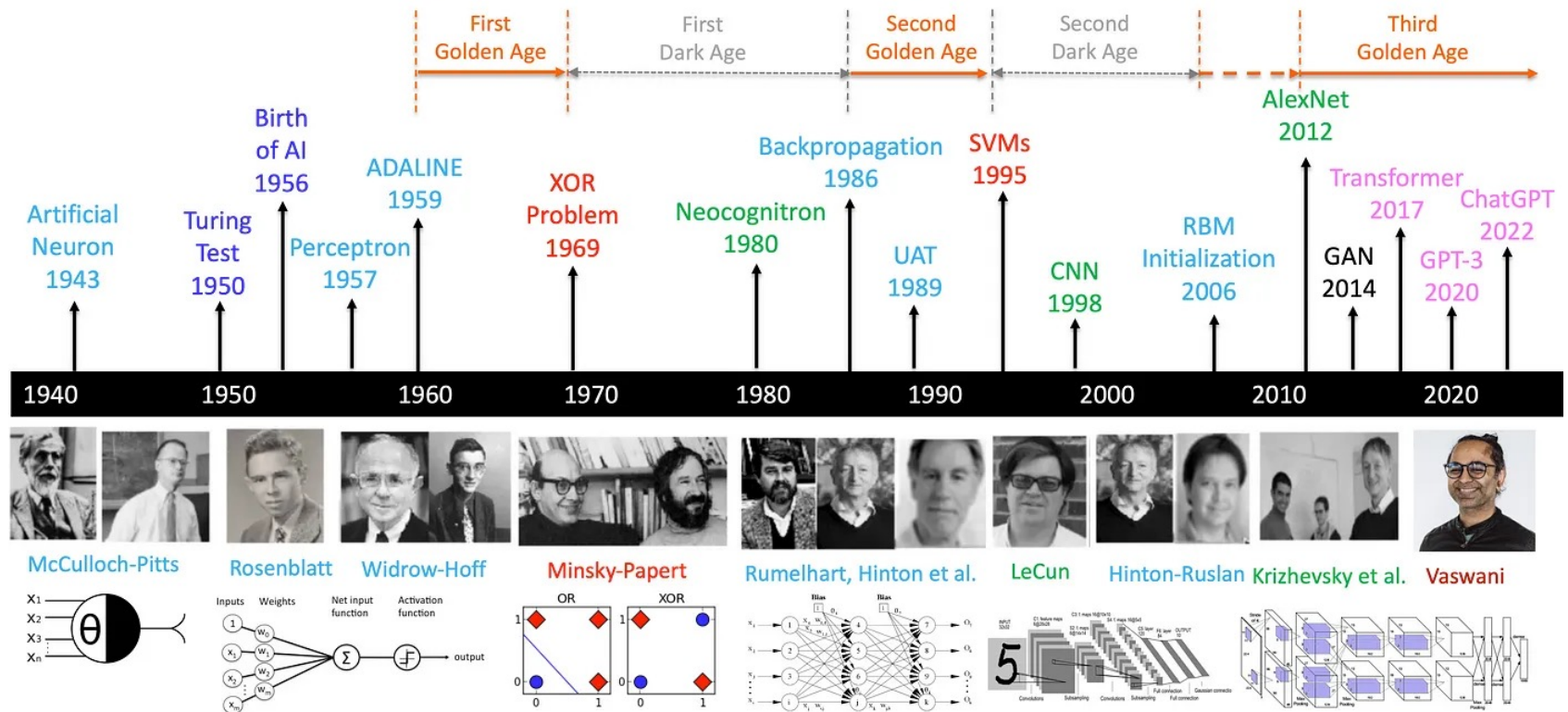


# Introduction

- Intérêts des réseaux de neurones ?
  - Résoudre des problèmes complexes tels que l'analyse d'images, le traitement de la parole et la compréhension du langage naturel
  - Gérer de grandes quantités de données et identifier des motifs que les humains pourraient ne pas remarquer
  - Les réseaux neuronaux sont des éléments clés des architectures modernes d'apprentissage profond (réseaux de neurones convolutifs, transformers, ...)

# Histoire des réseaux de neurones

## A Brief History of AI with Deep Learning



Source : <https://medium.com/@lmpo/a-brief-history-of-ai-with-deep-learning-26f7948bc87b>

# Quelques domaines d'application des RN

- Vision par Ordinateurs
- Traitement et analyse du langage (NLP)
- Traitement et analyse de l'audio
- Santé et Biomédecine
- Systèmes autonomes
- Jeux et divertissement
- Finance et économie
- ....

# Rappel : régression linéaire

- Fonction de prédiction :

$$f_{\theta}(x) = \sum_{i=0}^d \theta_i x_i = \theta^T x$$

- Fonction de coût :

$$J(\theta) = \frac{1}{2N} \sum_{i=1}^N (f_{\theta}(x^{(i)}) - y^{(i)})^2$$

# Rappel : régression linéaire

- Fonction de prédiction :

$$f_{\theta}(x) = \sum_{i=0}^d \theta_i x_i = \theta^T x$$

- Fonction de coût :

$$J(\theta) = \frac{1}{2N} \sum_{i=1}^N (f_{\theta}(x^{(i)}) - y^{(i)})^2$$

# Rappel : régression logistique

- Fonction de prédiction :

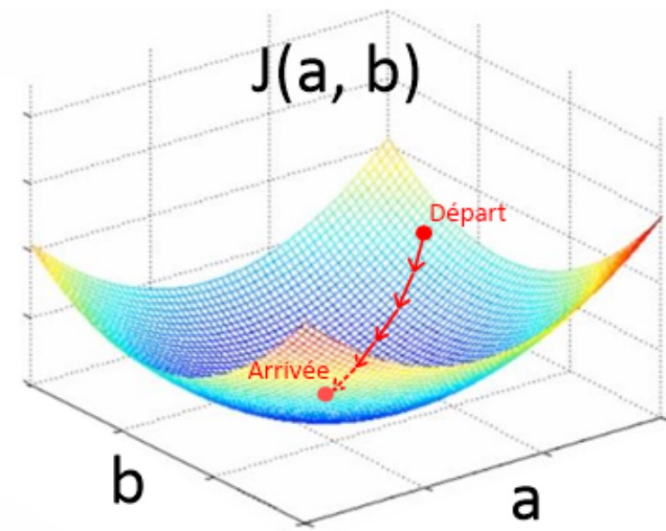
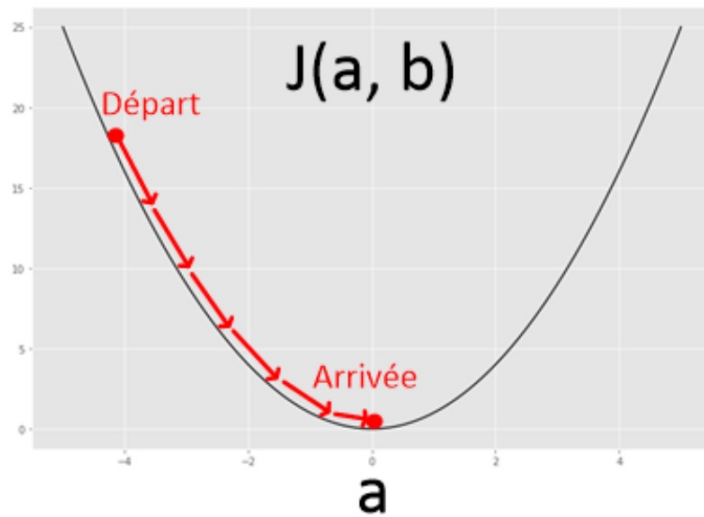
$$f_{\theta}(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

- Fonction de coût :

$$J(\theta) = -l(\theta) = -\sum_{i=1}^N y^{(i)} \log(f_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\theta}(x^{(i)}))$$



# Rappel : descente du gradient



# Rappel : descente du gradient

- Répéter itérativement jusqu'à convergence (simultanément pour chaque  $j$ ) :

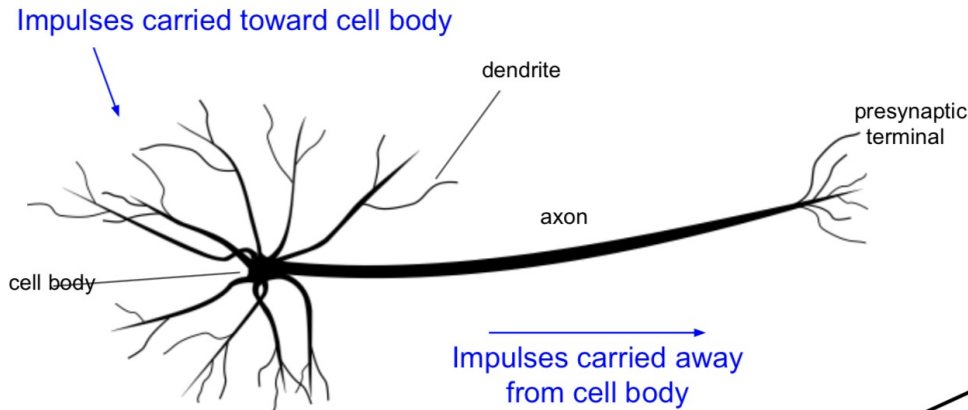
$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

- Soit :

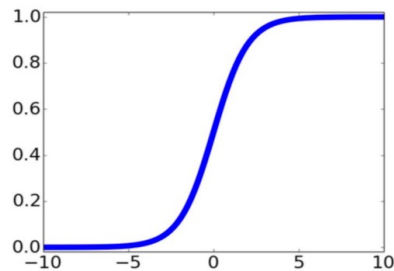
$$\theta_j = \theta_j - \alpha \frac{1}{N} \sum_{i=1}^N (f_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

- Expression identique pour les régressions linéaires et logistiques mais avec  $f_{\theta}$  différente

# Neurone biologique vs neurone artificiel

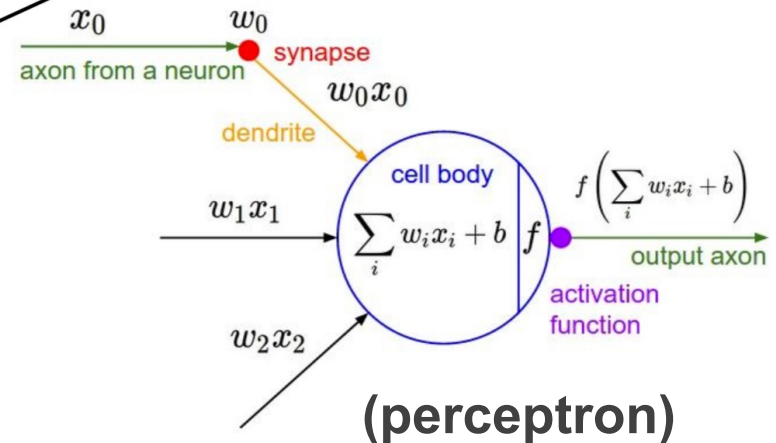


This image by Felipe Peruchio is licensed under [CC-BY 3.0](#)

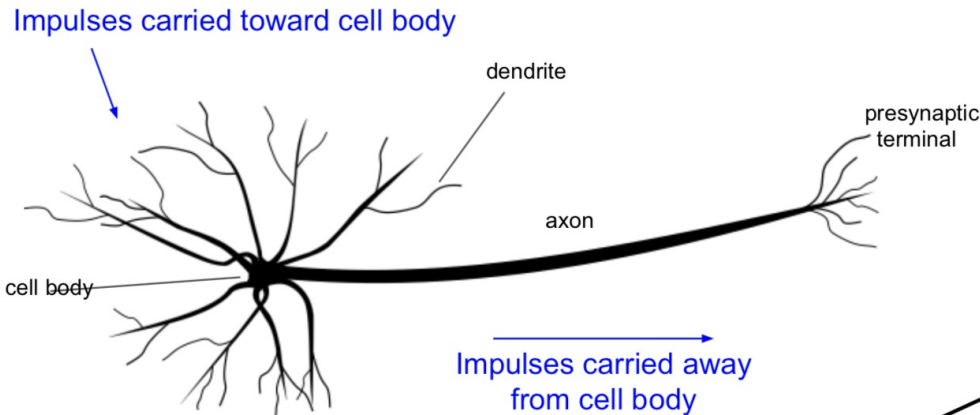


sigmoid activation function

$$\frac{1}{1 + e^{-x}}$$

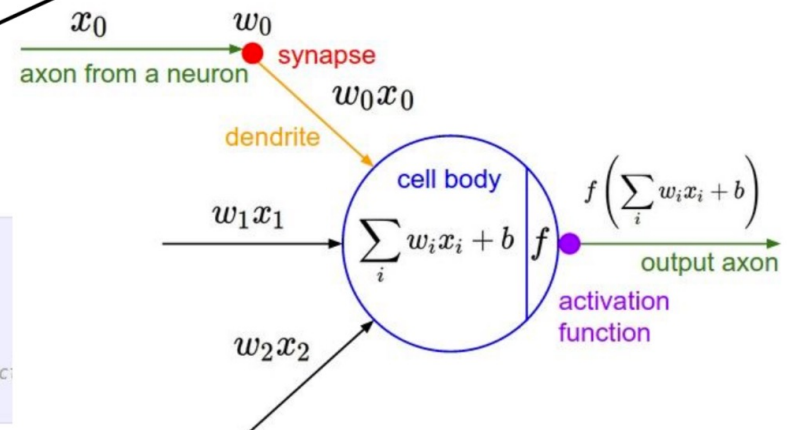


# Neurone biologique vs neurone artificiel



This image by Felipe Peruchio  
is licensed under [CC-BY 3.0](#)

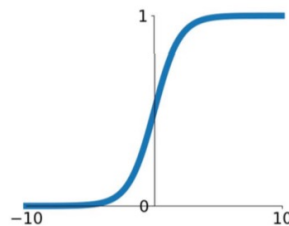
```
class Neuron:
    # ...
    def neuron_tick(inputs):
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation func
        return firing_rate
```



# Fonctions d'activation

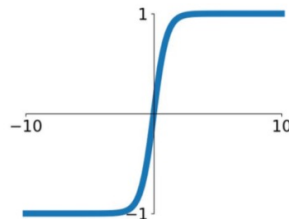
## Sigmoid

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



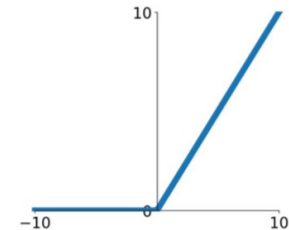
## tanh

$$\tanh(x)$$



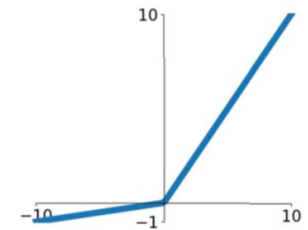
## ReLU

$$\max(0, x)$$



## Leaky ReLU

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

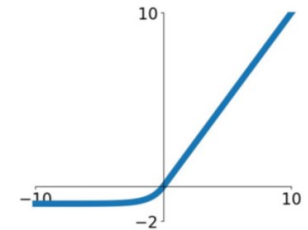


## Maxout

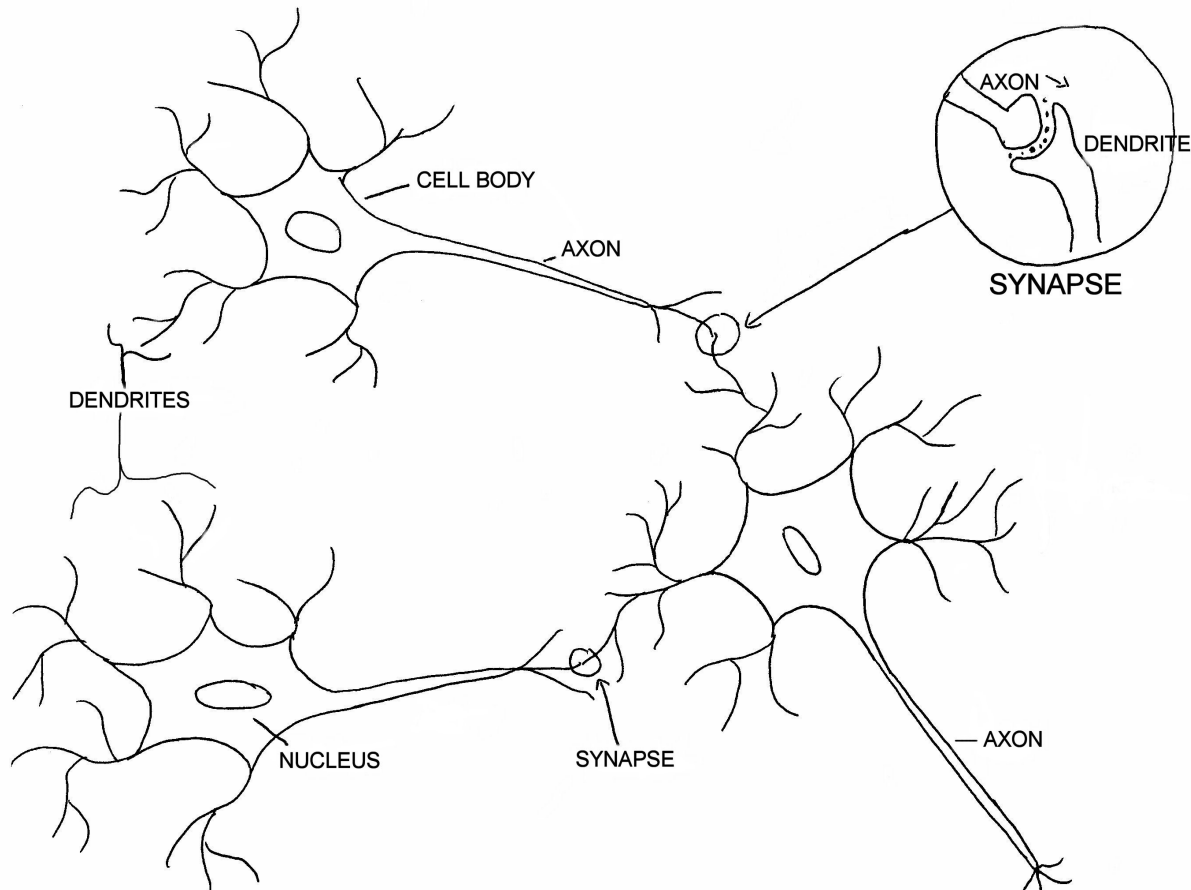
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

## ELU

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

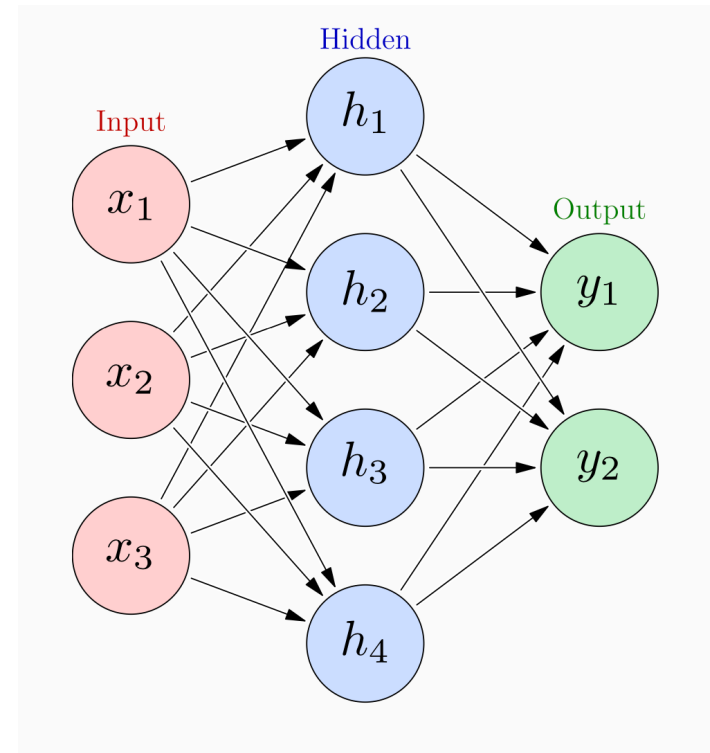


# Réseau de neurones biologiques

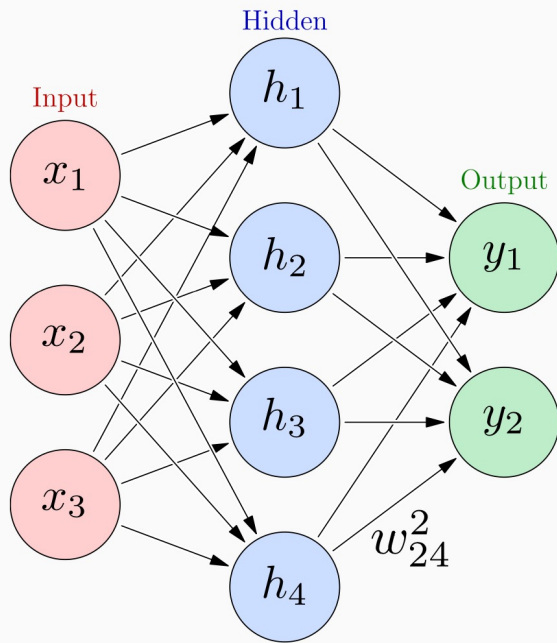


# Réseau de neurones artificiels

- Interconnexion des neurones
- Organisation des neurones en couches
  - Couche d'entrée
  - Couche(s) cachée(s)
  - Couche de sortie
- Chaque neurone est connecté à tous les neurones de la couche précédente et tous ceux de la couche suivante



# Réseau de neurones



$$h_1 = g_1 (w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1)$$

$$h_2 = g_1 (w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1)$$

$$h_3 = g_1 (w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1)$$

$$h_4 = g_1 (w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1)$$

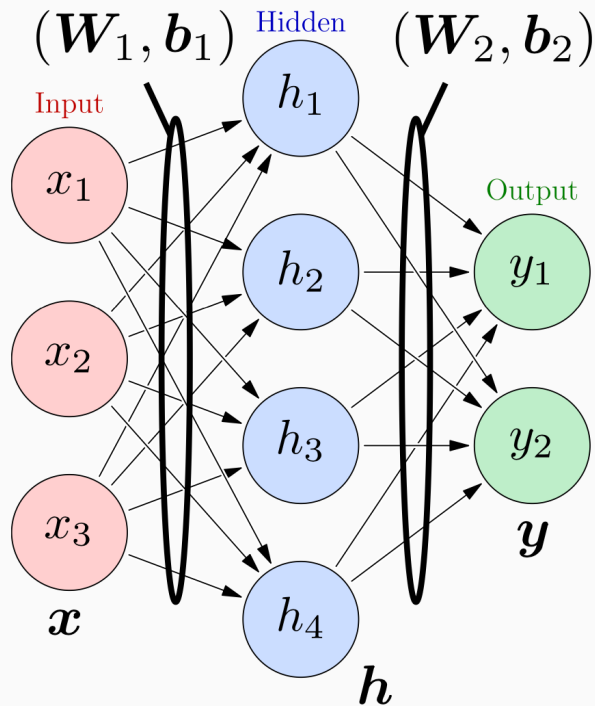
$$y_1 = g_2 (w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2)$$

$$y_2 = g_2 (w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2)$$

- $w_{ij}^k$  : poids synaptiques entre le neurone précédent  $j$  et le neurone suivant  $i$  dans la couche  $k$
- $g_k$  sont les fonctions d'activation appliquées à chaque potentiel d'entrée (unité linéaire)



# Réseau de neurones



$$h_1 = g_1 (w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1)$$

$$h_2 = g_1 (w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1)$$

$$h_3 = g_1 (w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1)$$

$$h_4 = g_1 (w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1)$$

---


$$h = g_1 (W_1 x + b_1)$$

$$y_1 = g_2 (w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2)$$

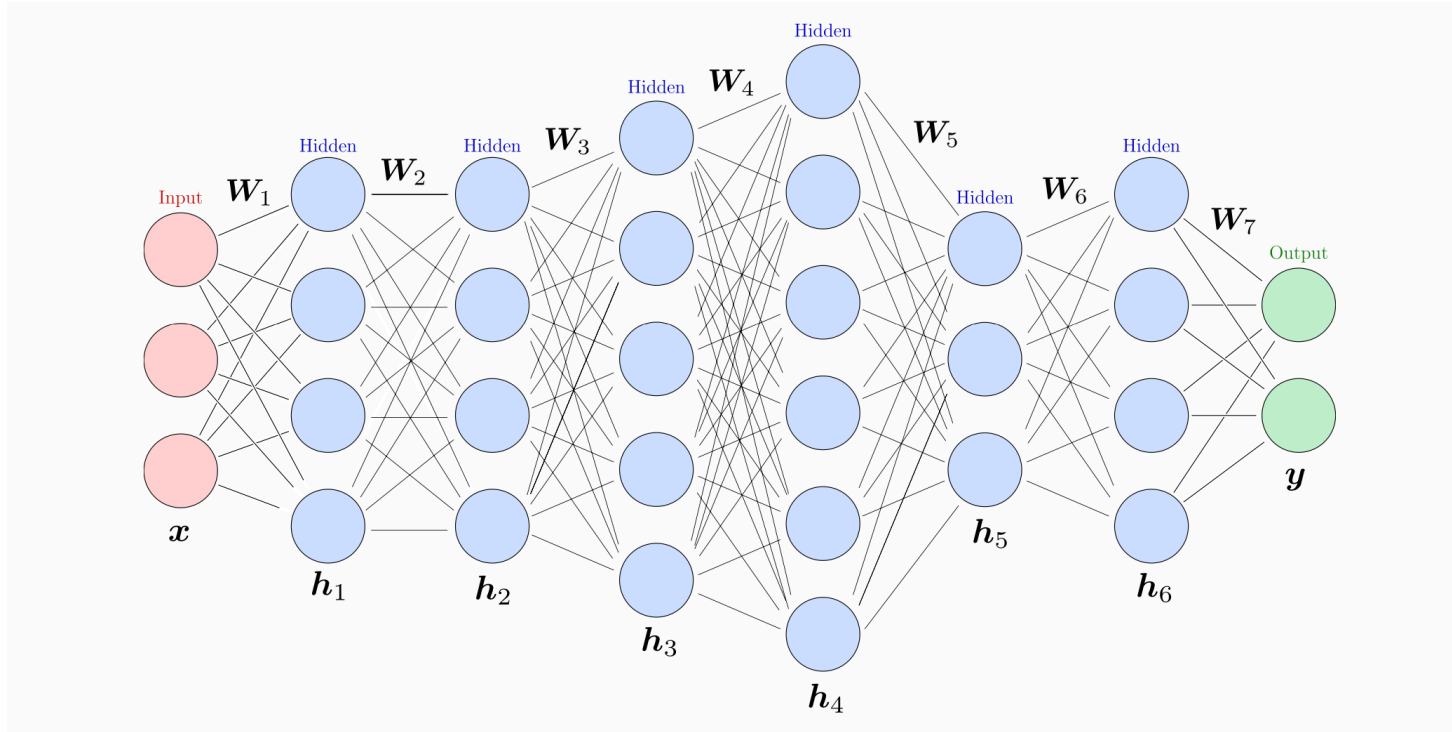
$$y_2 = g_2 (w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2)$$

---


$$y = g_2 (W_2 h + b_2)$$

- Les matrices de poids  $W_k$  et les vecteurs de biais  $b_k$  sont appris à partir des données d'apprentissage

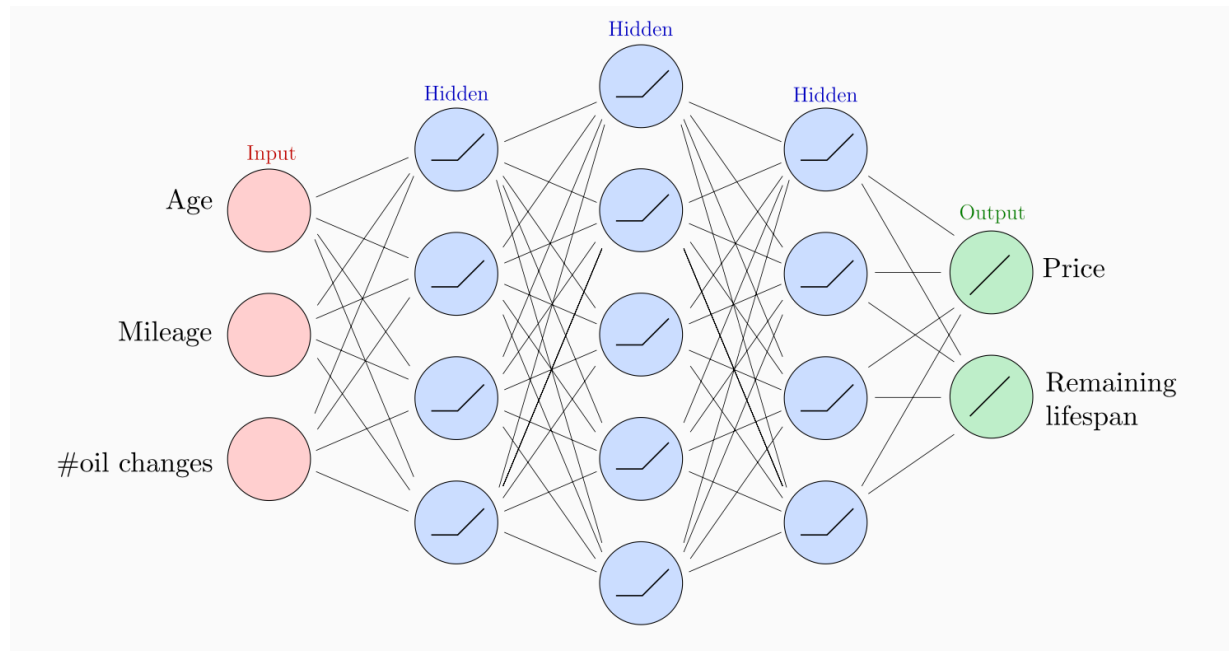
# Réseau de neurones



- Si plus d'une couche cachée : réseau « profond »
- Chaque couche peut avoir un nombre différent de neurones
- Les fonctions d'activation peuvent être différentes selon les couches

# Architectures typiques

## ■ Régression



- Couches cachées :  $\text{ReLU}(a) = \max(a, 0)$
- Couche de sortie linéaire :  $g(a) = a$

# Architectures typiques

- Régression

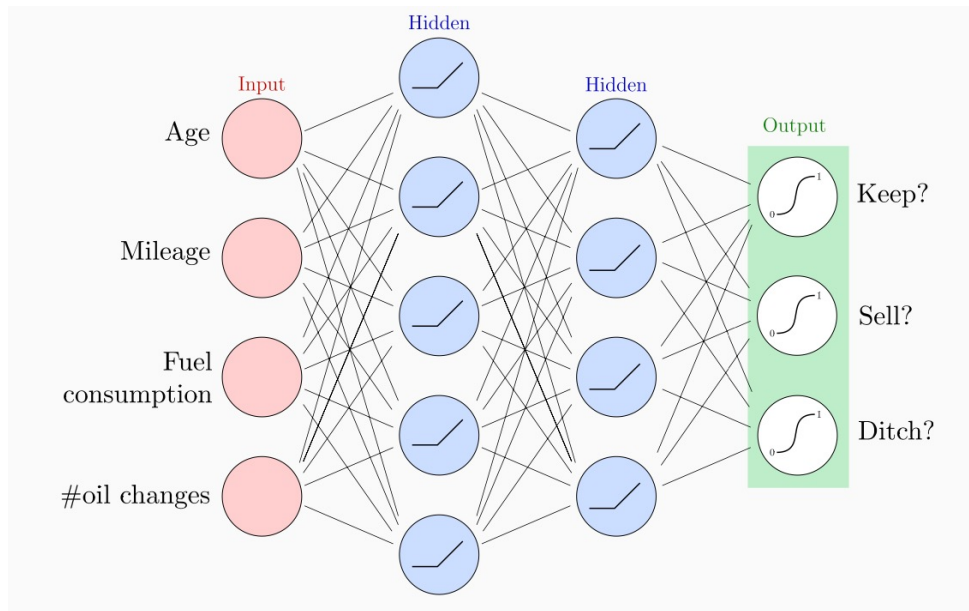
- Fonction de coût : Erreur quadratique moyenne

$$E(\mathbf{W}) = \sum_{i=1}^N \|\mathbf{y}^i - \mathbf{d}^i\|_2^2 = \sum_{i=1}^N \|f(\mathbf{x}^i; \mathbf{W}) - \mathbf{d}^i\|_2^2$$

- Minimiser  $E(\mathbf{W})$  par descente du gradient

# Architectures typiques

- Classification multi-classes



- Couches cachées :  $\text{ReLU}(a)$
- Couche de sortie : softmax (comme probabilité)
- Décision finale : classe ayant la plus grande probabilité

$$\text{softmax}(\mathbf{a})_k = \frac{\exp(a_k)}{\sum_{\ell=1}^K \exp(a_\ell)}$$

# Architectures typiques

- Classification multi-classes
  - Fonction de coût : Entropie croisée (cross-entropy)

$$E(\mathbf{W}) = - \sum_{i=1}^N \sum_{k=1}^K d_k^i \log y_k^i \quad \text{with} \quad \mathbf{y}^i = f(\mathbf{x}^i; \mathbf{W}) = \text{softmax}(\mathbf{a}) \in (0, 1)^K$$

- Minimiser  $E(\mathbf{W})$  par descente du gradient

# Optimisation

- Paramètres du réseau :

$$\mathbf{W} = (\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_L, \mathbf{b}_L)$$

- Apprentissage : minimiser la fonction de coût  $E(\mathbf{W})$  par descente du gradient

$$\text{Objectif : } \min_{\mathbf{W}} E(\mathbf{W}) \quad \text{où : } E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} L(\mathbf{y}^i; \mathbf{d}^i)$$

$$\Rightarrow \nabla E(\mathbf{W}) = \left( \frac{\partial E(\mathbf{W})}{\partial \mathbf{W}_1} \quad \frac{\partial E(\mathbf{W})}{\partial \mathbf{b}_1} \quad \dots \quad \frac{\partial E(\mathbf{W})}{\partial \mathbf{W}_L} \quad \frac{\partial E(\mathbf{W})}{\partial \mathbf{b}_L} \right)^T = 0$$

- Gradients :  $\nabla_{\mathbf{W}_k} E(\mathbf{W})$  and  $\nabla_{\mathbf{b}_k} E(\mathbf{W})$

# Optimisation

- Descente du gradient

$$w_{i,j}^{k,t+1} \leftarrow w_{i,j}^{k,t} - \gamma \frac{\partial E(\mathbf{W}^t)}{\partial w_{i,j}^k}$$

- Comment calculer  $\frac{\partial E(\mathbf{W}^t)}{\partial w_{i,j}^k}$  ??

→ par rétropropagation



# Optimisation

- Fonctions de coût standard
  - Régression :

$$E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} \frac{1}{2} \|\mathbf{y}^i - \mathbf{d}^i\|_2^2 = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} \frac{1}{2} \sum_k (y_k^i - d_k^i)^2$$

- Classification multi-classes :
  - Entropie croisée avec une couche de sortie softmax

$$\mathbf{d}^i \in \{1, \dots, K\}, \text{ codé par } \mathbf{d}^i \in \{0, 1\}^K$$

$$E(\mathbf{W}) = - \sum_{(\mathbf{x}^i, \mathbf{d}^i)} \sum_{k=1}^K d_k^i \log y_k^i \quad \text{avec} \quad \mathbf{y}^i = f(\mathbf{x}^i; \mathbf{W}) = \text{softmax}(\mathbf{a}^i) \in (0, 1)^K$$

# Optimisation

- Expression des fonctions de coût :

$$E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i)} L(\mathbf{y}^i; \mathbf{d}^i)$$

- Expression des gradients :

$$\nabla E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i)} \nabla L(\mathbf{y}^i; \mathbf{d}^i)$$

# Optimisation

- Gradient de  $L(\mathbf{y}; \mathbf{d})$  par rapport à  $\mathbf{y}$  :

- Cas de la régression

$$L(\mathbf{y}; \mathbf{d}) = \frac{1}{2} \|\mathbf{y} - \mathbf{d}\|_2^2 \quad \Rightarrow \quad \nabla_{\mathbf{y}} L(\mathbf{y}; \mathbf{d}) = \mathbf{y} - \mathbf{d}$$

- Cas de la classification

$$L(\mathbf{y}; \mathbf{d}) = - \sum_{k=1}^K d_k^i \log y_k^i \quad \Rightarrow \quad \nabla_{\mathbf{a}} L(\mathbf{y}; \mathbf{d}) = \mathbf{y} - \mathbf{d}$$

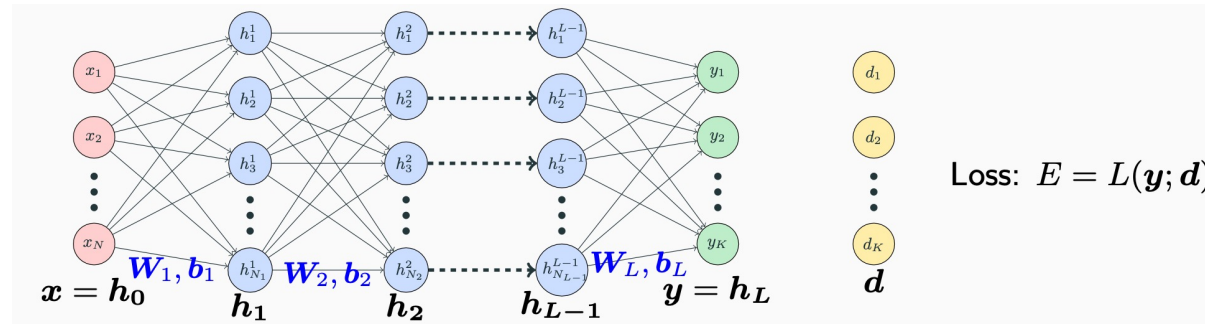
avec  $\mathbf{y} = \text{softmax}(\mathbf{a})$

# Optimisation

- On sait donc calculer le gradient de  $L(y; d)$  par rapport à  $y$  (dernière couche)
- Comment calculer les autres gradients pour les autres couches ?

$$\nabla_{w_k} L(\mathbf{y}; \mathbf{d}) \quad \text{and} \quad \nabla_{b_k} L(\mathbf{y}; \mathbf{d}) \quad \text{for } k = 0, \dots, L$$

# Rétropropagation de l'erreur



## Forward pass

Initialization:

$$h_0 = x$$

**for** layer  $k = 1$  **to**  $L$  **do**

Linear unit:

$$a_k = W_k h_{k-1} + b_k$$

Componentwise non-linear activation:

$$h_k = g_k(a_k)$$

**end**

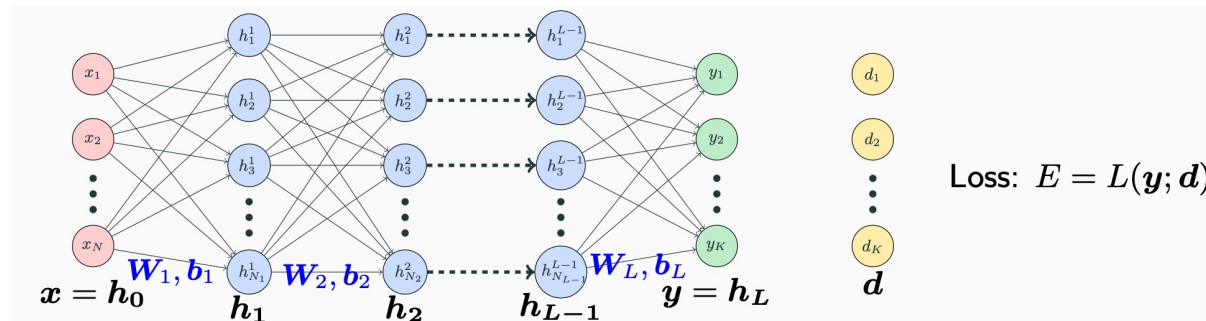
Output layer:

$$y = h_L$$

Compute loss:

$$E = L(y; d)$$

# Rétropropagation de l'erreur



## Forward pass

Initialization:

$$h_0 = x$$

**for** layer  $k = 1$  **to**  $L$  **do**

Linear unit:

$$a_k = \mathbf{W}_k \mathbf{h}_{k-1} + \mathbf{b}_k$$

Componentwise non-linear activation:

$$\mathbf{h}_k = g_k(a_k)$$

**end**

Output layer:

$$\mathbf{y} = \mathbf{h}_L$$

Compute loss:

$$E = L(\mathbf{y}; \mathbf{d})$$

## Backward pass

**Goal:** Compute the gradient with respect to all parameters

$$\frac{\partial E}{\partial w_{i,j}^k} = ? \quad \frac{\partial E}{\partial b_i^k} = ?$$

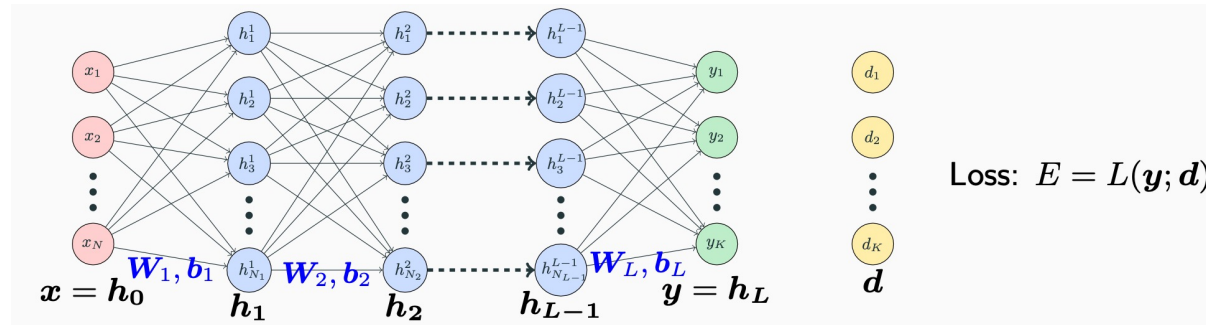
for all

$$k \in \{1, \dots, L\},$$

$$i \in \{1, \dots, N_k\},$$

$$j \in \{1, \dots, N_{k-1}\}.$$

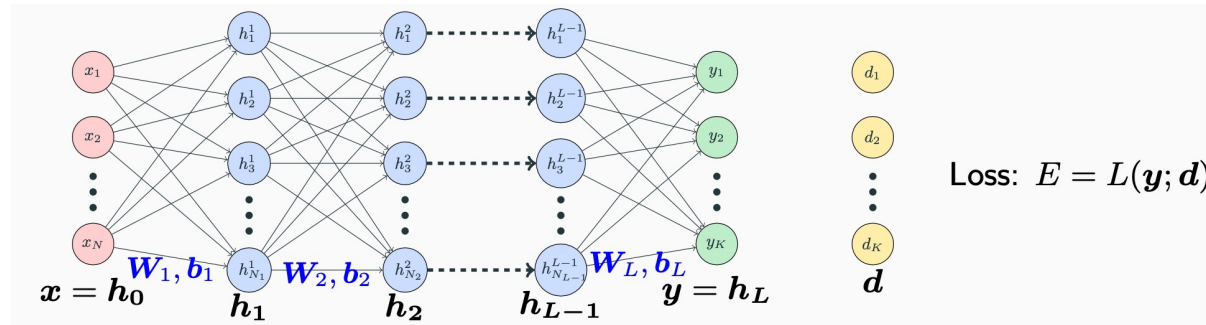
# Rétropropagation de l'erreur



- Passe arrière
  - On sait calculer la fonction de coût et son gradient

$$\nabla_{h_L} E = \nabla L(y; d)$$

# Rétropropagation de l'erreur



- Gradient par rapport à la dernière unité linéaire  $a_L$

$$h_L = g_L(a_L)$$

Pour tout  $i \in \{1, \dots, N_L\}$ ,  $h_i^L = g_L(a_i^L)$

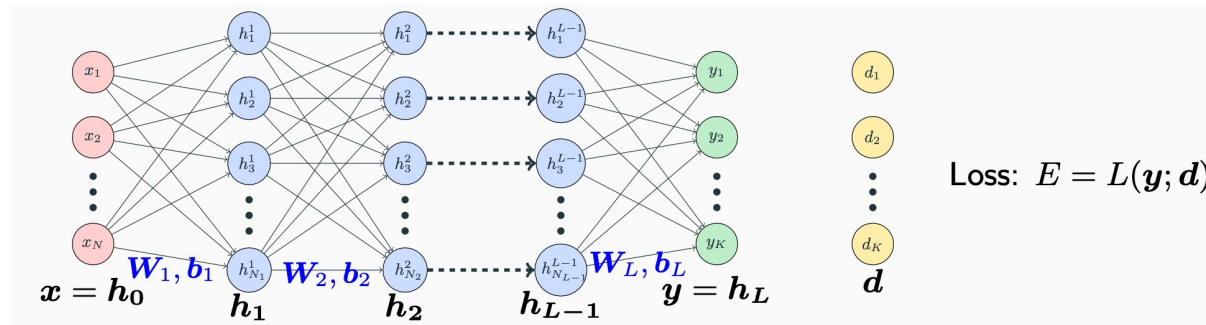
Chaînage des dérivées partielles :  $\frac{\partial E}{\partial a_i^L} = \frac{\partial E}{\partial h_i^L} \frac{\partial h_i^L}{\partial a_i^L} = [\nabla_{h_L} E]_i g'_L(a_i^L)$

- Formulation matricielle :  $\nabla_{a_L} E = \nabla_{h_L} E \odot g'_L(a_L)$

Où  $\odot$  correspond au produit terme à terme entre deux matrices



# Rétropropagation de l'erreur



- Gradient par rapport au biais  $b_L$  de la dernière unité linéaire

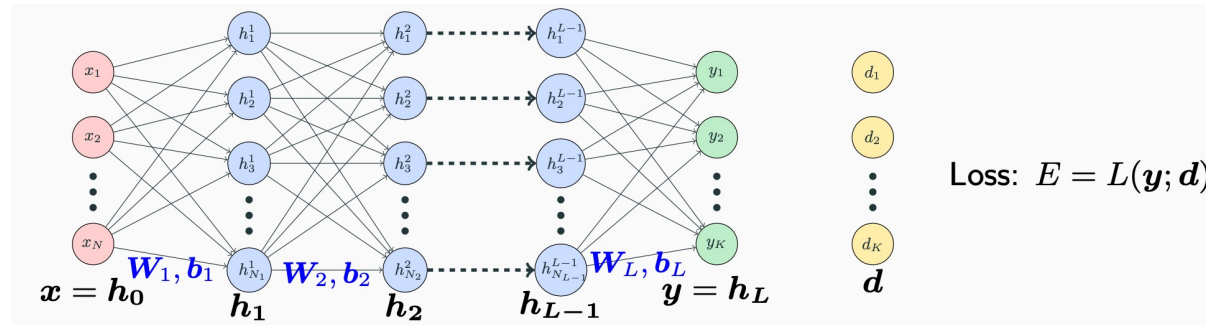
$$\mathbf{a}_L = \mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L$$

Pour tout  $i \in \{1, \dots, N_L\}$ ,  $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$

Chaînage des dérivées partielles :  $\frac{\partial E}{\partial b_i^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial b_i^L}}_{=1} = \frac{\partial E}{\partial a_i^L} = [\nabla_{\mathbf{a}_L} E]_i$

- Formulation matricielle :  $\nabla_{\mathbf{b}_L} E = \nabla_{\mathbf{a}_L} E$

# Rétropropagation de l'erreur



- Gradient par rapport aux poids  $W_L$  de la dernière unité linéaire

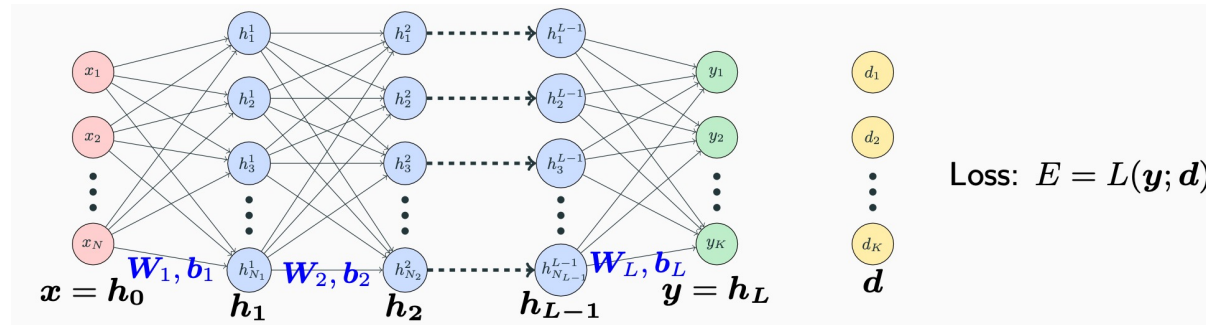
$$a_L = W_L h_{L-1} + b_L$$

$$\text{Pour tout } i \in \{1, \dots, N_L\}, a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$$

$$\text{Chaînage des dérivées partielles : } \frac{\partial E}{\partial w_{i,j}^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial w_{i,j}^L}}_{=h_j^{L-1}} = \frac{\partial E}{\partial a_i^L} h_j^{L-1} = [\nabla_{a_L} E]_i [h_{L-1}]_j$$

- Formulation matricielle :  $\nabla_{W_L} E = \nabla_{a_L} E h_{L-1}^T$

# Rétropropagation de l'erreur



## ■ Gradients pour les paramètres de la dernière couche

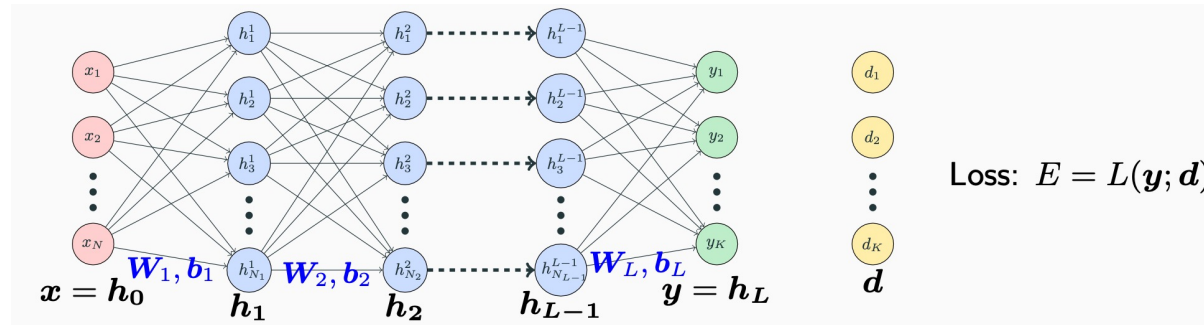
- Etant donné le gradient par rapport à la sortie de la dernière couche :

$$\nabla_{h_L} E$$

- On peut donc calculer :

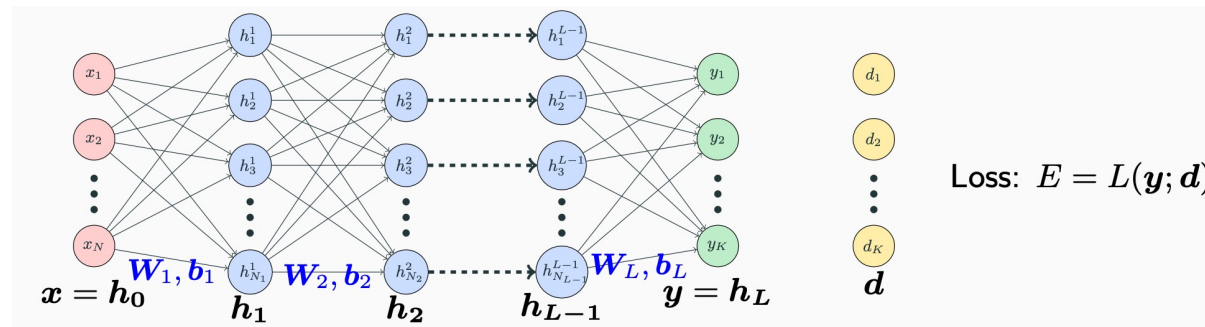
- $\nabla_{a_L} E = \nabla_{h_L} E \odot g'_L(a_L)$
- $\nabla_{b_L} E = \nabla_{a_L} E$
- $\nabla_{W_L} E = \nabla_{a_L} E h_{L-1}^T$

# Rétropropagation de l'erreur



- Comment maintenant calculer les gradients pour les paramètres de l'avant dernière couche  $L-1$  ?

# Rétropropagation de l'erreur

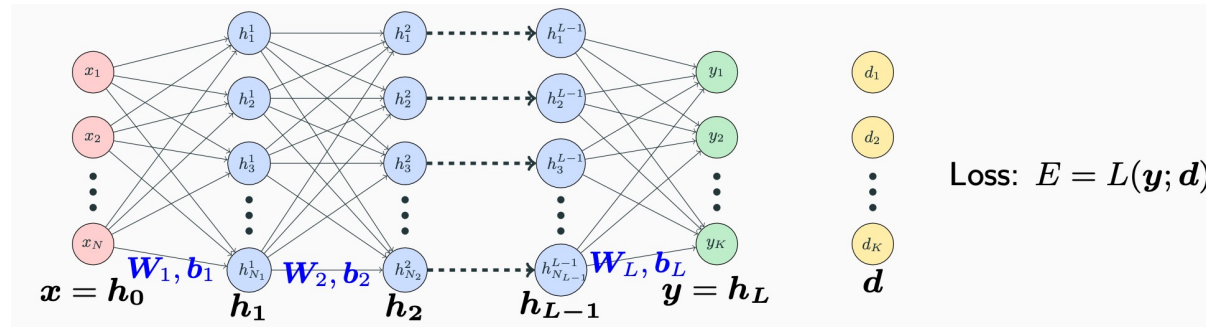


- Comment maintenant calculer les gradients pour les paramètres de l'avant dernière couche  $L-1$  ?

➔ il faut avoir le gradient en fonction de l'avant dernière couche cachée  $h_{L-1}$  et appliquer les mêmes formules

$$\nabla_{h_{L-1}} E = ?$$

# Rétropropagation de l'erreur



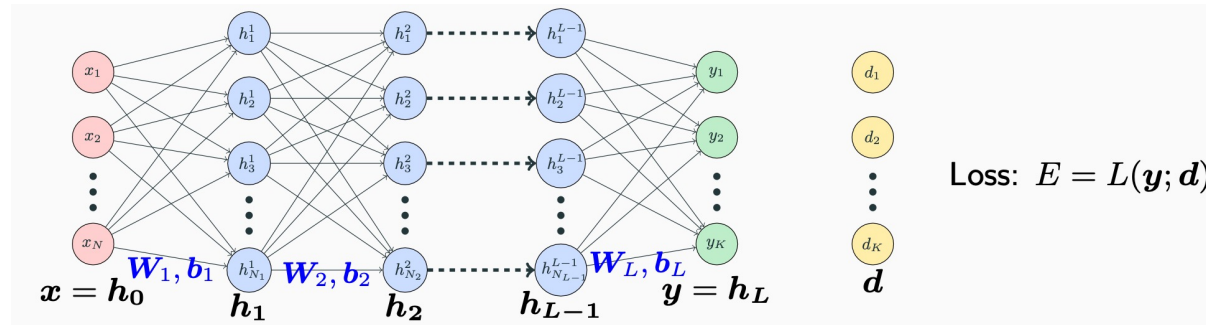
- Gradient par rapport à l'avant dernière couche cachée  $h_{L-1}$

On a pour tout  $i \in \{1, \dots, N_L\}$ ,  $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$

Et on souhaite calculer :

$$\frac{\partial E}{\partial h_j^{L-1}}$$

# Rétropropagation de l'erreur



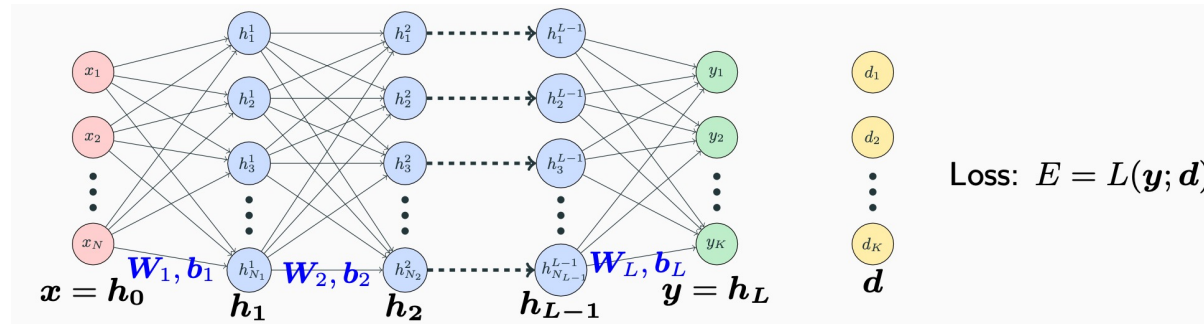
- Gradient par rapport à l'avant dernière couche cachée  $h_{L-1}$ 
  - Règle de calcul des dérivées pour les compositions de fonctions affines :

Pour :  $\varphi(x) = f(Ax + b)$  on a :  $\nabla \varphi(x) = A^T \nabla f(Ax + b)$

- Ce qui donne :  $\begin{array}{ccc} \mathbb{R}^{N_{L-1}} & \rightarrow & \mathbb{R}^{N_L} \rightarrow \mathbb{R} \\ h_{L-1} & \mapsto & a_L = W_L h_{L-1} + b_L \mapsto E \end{array}$

- Formulation matricielle :  $\nabla_{h_{L-1}} E = W_L^T \nabla_{a_L} E$

# Rétropropagation de l'erreur



## Forward pass

Initialization:

$$h_0 = x$$

for layer  $k = 1$  to  $L$  do

Linear unit:

$$a_k = W_k h_{k-1} + b_k$$

Componentwise non-linear activation:

$$h_k = g_k(a_k)$$

end

Output layer:

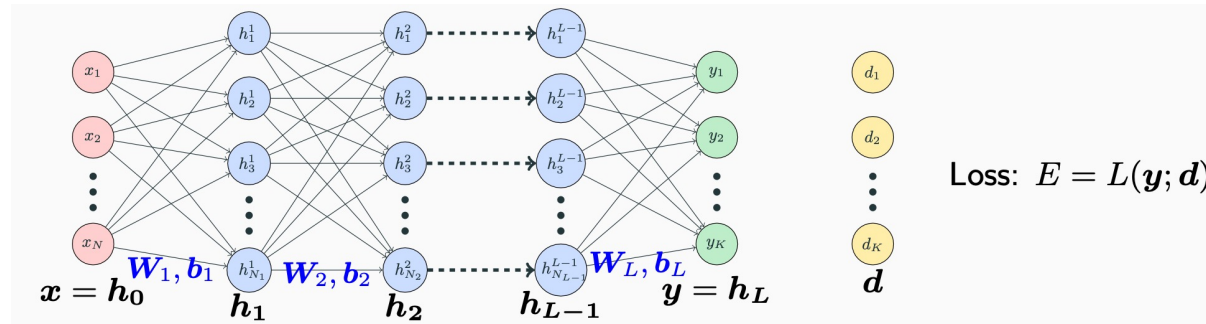
$$y = h_L$$

Compute loss:

$$E = L(y; d)$$



# Rétropropagation de l'erreur



## Forward pass

Initialization:

$$h_0 = x$$

for layer  $k = 1$  to  $L$  do

Linear unit:

$$a_k = W_k h_{k-1} + b_k$$

Componentwise non-linear activation:

$$h_k = g_k(a_k)$$

end

Output layer:

$$y = h_L$$

Compute loss:

$$E = L(y; d)$$

## Backward pass

Initialization: Gradient of output layer:

$$\nabla_{h_L} E = \nabla L(y; d)$$

for layer  $k = L$  to 1 do

Componentwise gain of error:

$$\delta_k = \nabla_{a_k} E = \nabla_{h_k} E \odot g'_k(a_k)$$

Gradient of layer bias:

$$\nabla_{b_k} E = \delta_k$$

Gradient of weights:

$$\nabla_{W_k} E = \delta_k h_{k-1}^T$$

Gradient of previous hidden layer:

$$\nabla_{h_{k-1}} E = W_k^T \delta_k$$

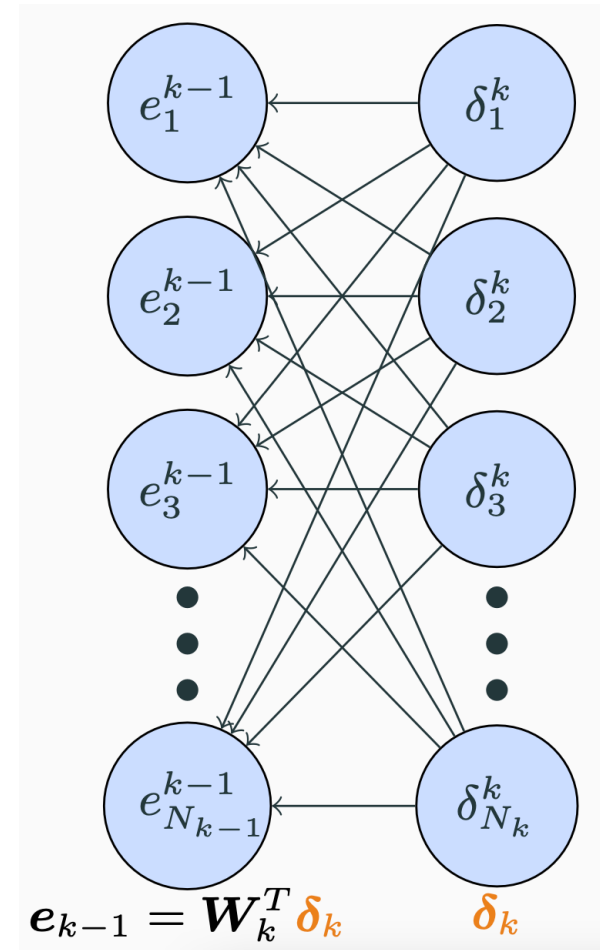
end

# Rétropropagation de l'erreur

- Gradient de la couche cachée précédente

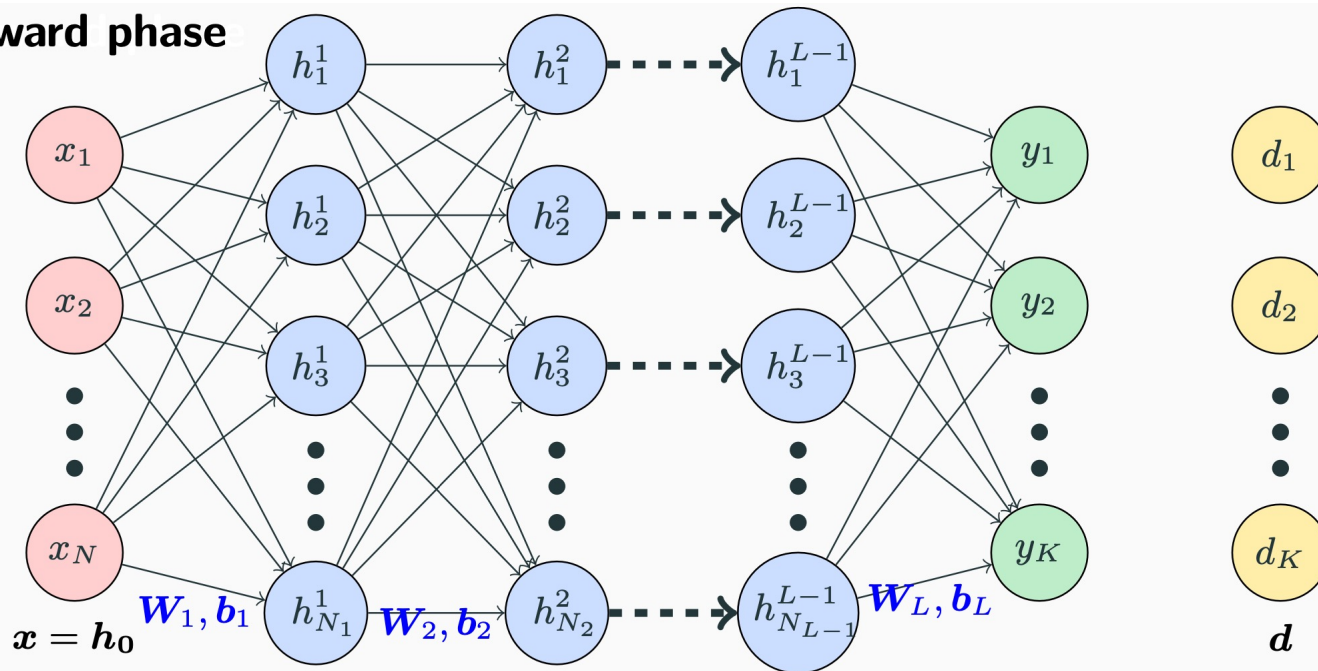
$$e_{k-1} = \nabla_{h_{k-1}} E = W_k^T \delta_k$$

- Multiplier par  $W_k^T$  revient à propager l'erreur dans la couche précédente
- L'erreur est rétropropagée de couche en couche afin de calculer le gradient de la fonction de coût par rapport aux paramètres de chaque couche



# Rétropropagation de l'erreur

Forward phase



$$E = L(y, d)$$

Input Layer

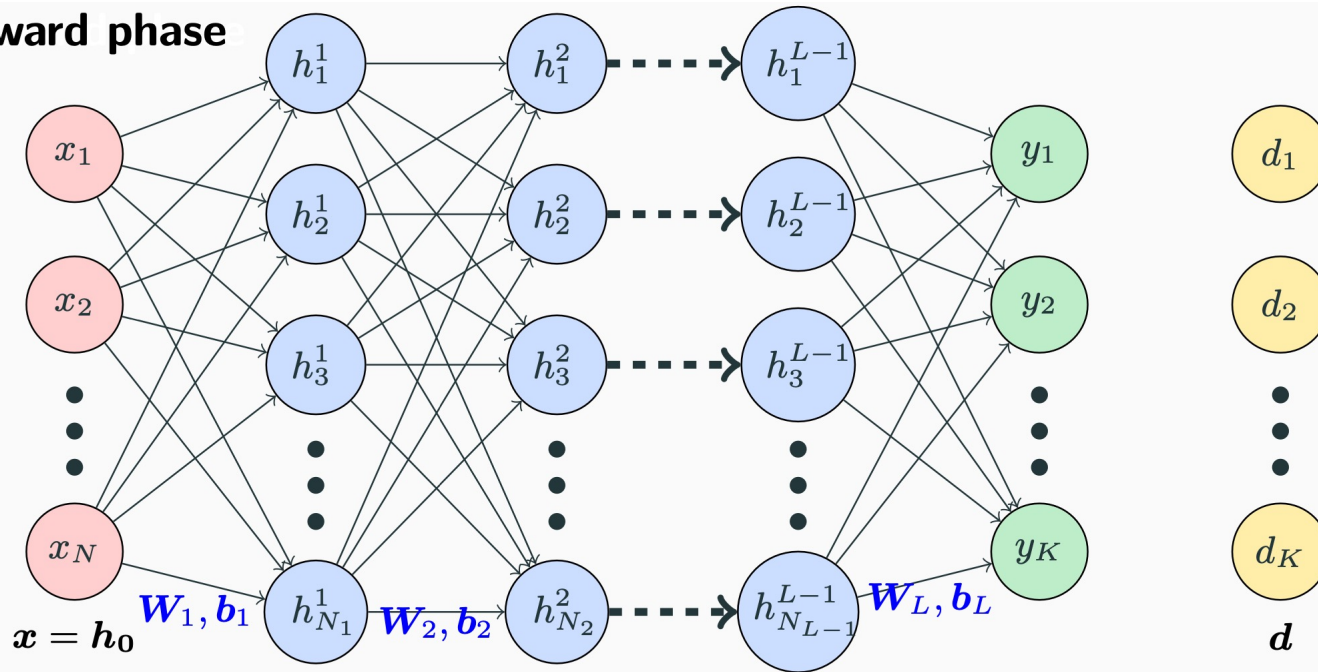
Hidden Layers

Output Layer

Label

# Rétropropagation de l'erreur

Forward phase



Input Layer

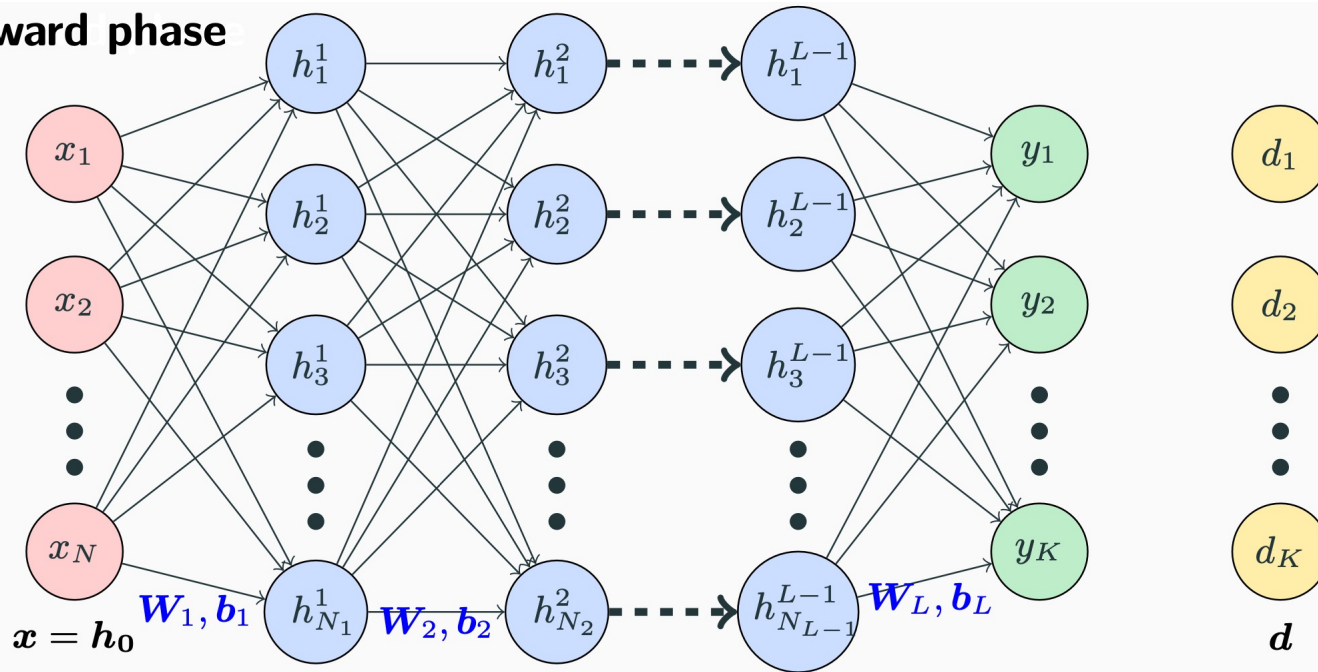
Hidden Layers

Output Layer

Label

# Rétropropagation de l'erreur

Forward phase



$$a_1 = W_1 h_0 + b_1 \quad a_2$$

$$h_1 = g_1(a_1) \quad h_2$$

$$E = L(y, d)$$

Input Layer

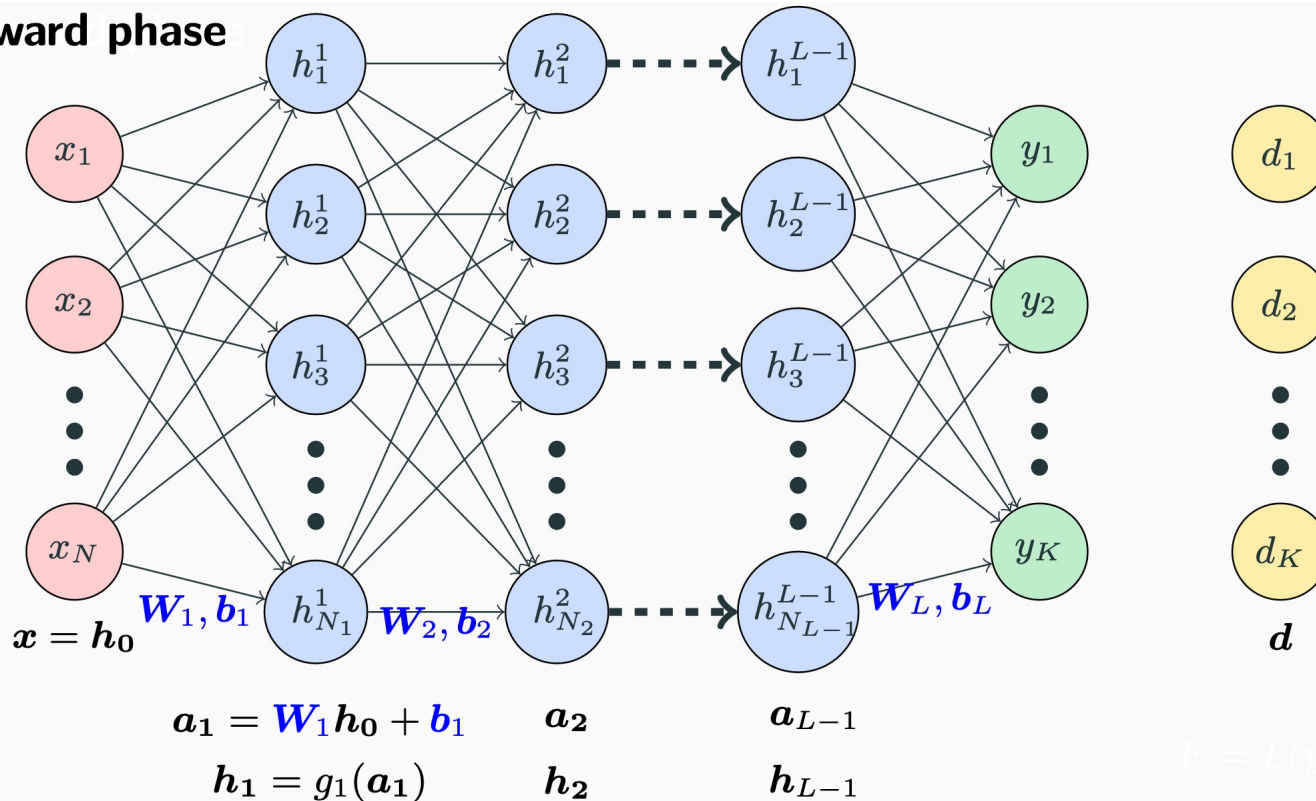
Hidden Layers

Output Layer

Label

# Rétropropagation de l'erreur

Forward phase



Input Layer

Hidden Layers

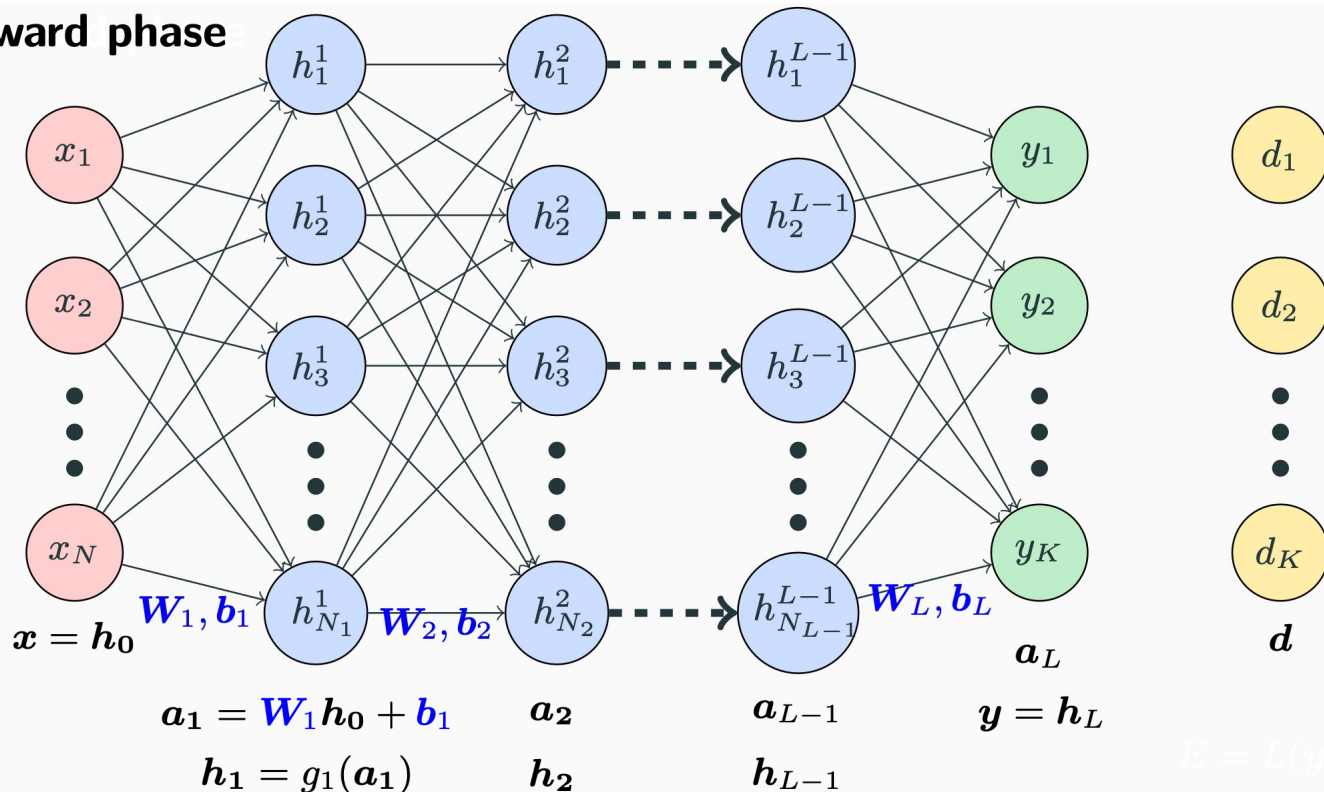
Output Layer

Label



# Rétropropagation de l'erreur

Forward phase



Input Layer

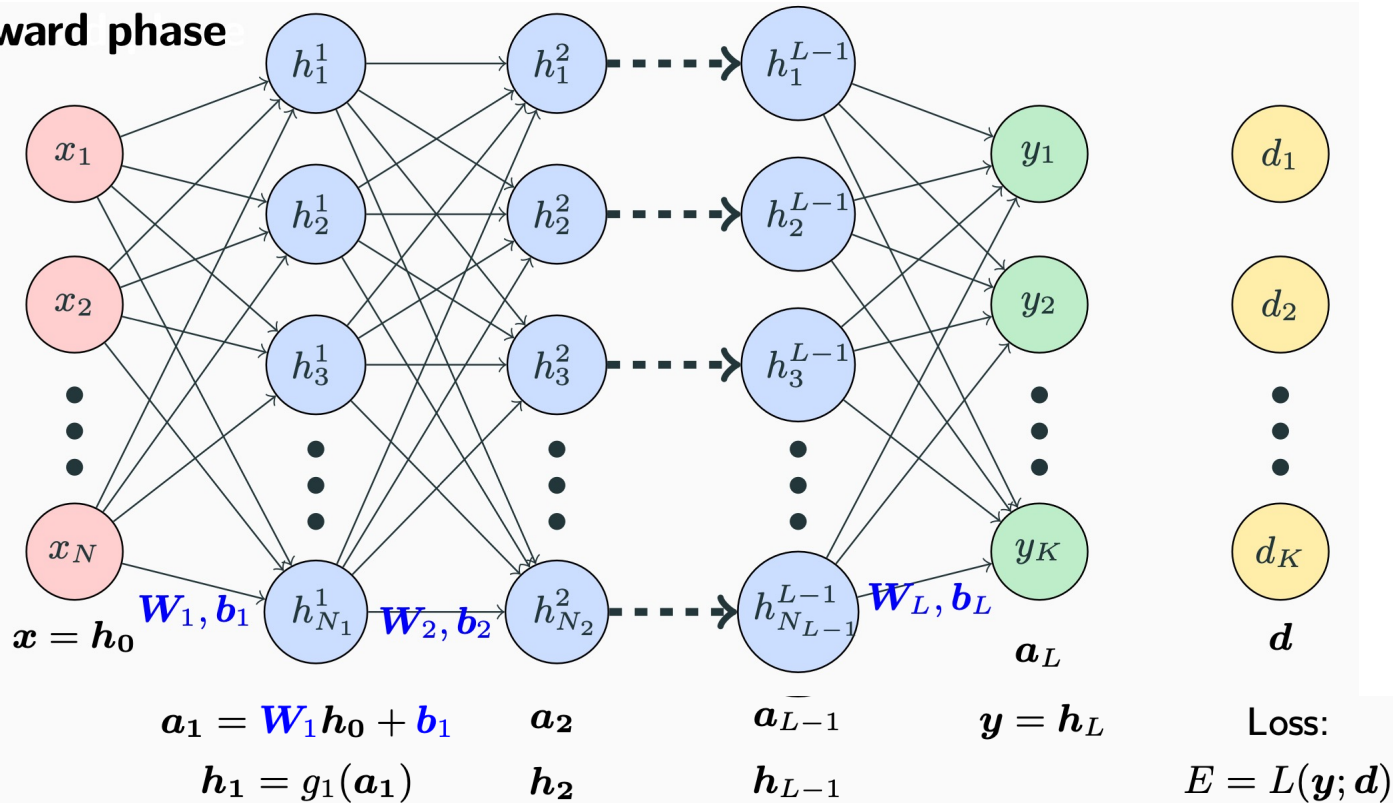
Hidden Layers

Output Layer

Label

# Rétropropagation de l'erreur

Forward phase



Input Layer

Hidden Layers

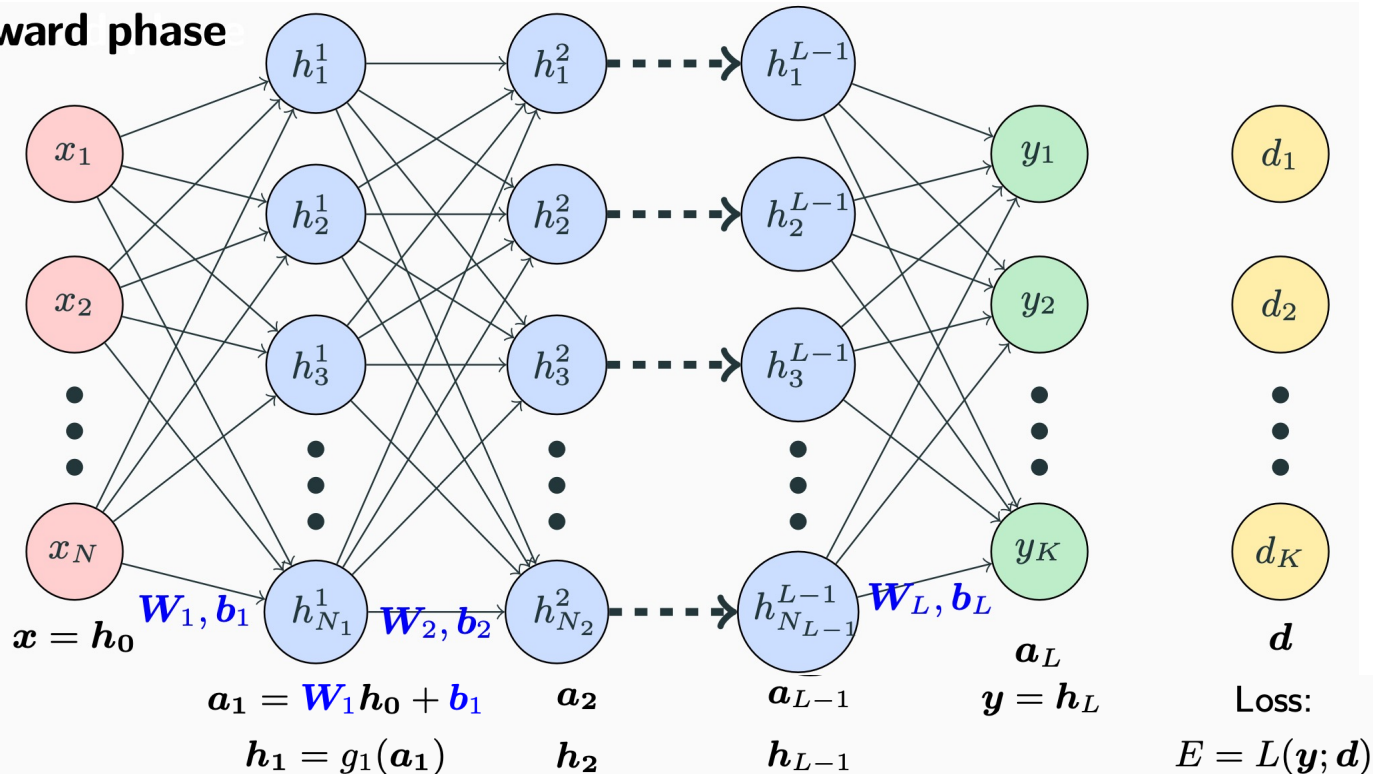
Output Layer

Label



# Rétropropagation de l'erreur

Forward phase



$$e_L = \nabla L(y; d)$$

$$\delta_L$$

Input Layer

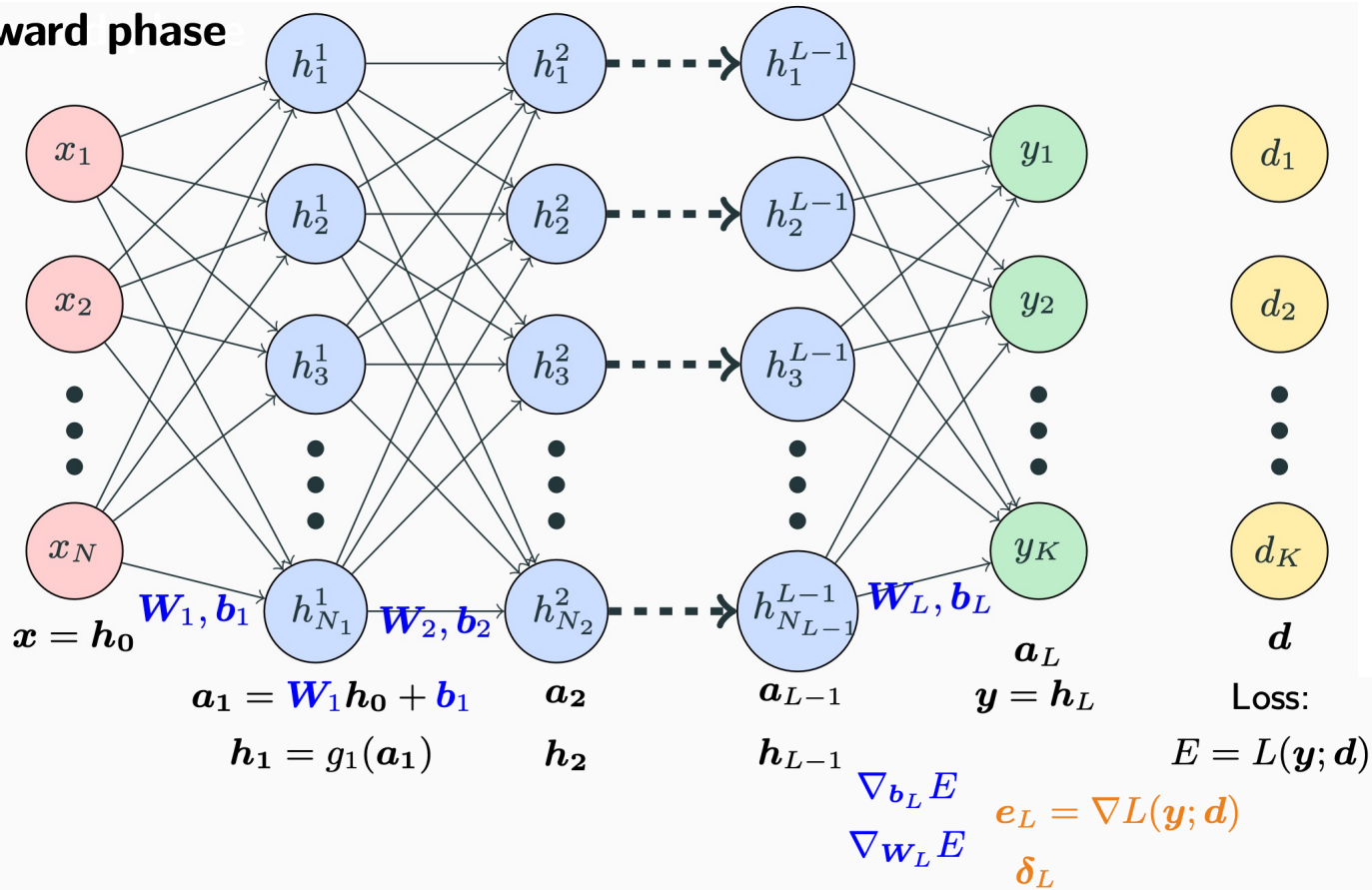
Hidden Layers

Output Layer

Label

# Rétropropagation de l'erreur

Forward phase



Input Layer

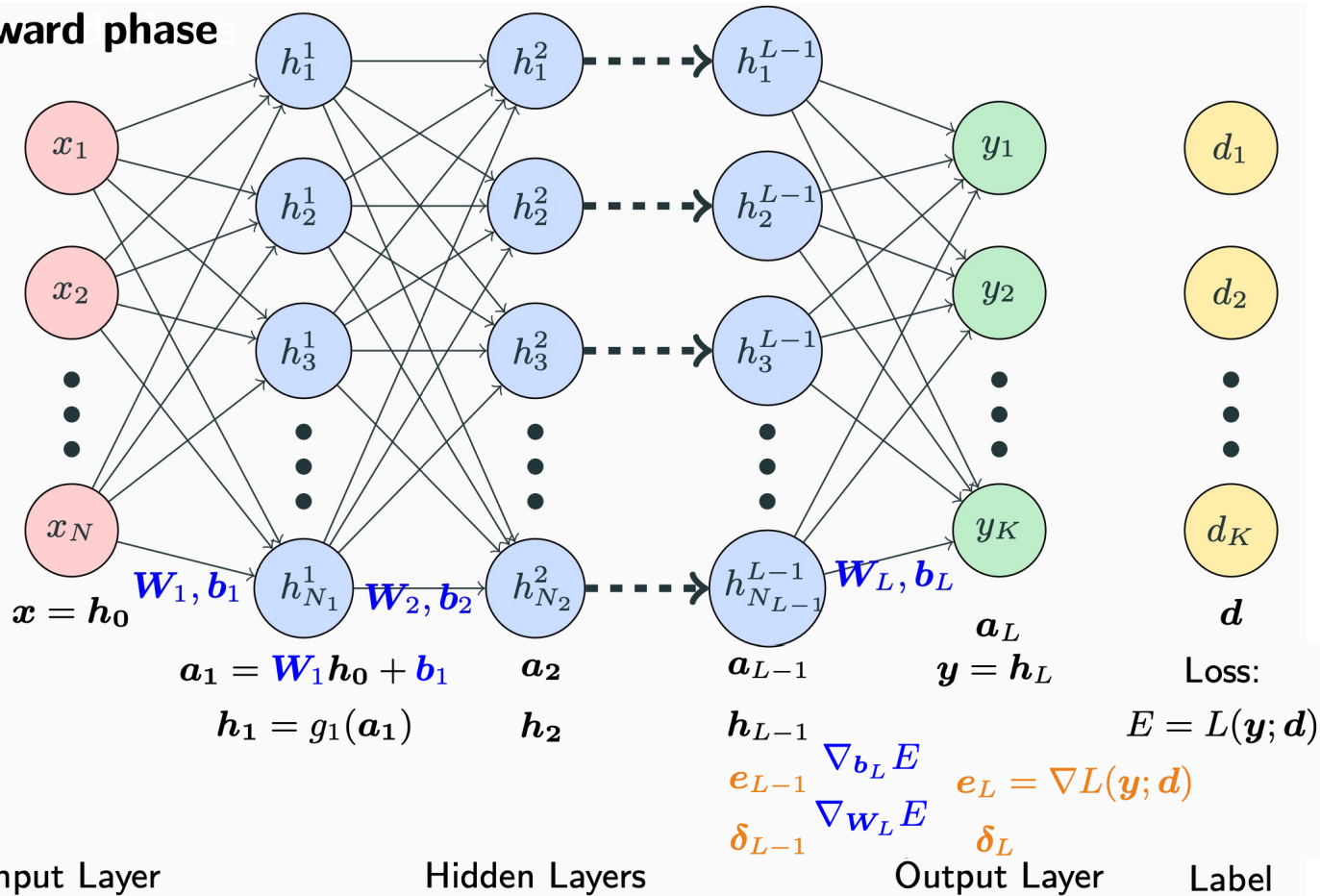
Hidden Layers

Output Layer

Label

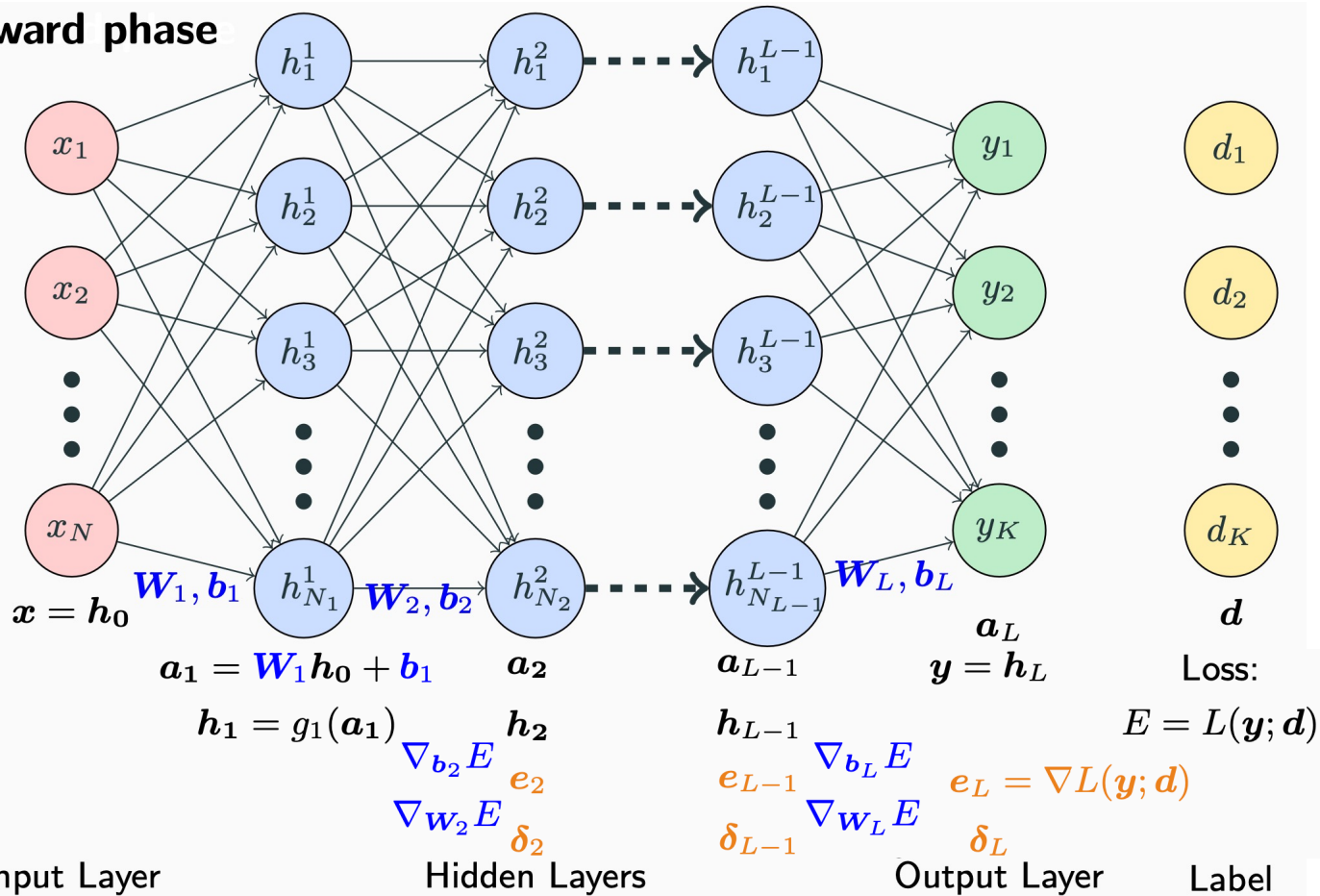
# Rétropropagation de l'erreur

Forward phase



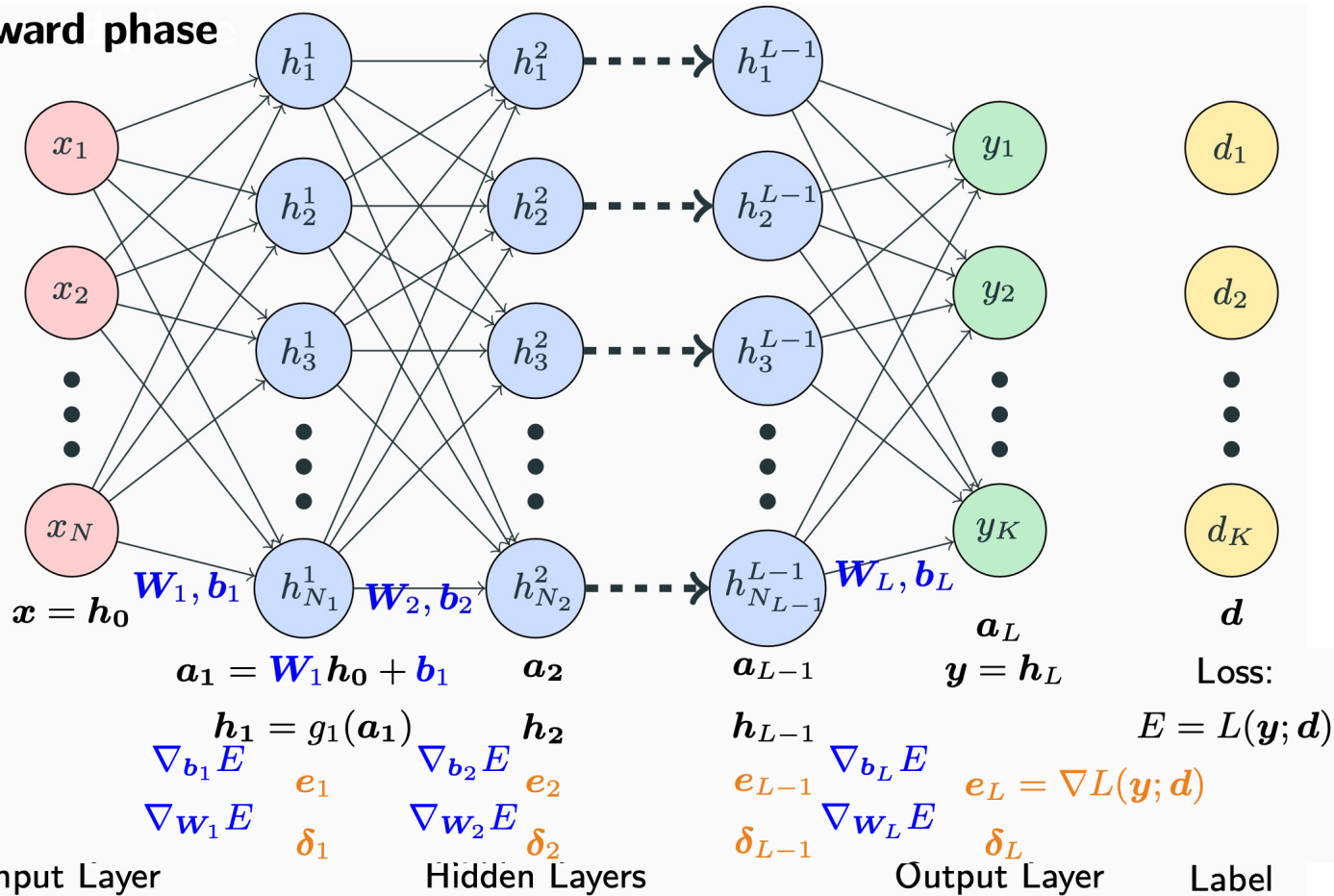
# Rétropropagation de l'erreur

Forward phase



# Rétropropagation de l'erreur

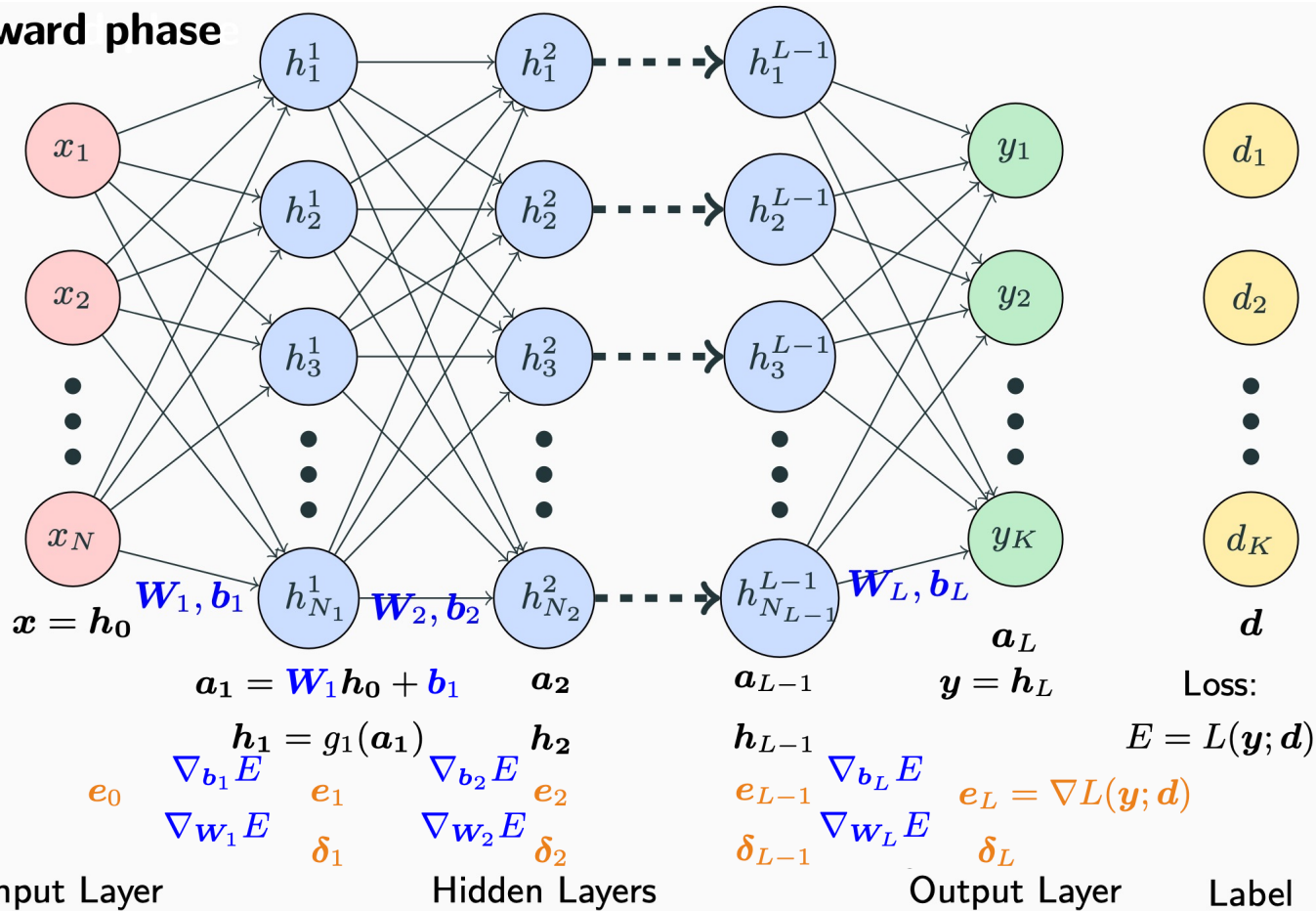
Forward phase





# Rétropropagation de l'erreur

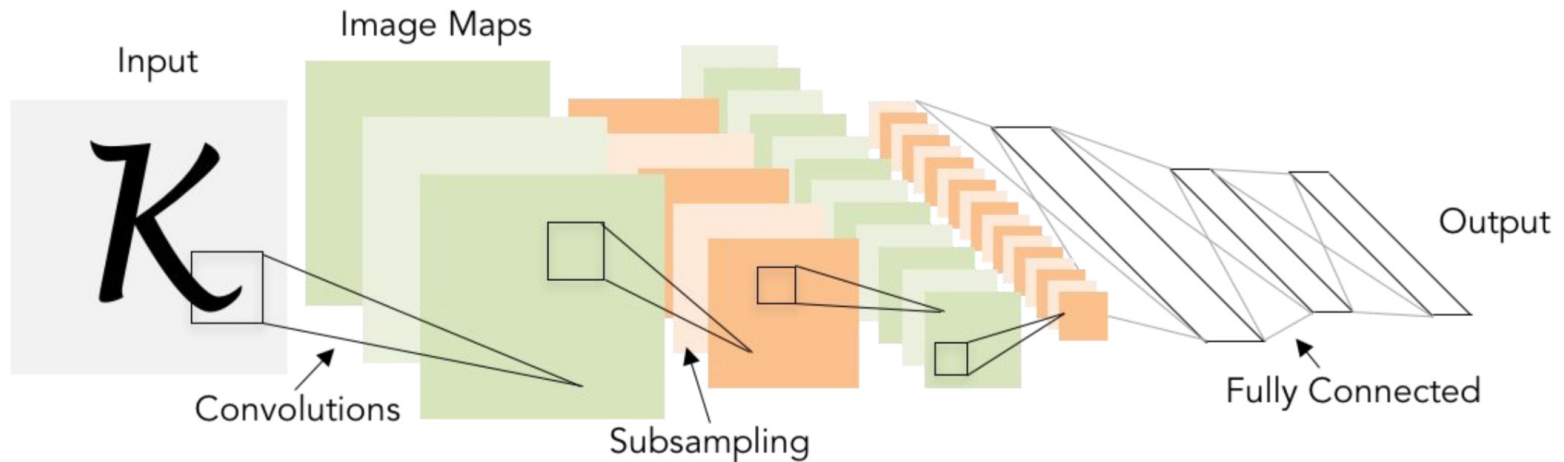
Forward phase



# Réseaux de neurones convolutifs

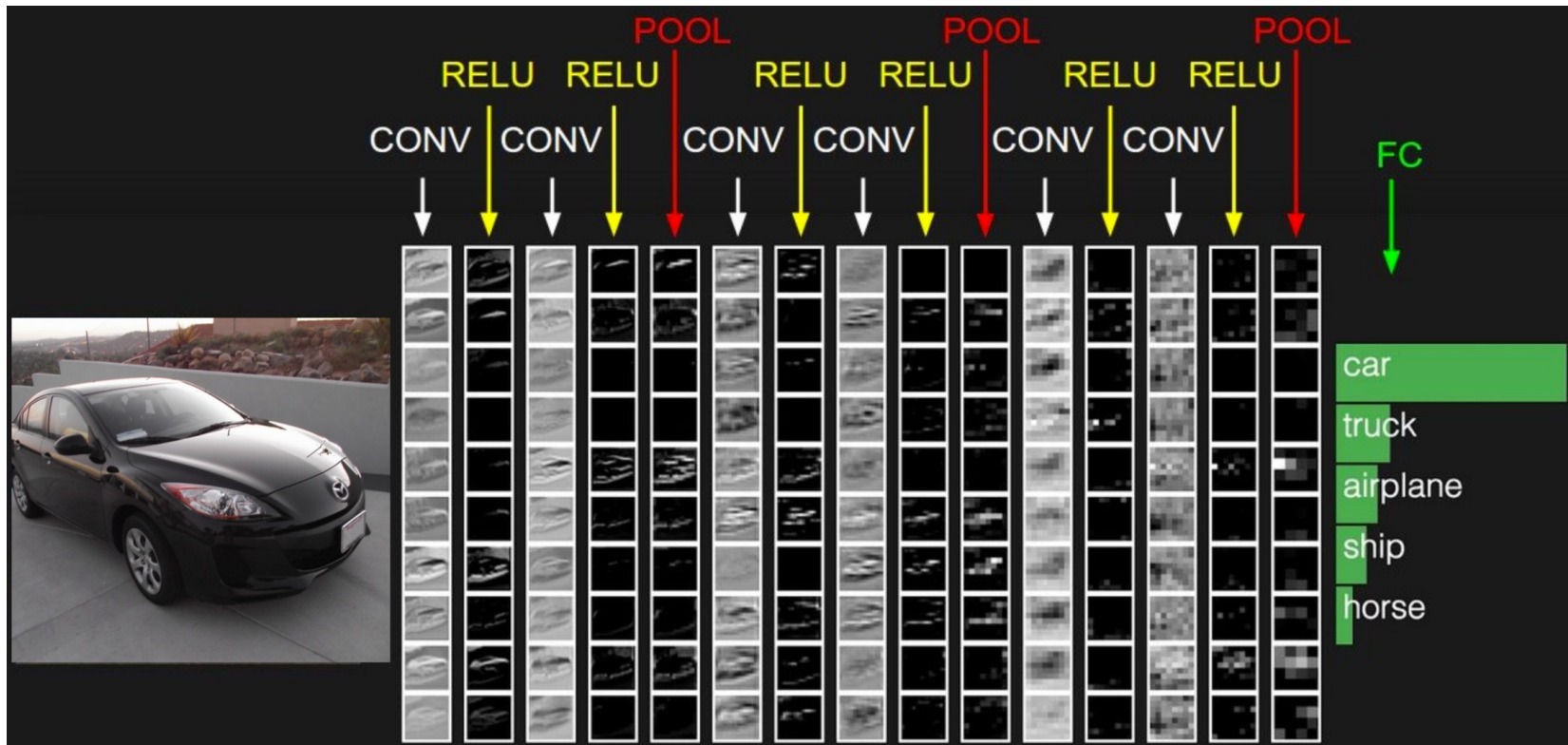
- LeNet

[LeCun et al., 1998]



Conv filters were 5x5, applied at stride 1  
Subsampling (Pooling) layers were 2x2 applied at stride 2  
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

# Réseaux de neurones convolutifs





# ImageNet

## IMAGENET Large Scale Visual Recognition Challenge

Steel drum

The Image Classification Challenge:

1,000 object classes

1,431,167 images



**Output:**

Scale

T-shirt

Steel drum

Drumstick

Mud turtle



**Output:**

Scale

T-shirt

Giant panda

Drumstick

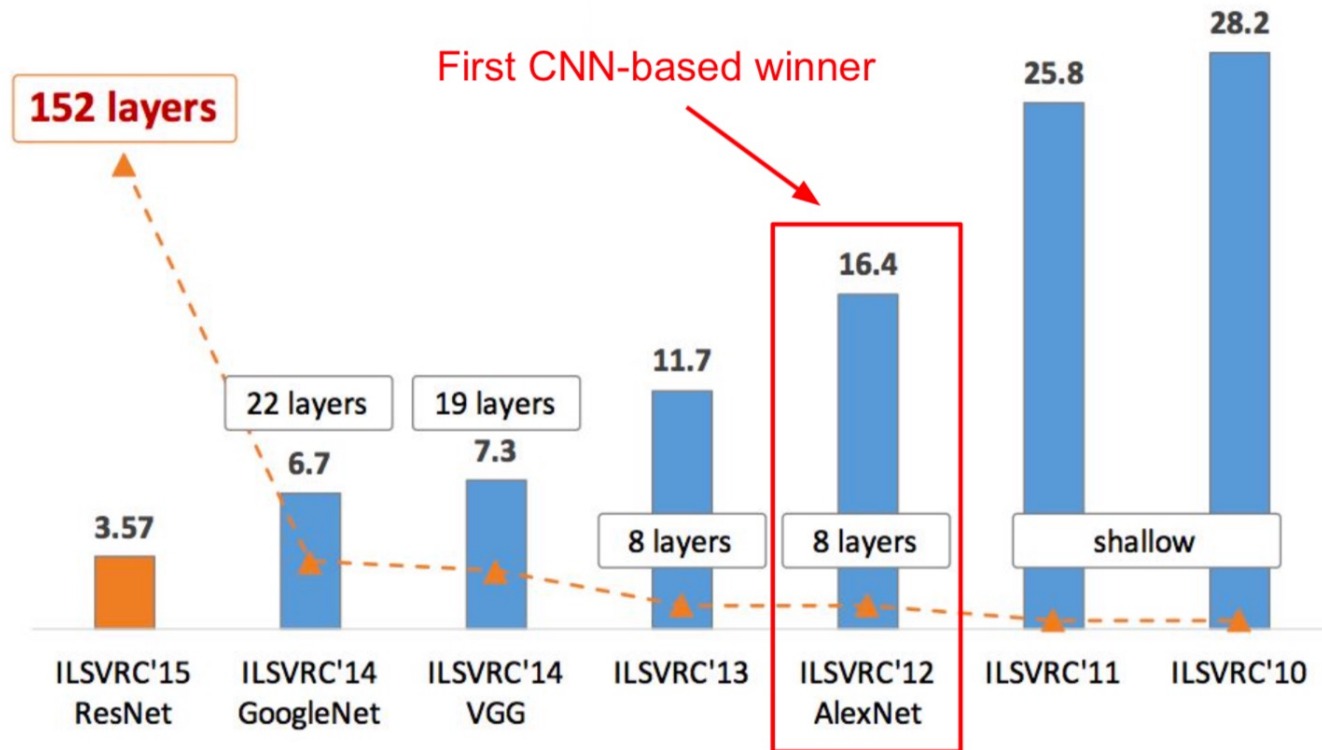
Mud turtle



Russakovsky et al. arXiv, 2014

# ImageNet Large Scale Visual Recognition Challenge (LSVRC)

- Les gagnants



# AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0

[27x27x96] **MAX POOL1**: 3x3 filters at stride 2

[27x27x96] **NORM1**: Normalization layer

[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2

[13x13x256] **MAX POOL2**: 3x3 filters at stride 2

[13x13x256] **NORM2**: Normalization layer

[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1

[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1

[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)

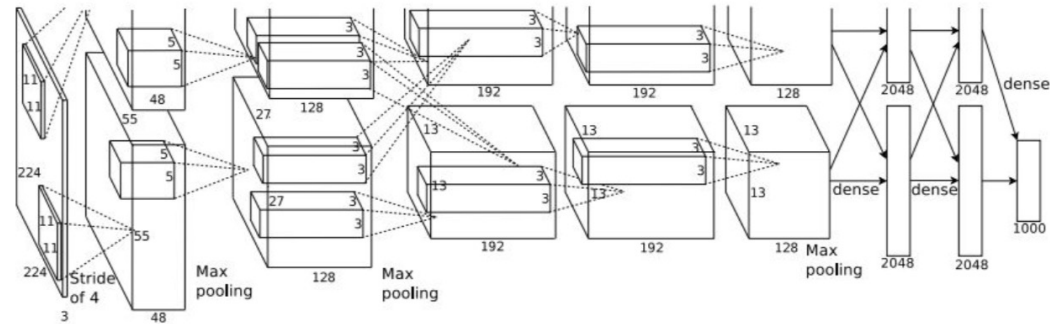
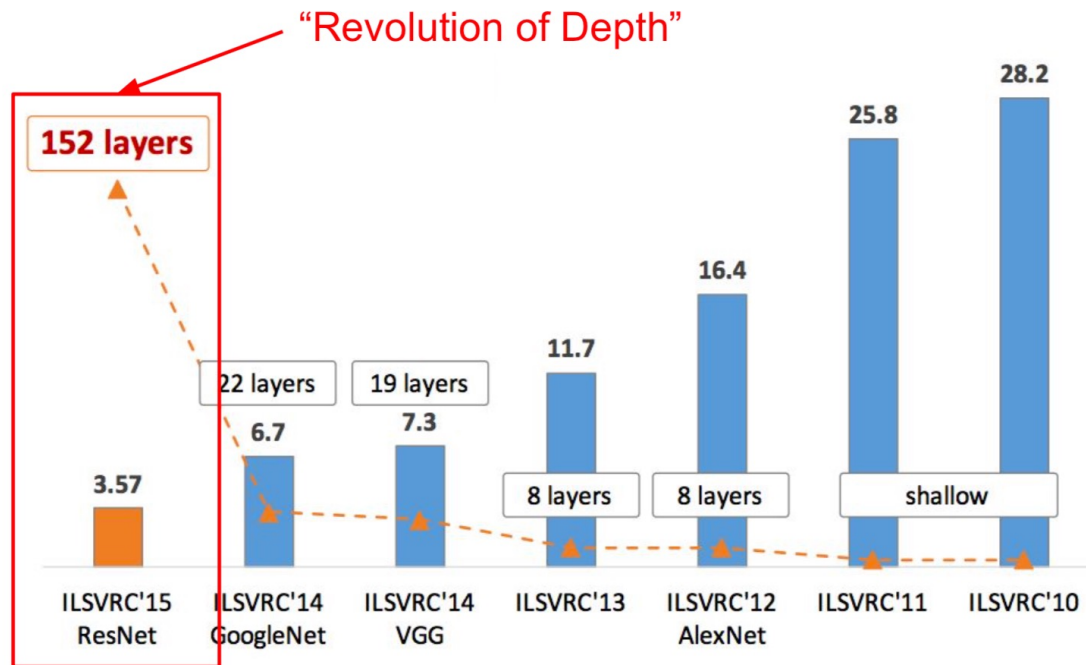


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# ImageNet Large Scale Visual Recognition Challenge (LSVRC)

- Les gagnants

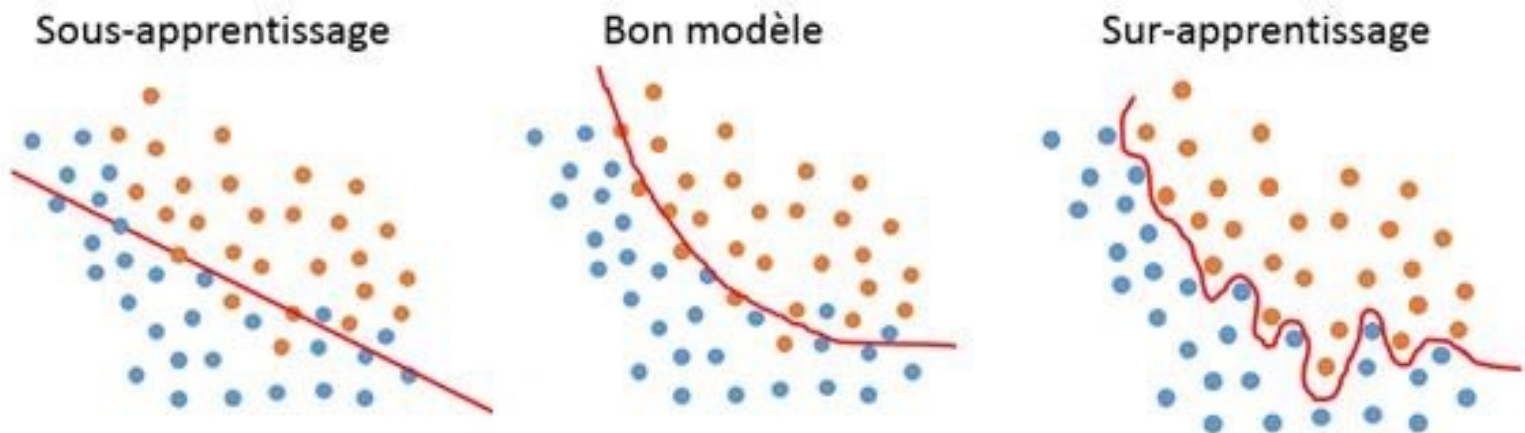




# Généralisation

→ Capacité d'un modèle ayant été entraîné sur un certain jeu de données à réaliser des prédictions précises sur un autre jeu de données (distributions potentiellement différentes)

■ Problèmes du sous-apprentissage et du sur-apprentissage:



# Sous-apprentissage

- Cause :

Le modèle n'arrive pas à capturer les corrélations (potentiellement complexes) entre les entrées et les sorties du système

On dit que le modèle a un biais important

- ➔ Solution :

- Utiliser un modèle plus complexe

# Sur-apprentissage

## ■ Cause :

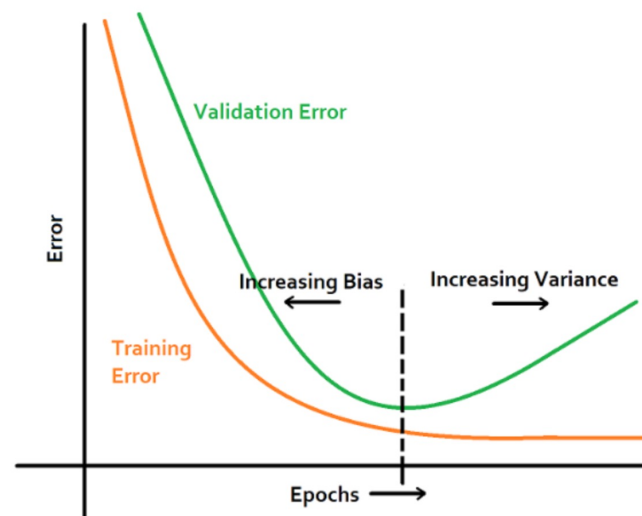
- Le modèle est trop sensible au bruit présent dans les données d'apprentissage
- On dit que le modèle a une variance importante

## ➔ Solutions :

- Augmenter la quantité et la variété des données d'apprentissage
- Réduire la complexité du système (nombre de paramètres)
  - ➔  $\text{nb\_ex\_apprentissage} \geq 3 * \text{nb\_params}$
- Early stopping
- Dropout
- Régularisation

# Early stopping

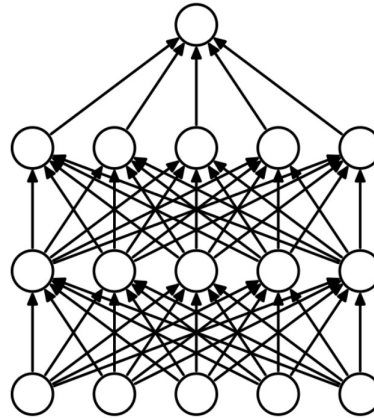
- Utilisation d'un ensemble de données de validation :
  - L'ensemble des données est divisé en trois sous-ensembles disjoints :
    - Apprentissage : utilisé apprendre les valeurs des paramètres
    - Validation : utilisé pour contrôler le sur-apprentissage
    - Test : utilisé pour évaluer le système sur de nouvelles données



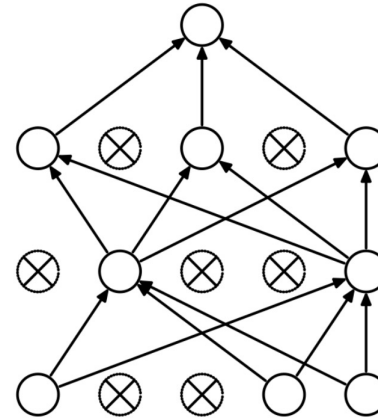


# Dropout

- Désactivation temporaire aléatoire de neurones pendant l'apprentissage



Réseau de Neurones Standard



Après avoir appliqué le Dropout

- Génération de modèles différents à chaque itération de l'apprentissage
- ➔ Améliore la robustesse en perturbant les caractéristiques apprises par le modèle

# Régularisation

- Ajout d'un terme représentant la complexité du modèle à la fonction de coût

$$J_{\lambda}(\theta) = J(\theta) + \lambda R(\theta) \quad \text{avec } \lambda \text{ le paramètre de régularisation}$$

- Objectif :

Réduire autant que possible le nombre de paramètres influents (simplifier le modèle) en rendant la valeur d'un certain nombre de paramètres proche de 0

- Exemple pour la régression linéaire

$$J_{\lambda}(\theta) = \frac{1}{2N} \sum_{i=1}^N (f_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^d \theta_j^2$$

# Découpage de l'ensemble des données

- **Sous-ensembles fixes :**

- Découpage aléatoire en trois sous-ensembles disjoints fixes
  - Apprentissage (par ex 80 %)
  - Validation (par ex 10%)
  - Test (par ex 10 %)

- **Validation croisée (k-fold cross-validation) :**

- Découpage aléatoire en k sous-ensembles (par ex k=10)
- Sélectionner 1 pour le test et les autres pour l'apprentissage (+validation) et répéter l'opération k fois
- Moyenner les performances sur les k expériences



ÉCOLE  
**CENTRALE** LYON

36, avenue Guy de Collongue 69130 Écully - France  
+33 (0)4 72 18 60 00

[www.ec-lyon.fr](http://www.ec-lyon.fr)