

**FORMATION IA LEGERE**  
**Deep Learning pour l'embarqué et les systèmes légers**  
**Selva Systems**

28 juin 2025

## Table des matières

<b>1 Introduction</b>	<b>5</b>
1.1 L'apprentissage automatique : intuition et motivation . . . . .	5
1.2 Les premiers neurones artificiels . . . . .	5
1.3 Le Perceptron et la naissance de l'apprentissage . . . . .	6
1.4 Limites du modèle linéaire et premier hiver de l'IA . . . . .	6
1.5 Le perceptron multicouches et la rétropagation . . . . .	7
1.6 L'essor du Deep Learning moderne . . . . .	8
1.7 Approche pédagogique de cette formation . . . . .	8
<b>2 Le Perceptron et l'apprentissage supervisé</b>	<b>8</b>
2.1 La fonction sigmoïde . . . . .	9
2.2 La fonction coût log loss . . . . .	9
2.3 Descente de gradient . . . . .	10
2.4 Evaluation des gradients et mise à jour des poids neuronaux . . . . .	10
2.4.1 Définitions . . . . .	11
2.4.2 Dérivée partielle de la fonction de cout . . . . .	11
2.5 Exercice : mise à jour du poids dans un neurone sigmoïde . . . . .	12
<b>3 Vectorisation des équations</b>	<b>12</b>
3.1 Pourquoi vectoriser ? . . . . .	12

3.2	Vectorisation du modèle . . . . .	13
3.2.1	Calcul du vecteur $Z$ (sortie linéaire) . . . . .	13
3.2.2	Activation sigmoïde vectorisée . . . . .	13
3.2.3	Fonction de coût vectorisée . . . . .	13
3.2.4	Gradient vectorisé . . . . .	14
3.2.5	Mise à jour des paramètres . . . . .	14
3.3	Extension à $n$ variables . . . . .	14
3.4	Intérêt de la vectorisation . . . . .	14
<b>4</b>	<b>Exercices guidés</b>	<b>15</b>
4.1	Implémenter un neurone artificiel en Python (vectorisé) . . . . .	15
4.2	Les chats et les chiens! . . . . .	17
4.2.1	Diagnostiquer et stabiliser l'apprentissage . . . . .	17
4.2.2	Remise en contexte : préparer le <i>dataset</i> . . . . .	18
4.2.3	Débordements exponentiels et logarithmiques . . . . .	18
4.2.4	Normaliser les données . . . . .	19
4.2.5	Impact sur la descente de gradient . . . . .	19
4.2.6	Réglage du taux d'apprentissage . . . . .	19
4.2.7	Déetecter le sur-apprentissage . . . . .	20
4.2.8	Progress bar et bonnes pratiques de code . . . . .	20
4.2.9	Vers un réseau multicouche . . . . .	20
<b>5</b>	<b>Les réseaux de neurones multicouches : puissance et rigueur du calcul matriciel</b>	<b>21</b>
5.1	Pourquoi les réseaux multicouches ? Limites de la linéarité . . . . .	21
5.2	De l'individu à la foule : la logique matricielle . . . . .	21
5.3	Propagation avant : l'art du calcul matriciel . . . . .	22
5.4	Superposition, abstraction, profondeur . . . . .	22
5.5	L'entraînement : rétropropagation matricielle . . . . .	22
5.6	Synthèse : penser en dimensions, visualiser en matrices . . . . .	23

<b>6 Forward Propagation dans un réseau à deux couches</b>	<b>24</b>
6.1 Schéma général du réseau . . . . .	24
6.2 Étape 1 : calcul des activations de la première couche . . . . .	24
6.3 Étape 2 : calcul des activations de la couche de sortie . . . . .	25
6.4 Étape 3 : calcul de la fonction coût . . . . .	25
6.5 Synthèse des dimensions . . . . .	25
6.6 Interprétation intuitive . . . . .	26
<b>7 Les réseaux de neurones convolutif</b>	<b>26</b>
7.1 Pourquoi utiliser la convolution ? . . . . .	26
7.2 Architecture d'un CNN . . . . .	27
7.3 Fonctionnement de la convolution . . . . .	27
7.4 Padding et stride . . . . .	28
7.5 Application aux images : exemple MNIST . . . . .	28
<b>8 Modèles CNN connus pour le computer vision et bonnes pratiques</b>	<b>29</b>
8.1 Architectures célèbres . . . . .	29
8.2 Bonnes pratiques pour les CNN . . . . .	30
<b>9 Une utilisation très riche des CNN pour l'embarqué : l'analyse de séquences temporelles</b>	<b>31</b>
<b>10 Keyword Spotting (KWS) et CNN</b>	<b>34</b>
10.1 Les MFCC : les premiers pas de la reconnaissance vocale . . . . .	34
10.2 L'Évolution des Réseaux de Neurones . . . . .	35
10.3 Les Spectrogrammes Mel et Techniques Hybrides . . . . .	36
10.4 Systèmes Embarqués et KWS : L'Art du Compromis . . . . .	37
10.5 Implémentation sur l'ESP32 . . . . .	37
<b>11 Quantification et pruning : comment rendre un modèle compatible avec du matériel léger</b>	<b>38</b>
11.1 Quantification . . . . .	38

11.2 Pruning . . . . .	38
11.3 Combinaison des deux approches . . . . .	38

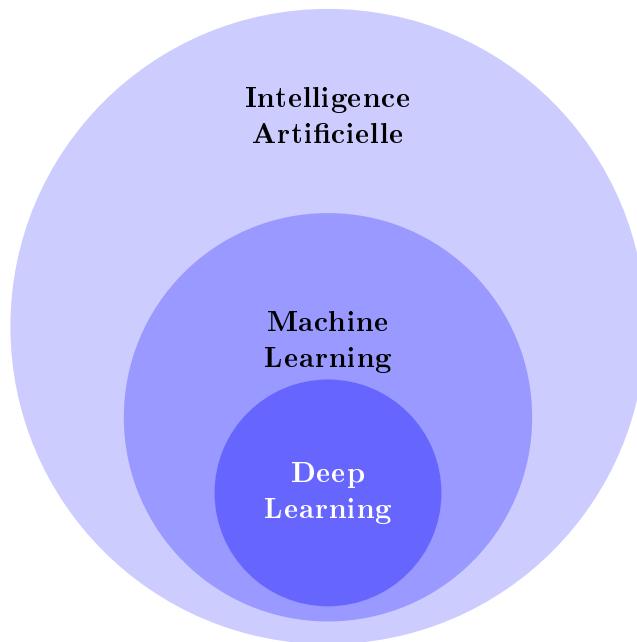
## 1 Introduction

Ce cours est consacré au *Deep Learning*, ou apprentissage profond, un des domaines les plus dynamiques de l'intelligence artificielle contemporaine. À travers ce cours, nous allons découvrir comment fonctionnent les réseaux de neurones artificiels, comment ils sont apparus dans l'histoire des sciences, et pourquoi ils sont aujourd'hui au cœur de nombreuses applications comme la reconnaissance d'images, la conduite autonome ou encore la traduction automatique.

Inspirés du fonctionnement des neurones biologiques, ces algorithmes sont capables d'apprendre à partir de données pour accomplir des tâches variées, parfois mieux que les humains. Le point de départ de ce voyage est une question simple mais fondamentale : comment une machine peut-elle apprendre ?

### 1.1 L'apprentissage automatique : intuition et motivation

Pour y répondre, nous commencerons par explorer les fondements du *Machine Learning*, une branche de l'intelligence artificielle qui consiste à créer des modèles capables d'apprendre à partir d'exemples. À partir de données, la machine ajuste des paramètres afin de prédire au mieux une sortie attendue. Cette logique nous amènera naturellement à parler des premiers modèles, comme la régression linéaire, avant de découvrir les modèles plus complexes qui ont marqué l'histoire du Deep Learning.



### 1.2 Les premiers neurones artificiels

Nous remonterons ainsi à 1943, année où deux scientifiques, Warren McCulloch et Walter Pitts, publient un article fondateur dans lequel ils modélisent le fonctionnement d'un neurone biologique par une fonction logique à seuil. Ce premier neurone artificiel, bien que rudimentaire, permet déjà de simuler des fonctions logiques comme AND ou OR. Mais ce modèle, purement symbolique, ne peut pas apprendre par lui-même.

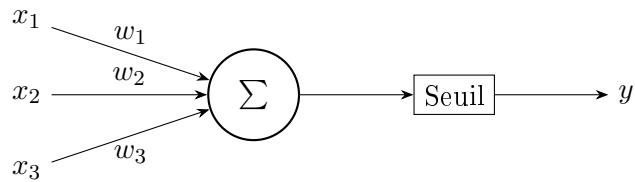


FIGURE 1 – Structure du neurone artificiel de McCulloch et Pitts

### 1.3 Le Perceptron et la naissance de l'apprentissage

Ce n'est qu'en 1957 qu'un progrès décisif est réalisé avec le perceptron de Frank Rosenblatt. Inspiré par les travaux du psychologue Donald Hebb, Rosenblatt introduit pour la première fois une règle d'apprentissage : les poids associés aux connexions du neurone sont ajustés automatiquement en fonction des erreurs commises. Le perceptron devient ainsi le premier neurone artificiel à apprendre par expérience.

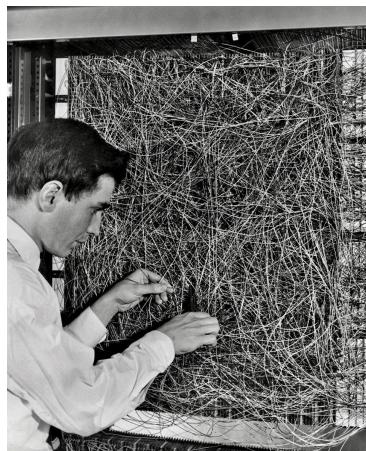


FIGURE 2 – Frank Rosenblatt devant le Mark I Perceptron en 1960.

Le Perceptron Mark I, développé au Cornell Aeronautical Laboratory, est considéré comme l'un des tout premiers ordinateurs dédiés à l'apprentissage automatique. La machine, conçue par Frank Rosenblatt, implémentait le modèle du perceptron pour la reconnaissance de formes visuelles à partir de photodiodes. Cette photographie emblématique illustre combien il devait galérer à l'époque et montre les premières réalisations matérielles en intelligence artificielle. Le Mark I Perceptron utilisait une architecture essentiellement analogique pour implémenter les connexions synaptiques, tandis que la prise de décision (sortie du neurone) était de type logique (numérique).

### 1.4 Limites du modèle linéaire et premier hiver de l'IA

Toutefois, le perceptron reste un modèle linéaire, incapable de résoudre certains problèmes simples comme le XOR. Ce constat conduit à un premier essoufflement de l'intelligence artificielle dans les années 1970, parfois appelé le premier "hiver de l'IA". Pendant cette période, les investissements et les espoirs dans le domaine diminuent fortement.

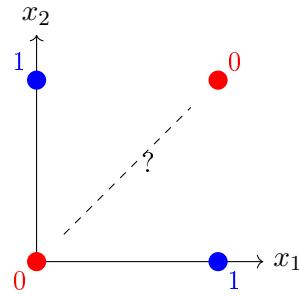


FIGURE 3 – Problème du XOR : aucune droite ne permet de séparer parfaitement les deux classes. Le perceptron, modèle linéaire, ne peut résoudre ce problème.

## 1.5 Le perceptron multicouches et la rétropropagation

Il faut attendre les années 1980 pour que Geoffrey Hinton et d'autres chercheurs développent le perceptron multicouches, capable d'exprimer des fonctions non linéaires complexes. Cette architecture, combinée à un nouvel algorithme appelé *rétropropagation du gradient*, ouvre la voie aux réseaux de neurones profonds. Le réseau devient un ensemble structuré de couches de neurones, chaque couche transformant progressivement l'information. L'apprentissage se fait par propagation de l'erreur depuis la sortie vers les couches internes, grâce à la descente de gradient.

Cette avancée marque le point de départ des réseaux de neurones profonds (pierre angulaire de toutes les solutions de Deep Learning). Le perceptron multicouches, entraîné par rétropropagation, constitue en réalité le socle conceptuel des architectures modernes dites « réseaux de neurones profonds entièrement connectés » (*fully connected deep neural networks* ou *deep feed-forward networks*). Ces réseaux demeurent aujourd'hui, comme vous le verrez dans les exemples illustrant ce cours, au cœur de nombreux systèmes d'apprentissage profond, même si d'autres structures spécialisées (convolutives, récurrentes, etc) ont vu le jour par la suite, notamment pour l'étude des séquences temporelles et des images.

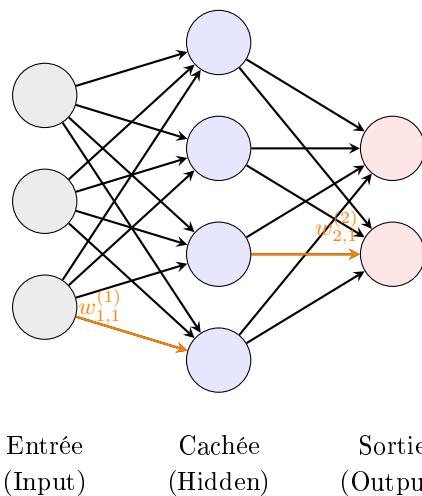


FIGURE 4 – Réseau de neurones multicouche entièrement connecté (MLP) : chaque neurone d'une couche est relié à tous les neurones de la couche suivante, avec des poids  $w_{i,j}^{(1)}$  et  $w_{k,l}^{(2)}$  indiqués ici sur deux connexions.

## 1.6 L'essor du Deep Learning moderne

Ce mécanisme, allié à l'essor de la puissance de calcul et à la disponibilité de grandes bases de données à partir des années 2010, permet un véritable bond en avant. Aujourd'hui, les réseaux de neurones sont capables de traiter des images, du texte, de la voix, et même de générer du contenu original. Le Deep Learning est devenu un outil incontournable dans de nombreux domaines scientifiques, industriels et artistiques.

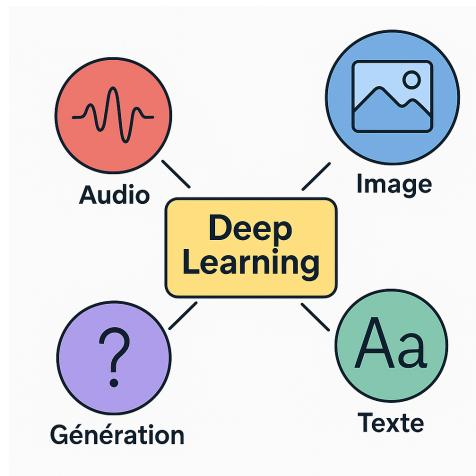


FIGURE 5 – Domaines d'application du Deep Learning : audio, image, texte, génération d'image, *Large Language Model*, ...

## 1.7 Approche pédagogique de cette formation

Dans cette formation, nous allons apprendre à construire de tels réseaux, d'abord à la main avec `Numpy`, pour bien comprendre les mécanismes internes, puis en utilisant les bibliothèques modernes comme `TensorFlow` et `Keras`. Chaque étape du cours correspondra à une notion-clé du Deep Learning, présentée de manière progressive, intuitive et rigoureuse, avec des exemples pratiques et des exercices d'application.

## 2 Le Perceptron et l'apprentissage supervisé

Le perceptron est le premier neurone artificiel doté d'une véritable capacité d'apprentissage. Proposé par Frank Rosenblatt en 1957, il reprend la structure du neurone formel de McCulloch et Pitts, mais y ajoute un mécanisme essentiel : une règle de mise à jour automatique des poids en fonction des erreurs d'apprentissage. Le perceptron devient ainsi le premier algorithme d'apprentissage supervisé.

Dans ce modèle, le neurone reçoit des entrées  $x_1, x_2, \dots, x_n$ , associées à des poids  $w_1, w_2, \dots, w_n$ . Il calcule une somme pondérée :

$$z = \sum_{i=1}^n w_i x_i + b$$

et applique une fonction d'activation. Dans sa forme historique, le perceptron utilisait une fonction seuil, mais cette dernière n'est pas différentiable. Pour le rendre compatible avec une mé-

thode d'optimisation efficace, on utilise désormais une fonction continue et dérivable : la fonction sigmoïde.

## 2.1 La fonction sigmoïde

La fonction sigmoïde transforme toute valeur réelle  $z$  en un nombre compris entre 0 et 1. Elle est définie par :

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

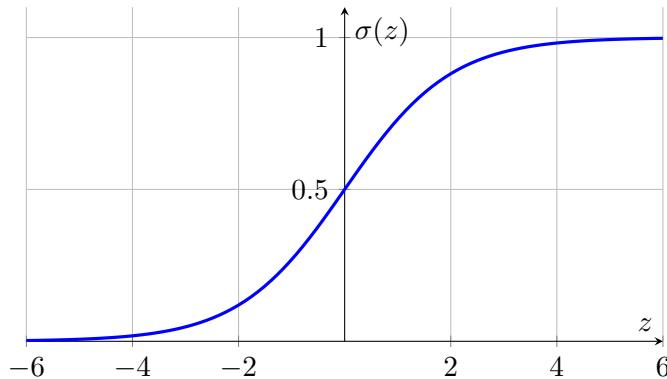


FIGURE 6 – Courbe de la fonction sigmoïde

Cette fonction est idéale pour des tâches de classification binaire, car sa sortie peut être interprétée comme une probabilité. Elle est continue, différentiable, et sa pente maximale se situe autour de  $z = 0$ .

## 2.2 La fonction coût log loss

L'apprentissage d'un réseau de neurones nécessite de mesurer l'erreur entre la sortie prédictive  $y_{\text{pred}}$  et la sortie attendue  $y_{\text{true}}$ . Pour cela, on utilise la fonction log loss :

$$\mathcal{L}(y_{\text{true}}, y_{\text{pred}}) = -(y_{\text{true}} \log y_{\text{pred}} + (1 - y_{\text{true}}) \log(1 - y_{\text{pred}})).$$

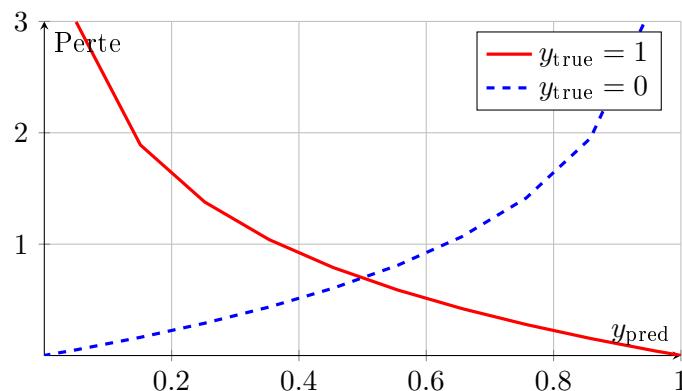


FIGURE 7 – Fonction log loss selon la valeur de vérité

Cette fonction est très pénalisante lorsque la prédiction est proche de 0 alors que la vérité est 1 (et inversement). Par exemple, si la vérité est  $y_{\text{true}} = 1$  et que le modèle prédit  $y_{\text{pred}} = 0.01$ , alors :

$$\mathcal{L}(1, 0.01) = -\log(0.01) \approx 4.60,$$

ce qui correspond à une pénalité sévère. À l'inverse, si  $y_{\text{pred}} = 0.99$ , alors :

$$\mathcal{L}(1, 0.99) = -\log(0.99) \approx 0.01,$$

ce qui indique une très bonne prédiction.

Cette fonction n'est pas choisie au hasard : elle est issue du cadre probabiliste. Lorsqu'on suppose que les données sont générées selon une loi de Bernoulli de paramètre  $y_{\text{pred}}$ , la \*\*log loss\*\* correspond à la \*\*négation de la log-vraisemblance\*\* :

$$\mathcal{L} = -\log P(y_{\text{true}} | y_{\text{pred}}),$$

autrement dit, on cherche à maximiser la probabilité d'observer les bonnes sorties. En minimisant la log loss, on maximise donc la vraisemblance du modèle.

## 2.3 Descente de gradient

Pour réduire l'erreur de prédiction, on ajuste les poids du modèle grâce à une méthode appelée *descente de gradient*. Celle-ci consiste à mettre à jour chaque paramètre dans la direction qui fait le plus diminuer la fonction coût :

$$w_i \leftarrow w_i - \alpha \frac{\partial \mathcal{L}}{\partial w_i}$$

où  $\alpha$  est un petit coefficient appelé taux d'apprentissage (*learning rate*).

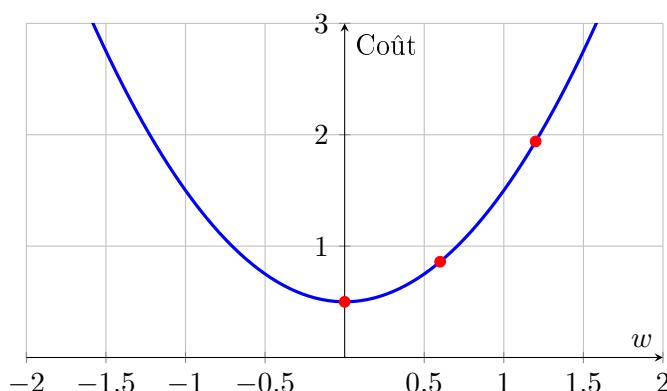


FIGURE 8 – Principe de la descente de gradient (exemple en 1D)

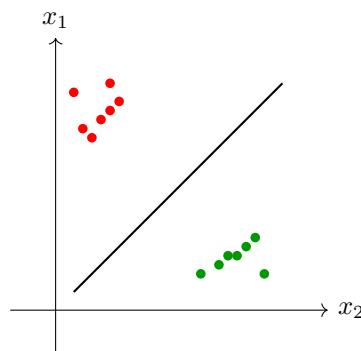
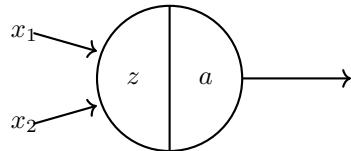
La descente de gradient est un processus itératif : à chaque étape, le modèle fait une prédiction, mesure l'erreur, calcule les dérivées partielles (les gradients), ajuste ses paramètres, et recommence.

## 2.4 Evaluation des gradients et mise à jour des poids neuronaux

Si vous êtes habitués aux méthodes numériques, la descente de gradient est tout à fait classique. Dans le contexte numérique qui nous préoccupe ici, il y a quand même une difficulté pour le calcul du gradient.

Dans cette section nous proposons le détail du calcul du gradient dans un exemple simple. Mais en fait la portée du calcul est généralisable à tous les calculs impliquant l'entraînement de réseaux neuronaux.

#### 2.4.1 Définitions



- fonction linéaire :  $z = w_1x_1 + w_2x_2 + b$
- fonction d'activation :  $a = \frac{1}{1+e^{-z}}$

#### 2.4.2 Dérivée partielle de la fonction de cout

La descente de gradient nécessite l'évaluation de la dérivée partielle de la fonction de coût.

Voici quelques éléments pour aider à réaliser le calcul :

Il faut décomposer la dérivée partielle de la manière suivante :

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial w_i}$$

Pour  $a$  il faut faire appel à la composition de fonction :

$$a = g \circ f = g(f(z))$$

$$g = 1/f$$

$$f = 1 + e^{-z}$$

La dérivée d'une composition de fonction s'écrit :

$$(g \circ f)' = g'(f(z)).f'(z)$$

En utilisant ces quelques éléments, il est facile de montrer que :

$$\boxed{\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_i} &= \frac{1}{m} \sum_j (a^{(j)} - y^{(j)}) x_i \\ \frac{\partial \mathcal{L}}{\partial b} &= \frac{1}{m} \sum_j (a^{(j)} - y^{(j)})\end{aligned}}$$

Ces résultats ont une portée générale : ils interviennent dans tous les entraînements.

## 2.5 Exercice : mise à jour du poids dans un neurone sigmoïde

On considère un neurone avec une seule entrée  $x$ , un poids  $w$ , et un biais  $b = 0$ . La fonction d'activation est la sigmoïde :

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \text{avec } z = w \cdot x.$$

On souhaite entraîner ce neurone à produire  $y_{\text{true}} = 1$  lorsque  $x = 2$ , en utilisant la fonction coût log loss :

$$\mathcal{L} = -(y_{\text{true}} \cdot \log(y_{\text{pred}}) + (1 - y_{\text{true}}) \cdot \log(1 - y_{\text{pred}})),$$

avec  $y_{\text{pred}} = \sigma(z)$ .

1. Calculez la prédiction  $y_{\text{pred}}$  pour  $w = 0.5$ .
2. Calculez la perte  $\mathcal{L}$ .
3. Calculez le gradient  $\frac{\partial \mathcal{L}}{\partial w}$ .
4. Mettez à jour le poids  $w$  avec un taux d'apprentissage  $\alpha = 0.1$ .

**Corrigé :**

1.  $z = 0.5 \cdot 2 = 1$ , donc  $y_{\text{pred}} = \sigma(1) = \frac{1}{1+e^{-1}} \approx 0.731$ .

2.  $\mathcal{L} = -\log(0.731) \approx 0.313$ .

3. Le gradient est :

$$\frac{\partial \mathcal{L}}{\partial w} = (y_{\text{pred}} - y_{\text{true}}) \cdot x = (0.731 - 1) \cdot 2 = -0.538.$$

4. Mise à jour :

$$w \leftarrow w - \alpha \cdot \frac{\partial \mathcal{L}}{\partial w} = 0.5 - 0.1 \cdot (-0.538) = 0.5 + 0.0538 \approx 0.554.$$

## 3 Vectorisation des équations

### 3.1 Pourquoi vectoriser ?

En deep learning, les modèles manipulent souvent un grand nombre d'exemples en parallèle. Il serait inefficace de traiter ces données une par une. La **vectorisation** consiste à reformuler les équations pour pouvoir effectuer les calculs de manière simultanée sur tous les exemples, en utilisant des opérations sur des matrices ou des vecteurs. Cette approche permet :

- d'accélérer considérablement les calculs (notamment grâce aux GPU) ;
- de rendre le code plus concis et plus lisible ;
- de généraliser les formules à un nombre quelconque de données ou de variables.

## 3.2 Vectorisation du modèle

On considère un ensemble de  $m$  exemples, chacun ayant  $n$  variables d'entrée. Ces entrées sont regroupées dans une matrice  $X$  de dimension  $m \times n$  :

$$X = \begin{pmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(m)} \end{pmatrix} \quad \text{où chaque } x^{(i)} \in \mathbb{R}^n$$

Les poids sont regroupés dans un vecteur colonne  $W \in \mathbb{R}^n$ , et le biais est noté  $b \in \mathbb{R}$ .

### 3.2.1 Calcul du vecteur $Z$ (sortie linéaire)

Au lieu de calculer chaque  $z^{(i)}$  séparément, on écrit :

$$Z = XW + b$$

avec :

- $X \in \mathbb{R}^{m \times n}$ ,
- $W \in \mathbb{R}^{n \times 1}$ ,
- $Z \in \mathbb{R}^{m \times 1}$ ,
- $b$  est broadcasté à l'ensemble des lignes.

### 3.2.2 Activation sigmoïde vectorisée

En appliquant la fonction sigmoïde à chaque composante de  $Z$ , on obtient :

$$A = \sigma(Z)$$

où  $A \in \mathbb{R}^{m \times 1}$  contient les prédictions du modèle pour les  $m$  exemples.

### 3.2.3 Fonction de coût vectorisée

La fonction log loss, définie précédemment pour un seul exemple, devient :

$$\mathcal{L} = -\frac{1}{m} [Y^T \log A + (1 - Y)^T \log(1 - A)]$$

avec  $Y \in \mathbb{R}^{m \times 1}$  le vecteur des sorties attendues.

### 3.2.4 Gradient vectorisé

Les gradients s'écrivent :

$$\boxed{\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{m} X^T (A - Y)} \quad \text{et} \quad \boxed{\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^m (a^{(i)} - y^{(i)})}$$

### 3.2.5 Mise à jour des paramètres

On utilise la descente de gradient vectorisée :

$$W \leftarrow W - \alpha \frac{1}{m} X^T (A - Y) \quad \text{et} \quad b \leftarrow b - \alpha \frac{1}{m} \sum (A - Y)$$

## 3.3 Extension à $n$ variables

Toutes les équations ci-dessus sont valables pour n'importe quelle valeur de  $n$ , le nombre de variables d'entrée. Il suffit que :

- $X$  soit de dimension  $m \times n$ ,
- $W$  de dimension  $n \times 1$ ,
- et  $Z, A, Y$  de dimension  $m \times 1$ .

## 3.4 Intérêt de la vectorisation

La vectorisation est indispensable pour exploiter la puissance du calcul matriciel et des bibliothèques comme NumPy, TensorFlow ou PyTorch. Elle permet d'entraîner efficacement des réseaux de neurones sur de grandes quantités de données, tout en conservant des formules compactes et lisibles.

**Point technique :** la vectorisation n'est pas seulement un amélioration de lisibilité, c'est une condition nécessaire pour tirer parti des architectures modernes :

- Les processeurs CPU modernes intègrent des instructions SIMD (AVX, SSE, NEON) permettant de traiter plusieurs données en une instruction : par exemple, AVX2 peut multiplier huit valeurs 32 bits simultanément, offrant un gain de performance jusqu'à 8.
- Les GPU utilisent un modèle SIMT (Single Instruction, Multiple Thread), exécutant en parallèle des milliers de threads sur des blocs d'instructions identiques.

Par exemple, un calcul de produit scalaire non vectorisé prendrait une boucle élément par élément, alors que la version vectorisée (e.g. AVX-256) traite 8 flottants en une instruction : sans vectorisation, on sous-exploite de 8 à 512 fois la capacité de calcul du matériel (GPU ou CPU).

Ainsi, la vectorisation permet non seulement un code plus clair, mais est essentielle pour bénéficier pleinement de la puissance SIMD/SIMT des CPU/GPU, et atteindre des accélérations de plusieurs ordres de grandeur.

## 4 Exercices guidés

### 4.1 Implémenter un neurone artificiel en Python (vectorisé)

Pour consolider la compréhension, nous proposons une série d'exercices progressifs pour implémenter à la main un neurone logistique vectorisé en Python, à l'aide de NumPy.

**Pré-requis :** `import numpy as np` en début de script. On suppose que  $X$  est une matrice de forme  $(m, n)$  ( $m$  exemples,  $n$  variables d'entrée).

#### Exercice 1 : Initialisation des paramètres

Écrire une fonction `initialize_parameters(X)` qui retourne un vecteur  $W$  (dimension  $(n, 1)$ ) initialisé à zéro, et un biais  $b = 0$ .

```
def initialize_parameters(X):
    n = X.shape[1]
    W = np.zeros((n, 1))
    b = 0.0
    return W, b
```

Tester avec :

```
W, b = initialize_parameters(X)
print(W.shape, type(b))
```

#### Exercice 2 : Propagation avant (forward)

Écrire la fonction `model_forward(X, W, b)` qui calcule  $Z = XW + b$ , puis applique la sigmoïde  $A = \sigma(Z)$ , avec :

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def model_forward(X, W, b):
    Z = X @ W + b
    A = sigmoid(Z)
    return A
```

Tester avec de petites valeurs, vérifier la forme et que  $0 \leq A \leq 1$ .

#### Exercice 3 : Fonction coût (log-loss)

Écrire `compute_cost(A, Y)` qui calcule :

$$\mathcal{L} = -\frac{1}{m} [Y^T \log(A) + (1 - Y)^T \log(1 - A)]$$

```
def compute_cost(A, Y):
    m = Y.shape[0]
    cost = - (1/m) * (Y.T @ np.log(A) + (1 - Y).T @ np.log(1 - A))
    return np.squeeze(cost)
```

#### Exercice 4 : Calcul des gradients

Écrire `compute_gradients(X, A, Y)` qui retourne :

$$dW = \frac{1}{m} X^T (A - Y), \quad db = \frac{1}{m} \sum (A - Y)$$

```
def compute_gradients(X, A, Y):
    m = X.shape[0]
    dW = (1/m) * (X.T @ (A - Y))
    db = (1/m) * np.sum(A - Y)
    return dW, db
```

#### Exercice 5 : Mise à jour des paramètres

Écrire `update_parameters(W, b, dW, db, alpha)` qui retourne :

```
def update_parameters(W, b, dW, db, alpha):
    W_new = W - alpha * dW
    b_new = b - alpha * db
    return W_new, b_new
```

#### Exercice 6 : Boucle d'entraînement

Assembler ces fonctions dans `train_model(X, Y, num_iterations, alpha)` :

- Initialiser  $W, b$
- Pour chaque itération :
  - Forward :  $A = \text{model\_forward}(X, W, b)$
  - Calcul du coût
  - Backward :  $dW, db$
  - Mise à jour :  $W, b$
  - (optionnel) Stocker le coût tous les 10 itérations
- Retourner les paramètres et la liste des coûts

### Exercice 7 : Prédiction et évaluation

Écrire `predict(X, W, b, threshold=0.5)` qui applique le modèle, compare à 0.5, et retourne un vecteur de classes (0 ou 1). Calculer le taux de bonnes réponses (accuracy).

```
def predict(X, W, b, threshold=0.5):
    A = model_forward(X, W, b)
    return (A > threshold).astype(int)
```

### Exercice 8 : Visualisation de la frontière de décision

Pour  $n = 2$ , extraire  $w_1, w_2, b$ , générer une grille  $(x_1, x_2)$ , calculer  $x_2 = -(w_1x_1 + b)/w_2$ , et tracer la droite sur le nuage de points. Utiliser matplotlib.

### Exercice 9 (optionnel) : Généralisation à $n$ variables

Modifier  $X$  pour avoir plus de colonnes, générer de nouveaux  $Y$ , relancer `train_model`, vérifier que tout fonctionne.

### Exercice bonus : Passage en style orienté objet (OOP)

Regrouper les fonctions dans une classe `LogisticNeuron` avec méthodes pour chaque étape (initialize, forward, cost, backward, update, predict).

**Conseils :**

- Utiliser la vectorisation partout, pas de boucles sur les exemples.
- Vérifier systématiquement les `shape` des matrices/vecteurs.
- Tracer régulièrement la courbe du coût pour diagnostiquer la convergence.

## 4.2 Les chats et les chiens !

### 4.2.1 Diagnostiquer et stabiliser l'apprentissage

Après avoir implémenté notre premier neurone logistique et vérifié son fonctionnement sur de petits jeux de données artificiels, nous allons à présent l'entraîner sur un *véritable* problème de vision : distinguer des photos de chats et de chiens<sup>1</sup>. Cette nouvelle étape nous confronte à trois difficultés classiques :

- 1) la gestion des **débordements numériques** (overflow / underflow) ;
- 2) la **normalisation** indispensable des données d'entrée ;
- 3) le **réglage des hyper-paramètres** et la détection du sur-apprentissage.

---

1. Le jeu de données utilisé est un sous-ensemble 64 pixel  $\times$ 64 pixel du "Cats vs Dogs" de Kaggle, converti au format `HDF5`.

Le présent chapitre détaille chacune de ces difficultés, propose des expériences reproductibles en NumPy et fournit les bonnes pratiques pour les contourner.

#### 4.2.2 Remise en contexte : préparer le *dataset*

Le fichier `cats_vs_dogs.h5` contient deux ensembles :

- `X_train` : 1000 images  $\in \mathbb{R}^{1000 \times 64 \times 64}$  ;
- `X_test` : 200 images  $\in \mathbb{R}^{200 \times 64 \times 64}$ .

Pour pouvoir utiliser notre neurone, chaque image doit être *aplatie* en un vecteur de 4096 pixels :

```
X_train_flat = X_train.reshape(X_train.shape[0], -1)
X_test_flat  = X_test.reshape(X_test.shape[0], -1)
print(X_train_flat.shape) # (1000, 4096)
```

**Question** : pourquoi ne pas avoir laissé la dimension  $64 \times 64$ ? *Réponse* : notre neurone est *linéaire*. Il attend un seul produit scalaire  $z = W^\top x + b$  et ne dispose d'aucune structure 2D (contrairement, par exemple, aux réseaux convolutifs, que nous utiliseront largement par la suite).

#### 4.2.3 Débordements exponentiels et logarithmiques

Lors du premier entraînement, de nombreux participants ont rencontré deux messages d'erreur récurrents :

- `RuntimeWarning: overflow encountered in exp`
- `RuntimeWarning: divide by zero encountered in log`

Ces problèmes proviennent directement des bornes de nos pixels (0 à 255). Avec 4096 variables, le terme linéaire  $z = W^\top x + b$  peut atteindre des valeurs absolues supérieures à  $10^4$ , ce qui entraîne :

$$e^{-|z|} \longrightarrow \begin{cases} 0, & z \ll 0, \\ +\infty, & z \gg 0. \end{cases}$$

En cascade, la *log-loss*  $-(y \log a + (1-y) \log(1-a))$  échoue car  $\log 0$  n'est pas défini.

**Antidote : le *clipping*.** On insère un *epsilon* minuscule dans les logarithmes :

```
eps = 1e-15
loss = - np.mean( y*np.log(a+eps) + (1-y)*np.log(1-a+eps) )
```

Cette simple astuce élimine toutes les divisions par zéro.

#### 4.2.4 Normaliser les données

Même après *clipping*, les débordements de `exp` persistent. La véritable solution est de mettre chaque pixel sur une même échelle, en remplaçant  $\{0, \dots, 255\}$  par  $\{0, 1\}$  :

$$x_{ij}^{\text{norm}} = \frac{x_{ij}}{255}.$$

```
X_train_flat = X_train_flat / 255.
X_test_flat  = X_test_flat  / 255.
```

**Pourquoi cela fonctionne-t-il ?** La borne supérieure de  $z$  devient  $\|W\|_1 \leq 4096$ , au lieu de  $4096 \times 255$ . Les sorties de la sigmoïde  $\sigma(z) = 1/(1 + e^{-z})$  restent donc strictement comprises entre  $10^{-12}$  et  $1 - 10^{-12}$  : plus aucun overflow.

#### 4.2.5 Impact sur la descente de gradient

Une normalisation correcte améliore la géométrie de la fonction coût. La figure ?? compare l'allure de  $\mathcal{L}(w_1, w_2)$  avant et après normalisation (variables  $w_1, w_2$  illustratives). Sans normalisation, le paysage est une falaise quasi verticale ; la descente de gradient « rebondit » sans jamais converger.

#### 4.2.6 Réglage du taux d'apprentissage

Une fois le *dataset* stabilisé, le principal hyper-paramètre reste le **learning rate  $\alpha$** .

$$w \leftarrow w - \alpha \frac{\partial \mathcal{L}}{\partial w}.$$

- $\alpha = 10^{-1}$  : pas trop grands  $\rightarrow$  oscillations.
- $\alpha = 10^{-2}$  : compromis sûr pour la plupart des jeux de données tabulaires.
- $\alpha < 10^{-3}$  : risque de stagnation (apprentissage interminable).

Le diagnostic le plus simple consiste à *visualiser* simultanément la *loss* et l'*accuracy*. Le gabarit suivant affiche deux courbes, enregistrées toutes les `display_step = 10` itérations :

```
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(train_loss, label='train')
plt.plot(test_loss, label='test' )
plt.legend(); plt.title('Log-loss')

plt.subplot(1,2,2)
plt.plot(train_acc , label='train')
plt.plot(test_acc , label='test' )
plt.legend(); plt.title('Accuracy')
```

#### 4.2.7 Déetecter le sur-apprentissage

Lorsque la courbe TRAIN / LOSS continue de baisser tandis que la courbe TEST / LOSS stagne (ou remonte), le modèle souffre d'*over-fitting*. Sur notre exemple, le phénomène apparaît après  $\approx 2000$  itérations, malgré la normalisation (figure ??).

#### Contremesures principales

- a) Augmenter la taille du *dataset* (collecte ou *data-augmentation*).
- b) Ajouter une **régularisation**  $L_2$  :  $\mathcal{L}_\lambda = \mathcal{L} + \frac{\lambda}{2m} \|W\|_2^2$ .
- c) Réduire la complexité du modèle<sup>2</sup>.

#### 4.2.8 Progress bar et bonnes pratiques de code

Un entraînement long sans retour visuel est frustrant. La bibliothèque `tqdm` fournit une barre de progression en une seule ligne :

```
from tqdm import tqdm

for i in tqdm(range(num_iterations)):  
    ...
```

Pensez également à :

- **clipper** les activations : `np.clip(a, eps, 1-eps)`.
- N'enregistrer les métriques (`loss, accuracy`) que toutes les  $k$  itérations (`display_step`) pour réduire la charge mémoire.

#### 4.2.9 Vers un réseau multicouche

La morale de cette séance est double :

1. Avant d'accuser votre algorithme, vérifiez que les données sont *propres* (shape, normalisation, epsilons).
2. Un neurone isolé est un *classifieur linéaire*. Pour traiter des images réalistes, il faut lui adjoindre des couches cachées : perceptron multicouche (MLP), puis réseau convolutif (CNN).

Le prochain chapitre présentera la construction pas-à-pas d'un premier MLP entièrement vecteurisé, l'algorithme de *rétro-propagation* complet, puis sa mise en œuvre sur notre jeu de photos « Chats vs Chiens ».

---

2. Ici, diminuer le nombre de poids n'a guère de sens : un seul neurone est déjà trop *simple*. La vraie solution est de passer à un réseau multicouche, objet du chapitre suivant.

## 5 Les réseaux de neurones multicouches : puissance et rigueur du calcul matriciel

### 5.1 Pourquoi les réseaux multicouches ? Limites de la linéarité

Un neurone isolé (ou perceptron) est un classifieur linéaire. Or, la majorité des problèmes réels présentent des structures bien plus complexes : imaginer séparer des spirales, reconnaître des chiffres manuscrits, analyser une image couleur... On pourrait tenter de « bricoler » des combinaisons non linéaires des variables (*feature engineering*), mais cela demande une expertise humaine considérable, et reste souvent insuffisant.

**L'idée centrale du deep learning** : empiler plusieurs couches de neurones, chacune transformant les représentations de la précédente, pour aboutir à une modélisation hiérarchique, automatique et puissante de la complexité du réel. C'est ce qu'on appelle un *réseau de neurones multicouches* (MLP, *multi-layer perceptron*).

### 5.2 De l'individu à la foule : la logique matricielle

Dès que l'on souhaite traiter un nombre non trivial d'entrées, ou d'exemples, il devient essentiel d'adopter une représentation **vectorielle puis matricielle**.

**Notations :**

- $m$  : nombre d'exemples du batch (parfois  $m = 1000$  ou  $32\,768$  !)
- $n^{[l]}$  : nombre de neurones dans la couche  $l$
- $L$  : nombre total de couches (hors entrée)

À chaque couche  $l$  :

Entrée	:	$A^{[l-1]} \in \mathbb{R}^{n^{[l-1]} \times m}$
Poids	:	$W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$
Biais	:	$b^{[l]} \in \mathbb{R}^{n^{[l]} \times 1}$
Sortie linéaire	:	$Z^{[l]} \in \mathbb{R}^{n^{[l]} \times m}$
Activation	:	$A^{[l]} \in \mathbb{R}^{n^{[l]} \times m}$

#### Exemple de dimensions

Imaginons un réseau à 3 couches :

- **Entrée** :  $X \in \mathbb{R}^{784 \times 100}$  (100 images de  $28 \times 28$  pixels)
- **Cachée 1** :  $n^{[1]} = 64$  neurones  
 $W^{[1]} \in \mathbb{R}^{64 \times 784}, \quad b^{[1]} \in \mathbb{R}^{64 \times 1}, \quad A^{[1]} \in \mathbb{R}^{64 \times 100}$
- **Cachée 2** :  $n^{[2]} = 32$  neurones  
 $W^{[2]} \in \mathbb{R}^{32 \times 64}, \quad b^{[2]} \in \mathbb{R}^{32 \times 1}, \quad A^{[2]} \in \mathbb{R}^{32 \times 100}$
- **Sortie** :  $n^{[3]} = 10$  neurones (classification à 10 classes)  
 $W^{[3]} \in \mathbb{R}^{10 \times 32}, \quad b^{[3]} \in \mathbb{R}^{10 \times 1}, \quad A^{[3]} \in \mathbb{R}^{10 \times 100}$

À chaque étape, il est indispensable de vérifier la cohérence des dimensions, sinon toute l'architecture s'effondre !

### 5.3 Propagation avant : l'art du calcul matriciel

Le passage d'une couche à la suivante se décompose en deux étapes :

$$\begin{aligned} Z^{[l]} &= W^{[l]} A^{[l-1]} + b^{[l]} \\ A^{[l]} &= g^{[l]}(Z^{[l]}) \end{aligned}$$

Où  $g^{[l]}$  est la fonction d'activation (souvent ReLU ou sigmoïde).

- $W^{[l]} A^{[l-1]}$  est un produit matriciel : chaque neurone « mélange » toutes les activations de la couche précédente.
- $b^{[l]}$  est ajouté à chaque colonne (*broadcasting*).
- $A^{[l]}$  contient la sortie activée pour chaque neurone, sur chaque exemple du batch.

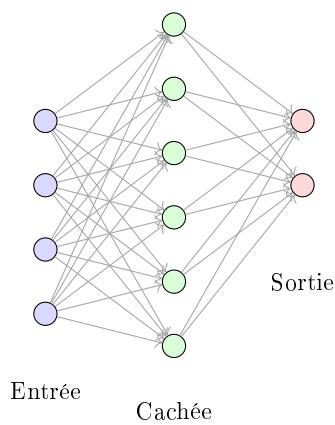
**Exemple :** Pour la première couche cachée d'un réseau images MNIST, sur un batch de 100 images :

- $W^{[1]} \in \mathbb{R}^{64 \times 784}$
- $A^{[0]} = X \in \mathbb{R}^{784 \times 100}$
- $Z^{[1]} = W^{[1]} X + b^{[1]} \in \mathbb{R}^{64 \times 100}$

Chaque colonne de  $Z^{[1]}$  correspond à une image du batch, chaque ligne à un neurone de la couche cachée.

### 5.4 Superposition, abstraction, profondeur

L'empilement de couches permet au réseau d'apprendre des représentations de plus en plus abstraites : *une première couche détecte des motifs simples, la suivante compose ces motifs pour former des formes, la suivante encore peut détecter des objets entiers*. Chaque couche est donc un espace intermédiaire, une transformation géométrique de l'espace des données.



### 5.5 L'entraînement : rétropropagation matricielle

Une fois la propagation avant réalisée, il s'agit d'**ajuster tous les poids et biais du réseau** pour réduire l'erreur entre la sortie produite et la vérité (loss). On utilise la rétropropagation (*backpropagation*), qui repose **elle aussi sur la rigueur du calcul matriciel**.

- On part du gradient de la perte par rapport à la sortie.
- On rétro-propage ce gradient couche par couche, en utilisant la dérivée de la fonction d'activation et la transposée de la matrice de poids.
- À chaque étape, on met à jour  $W^{[l]}$  et  $b^{[l]}$  avec leurs gradients, qui ont exactement les mêmes dimensions.

**Formule :**

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} \delta^{[l]} (A^{[l-1]})^T$$

où  $\delta^{[l]}$  est l'erreur « locale » (dimension  $n^{[l]} \times m$ ).

**Vigilance : à chaque instant, vérifier les tailles !**

- $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$
- $A^{[l-1]} \in \mathbb{R}^{n^{[l-1]} \times m}$
- $(A^{[l-1]})^T \in \mathbb{R}^{m \times n^{[l-1]}}$
- Donc  $\delta^{[l]}(A^{[l-1]})^T$  produit bien une matrice de taille  $n^{[l]} \times n^{[l-1]}$

## 5.6 Synthèse : penser en dimensions, visualiser en matrices

- Chaque couche est une transformation matricielle de l'espace.
- La rigueur sur les tailles évite 90% des bugs.
- Les matrices intermédiaires ( $Z$ ,  $A$ ,  $\delta$ , gradients) sont la clé pour comprendre ce qui se passe « sous le capot ».
- Plus le réseau est profond et large, plus il est capable de modéliser des phénomènes complexes, mais plus l'apprentissage devient exigeant.

**Exercice d'entraînement :** Compléter le tableau des tailles pour un réseau avec :

- $n^{[0]} = 3$  (entrée),  $n^{[1]} = 4$ ,  $n^{[2]} = 2$  (sortie),  $m = 5$  exemples.

Indiquez la taille de  $X$ ,  $W^{[1]}$ ,  $b^{[1]}$ ,  $A^{[1]}$ ,  $W^{[2]}$ ,  $b^{[2]}$ ,  $A^{[2]}$ .

Objet	Signification	Taille
$X$	Données d'entrée	$3 \times 5$
$W^{[1]}$	Poids, couche 1	$4 \times 3$
$b^{[1]}$	Biais, couche 1	$4 \times 1$
$A^{[1]}$	Activations cachées	$4 \times 5$
$W^{[2]}$	Poids, couche 2	$2 \times 4$
$b^{[2]}$	Biais, couche 2	$2 \times 1$
$A^{[2]}$	Sorties finales	$2 \times 5$

**Conclusion :** Maîtriser la structure et la dimension des matrices n'est pas un détail de code, c'est le cœur même du deep learning moderne. Toute implémentation, toute compréhension des phénomènes d'apprentissage, toute visualisation ou debug en dépend : pensez « opérations sur les matrices », visualisez vos shapes, et le deep learning deviendra un espace de jeu structuré et puissant.

## 6 Forward Propagation dans un réseau à deux couches

Avant d'aborder la rétropropagation, il est crucial de bien comprendre le fonctionnement de la **forward propagation**, ou propagation avant. C'est le processus par lequel les données, initialement brutes, traversent les couches successives d'un réseau de neurones jusqu'à produire une prédiction en sortie. Cette étape est fondamentale, car c'est sur sa structure que s'appuie ensuite la rétropropagation pour calculer les gradients et ajuster les paramètres du modèle.

Comprendre la forward propagation ne signifie pas seulement savoir écrire les équations : cela implique de saisir la logique sous-jacente, la nature des transformations effectuées couche par couche, et surtout de bien maîtriser les dimensions des matrices manipulées, car une grande partie des erreurs en deep learning provient de décalages de dimensions ou d'une mauvaise compréhension de la structure des tenseurs.

### 6.1 Schéma général du réseau

On considère un réseau de neurones à deux couches, comprenant :

- Une couche d'entrée, qui reçoit une matrice  $X$  de taille  $n_0 \times m$ ,
- Une première couche cachée avec  $n_1$  neurones,
- Une couche de sortie avec  $n_2$  neurones (souvent  $n_2 = 1$  pour la classification binaire).

Chaque colonne de  $X$  correspond à un exemple dans le batch. Par exemple, si on a  $m = 100$ , on traite simultanément 100 exemples. Chaque exemple est un vecteur de  $n_0$  features.

**Récapitulatif des dimensions :**

- $X \in \mathbb{R}^{n_0 \times m}$
- $W^{[1]} \in \mathbb{R}^{n_1 \times n_0}$ ,  $b^{[1]} \in \mathbb{R}^{n_1 \times 1}$
- $W^{[2]} \in \mathbb{R}^{n_2 \times n_1}$ ,  $b^{[2]} \in \mathbb{R}^{n_2 \times 1}$

### 6.2 Étape 1 : calcul des activations de la première couche

Le calcul se fait en deux temps : une transformation linéaire, suivie d'une activation non-linéaire :

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} \\ A^{[1]} &= g_1(Z^{[1]}) \end{aligned}$$

Le biais  $b^{[1]}$ , de taille  $n_1 \times 1$ , est ajouté par broadcasting à chaque colonne de  $W^{[1]}X$ .

Point technique : Broadcasting

Le **broadcasting** est une opération automatique dans les frameworks comme NumPy, TensorFlow ou PyTorch, qui permet d'ajouter une matrice de dimension  $(n \times 1)$  à une matrice de dimension  $(n \times m)$  en dupliquant implicitement la première sur les  $m$  colonnes.

Cette étape donne une matrice  $A^{[1]} \in \mathbb{R}^{n_1 \times m}$ , où chaque colonne représente les activations de la couche cachée pour un exemple donné.

### 6.3 Étape 2 : calcul des activations de la couche de sortie

La logique est exactement la même :

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

$$A^{[2]} = g_2(Z^{[2]})$$

On applique une transformation affine, puis une fonction d'activation (sigmoïde pour la classification binaire, softmax pour le multi-classes, etc.).

### 6.4 Étape 3 : calcul de la fonction coût

La sortie  $A^{[2]}$  est comparée à la vérité terrain  $Y$ . Pour la classification binaire, la fonction coût la plus utilisée est la log-loss (ou binary cross-entropy) :

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log a^{[2](i)} + (1 - y^{(i)}) \log(1 - a^{[2](i)}) \right]$$

En notation matricielle :

$$\mathcal{L} = -\frac{1}{m} \left[ Y \log A^{[2]} + (1 - Y) \log(1 - A^{[2]}) \right]$$

Attention aux dimensions de  $Y$

La matrice  $Y$  doit avoir exactement les mêmes dimensions que  $A^{[2]}$ , soit  $n_2 \times m$ . C'est fondamental pour que les opérations élément par élément soient bien définies.

### 6.5 Synthèse des dimensions

Symbol	Signification	Dimension
$X$	Données d'entrée	$n_0 \times m$
$W^{[1]}$	Poids couche 1	$n_1 \times n_0$
$b^{[1]}$	Biais couche 1	$n_1 \times 1$
$Z^{[1]}$	Sortie linéaire 1	$n_1 \times m$
$A^{[1]}$	Activation 1	$n_1 \times m$
$W^{[2]}$	Poids couche 2	$n_2 \times n_1$
$b^{[2]}$	Biais couche 2	$n_2 \times 1$
$Z^{[2]}$	Sortie linéaire 2	$n_2 \times m$
$A^{[2]}$	Prédiction finale	$n_2 \times m$
$Y$	Vérité terrain	$n_2 \times m$

## 6.6 Interprétation intuitive

- Chaque couche transforme l'espace des données en lui appliquant une géométrie nouvelle : les poids définissent des directions, les biais des translations.
- Chaque colonne d'une matrice d'activations correspond à un exemple du batch, et chaque ligne à la sortie d'un neurone donné sur tous les exemples.
- Le calcul matriciel rend possible un traitement parallèle efficace des données, indispensable à l'entraînement moderne des réseaux.

### En résumé

La forward propagation repose sur deux piliers : l'algèbre linéaire (produits matriciels, biais, broadcasting) et les fonctions non-linéaires. Maîtriser les dimensions à chaque étape est essentiel pour construire, comprendre et débuguer un réseau de neurones.

## 7 Les réseaux de neurones convolutif

Les réseaux neuronaux convolutionnels (CNN) sont une classe spécialisée de réseaux de neurones profonds, principalement conçus pour le traitement d'images et la reconnaissance de formes. Contrairement aux modèles classiques (SVM, réseaux multi-couches) qui n'exploitent pas la structure spatiale des images, les CNN apprennent à extraire automatiquement des caractéristiques pertinentes à grande échelle, à partir des images brutes. Cette capacité d'extraction automatique des représentations visuelles (sans avoir à définir manuellement les descripteurs) est un avantage clé. De plus, les couches convolutionnelles offrent une invariance aux translations : elles peuvent détecter un même motif (bord, texture, coin, etc.) indépendamment de sa position dans l'image. Ce comportement rappelle l'organisation hiérarchique du cortex visuel : chaque neurone détecte des caractéristiques sur une petite région locale (champ réceptif) et partage ses poids avec ses voisins. En pratique, ces propriétés rendent les CNN très efficaces pour les tâches de classification d'images, détection d'objets ou de segmentation.

### 7.1 Pourquoi utiliser la convolution ?

Dans un réseau de type perceptron multi-couches classique, chaque pixel d'entrée est connecté à chaque neurone de la couche suivante. Pour des images de grande taille, cela entraîne un nombre de paramètres très important (proportionnel à  $wh \times n$  pour une image de  $w \times h$  et  $n$  neurones cachés. Par exemple, pour des images  $1000 \times 1000$  et une couche cachée de seulement 1000 neurones, il faudrait plus d'un milliard de poids. Cette explosion du nombre de poids accroît le coût de calcul et aggrave le sur-apprentissage. De plus, un perceptron classique ne tient pas compte de la structure spatiale locale : il traite l'image comme un simple vecteur de pixels non ordonnés. Or, dans une image, les informations pertinentes sont souvent locales (bords, zones de texture, coins, etc.) et la catégorie visuelle d'un objet reste invariante à certaines transformations comme la translation (un chat reste un chat, quel que soit son placement dans l'image).

Le CNN introduit deux idées fondamentales pour surmonter ces limites : la connectivité locale et le partage de poids. Plutôt que de connecter chaque neurone à tous les pixels, chaque neurone convolutif ne regarde qu'une petite région locale de l'image (son champ réceptif). De plus, les poids du filtre sont partagés entre toutes les régions : un même petit filtre (noyau de convolution) balaye toute l'image. Cette notion de partage de poids (issue du Neocognitron de Fukushima)

permet de détecter le même motif en tout point de l'image en utilisant très peu de paramètres. La figure suivante illustre cette différence de connectivité : un réseau entièrement connecté (gauche) relie chaque pixel à chaque neurone de sortie, alors qu'en convolution (droite) chaque neurone n'englobe qu'une petite zone locale.

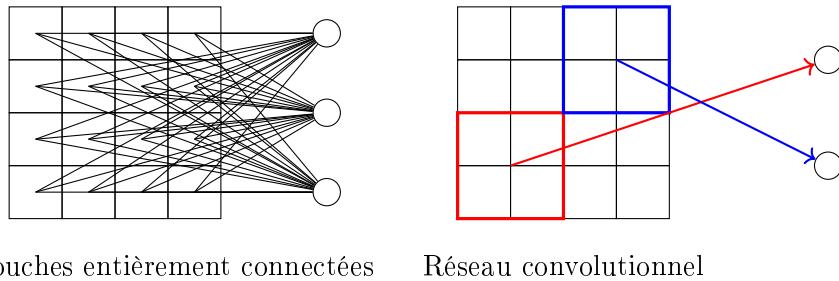


FIGURE 9 – Comparaison des schémas de connexions dans un réseau entièrement connecté (gauche) et dans un réseau convolutionnel (droite). En convolution, chaque neurone ne prend en compte qu'une région locale de l'image (deux régions colorées) et tous les filtres partagent les mêmes poids.

## 7.2 Architecture d'un CNN

Un CNN se compose typiquement de plusieurs blocs successifs alternant des couches de convolution et de pooling, suivis d'une ou plusieurs couches entièrement connectées en fin de réseau. Chaque couche convolutionnelle (**Conv**) applique plusieurs filtres (noyaux) sur l'image d'entrée pour extraire autant de cartes d'activations (ou « feature maps »). Par exemple, un filtre de taille  $F \times F$  appliqué à une image d'entrée  $I \times I$  produit une carte de sortie de taille approximativement  $O \times O$ , où  $O = (I - F + 2P)/S + 1$  selon le padding  $P$  et le pas (*stride*)  $S$ . Plusieurs filtres ( $K$  filtres) génèrent un volume de sortie de taille  $O \times O \times K$ . Ensuite, une couche de pooling (typiquement max pooling ou moyennage) réduit localement la dimension spatiale (par exemple en prenant les valeurs maximales sur des blocs) pour obtenir un « sous-échantillonage » et renforcer l'invariance locale.

Après plusieurs de ces blocs (« conv + pooling »), on a condensé l'information visuelle en un petit volume de caractéristiques. On aplatis alors ce volume 2D/3D pour le transformer en vecteur, puis on le branche sur un perceptron multi-couche final (**Fully Connected**) qui effectue la classification. Ce dernier objectif (par ex. prédition de la classe de l'image) utilise les représentations apprises des couches précédentes. En résumé, un CNN apprend à transformer automatiquement l'image d'entrée en un ensemble de descripteurs hiérarchiques (via convolutions et poolings), puis effectue la décision de classification comme un perceptron ordinaire.

## 7.3 Fonctionnement de la convolution

Une couche de convolution opère en faisant glisser (scannant) chaque filtre sur l'image d'entrée. À chaque position, on effectue un produit scalaire entre les valeurs du filtre et les pixels correspondants de l'image, puis on somme ces produits pour obtenir un pixel de sortie dans la feature map. Ce processus est répété pour toutes les positions où le filtre peut s'appliquer. Par exemple, si l'image d'entrée fait  $4 \times 4$  et que le filtre fait  $3 \times 3$ , on peut positionner le filtre en 4 endroits, produisant une sortie de  $2 \times 2$ . La figure suivante illustre le concept : un filtre  $3 \times 3$  (rectangle rouge) glisse sur une portion de l'image, calculant en chaque endroit la résultat affiché dans la

carte de caractéristiques sur la droite (la valeur colorée correspond à l'endroit où le motif a été détecté).

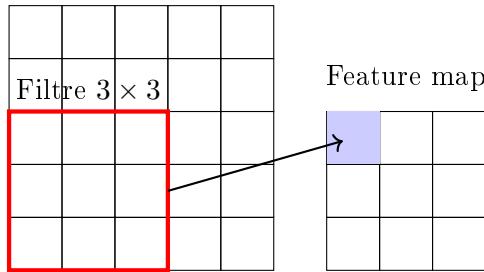


FIGURE 10 – Exemple de convolution : un noyau de taille  $3 \times 3$  balaie l'image d'entrée pour produire la carte de caractéristiques (feature map). Le pixel coloré dans la sortie correspond à la position où le motif du filtre a été fortement répondu.

Les filtres peuvent apprendre à détecter différents motifs locaux (bords orientés, textures, coins, etc.). Par exemple, les filtres dits de Sobel sont des détecteurs d'arêtes (un filtre vertical répond aux contours verticaux) et produisent des cartes mettant en évidence ces arêtes. De manière générale, un CNN va apprendre, grâce à la rétropropagation, quels filtres sont utiles pour la tâche cible. Dans une tâche de classification de chiffres manuscrits (MNIST), le réseau peut par exemple apprendre un filtre vertical qui répond fortement au chiffre 1 (deux arêtes verticales parallèles), facilitant ainsi sa reconnaissance.

#### 7.4 Padding et stride

Lorsqu'on applique un filtre de taille  $F \times F$  avec un stride de 1 sans aucune bordure, la sortie est plus petite que l'entrée. Pour contrer cela, on utilise souvent le *padding* : on ajoute  $P$  zéros autour de l'image d'entrée pour que le filtre puisse la balayer jusqu'aux bords. Par exemple, ajouter un padding de 1 pixel à une image  $4 \times 4$  avant d'y appliquer un filtre  $3 \times 3$  produit une sortie  $4 \times 4$  (même taille qu'initialement). Cette configuration, appelée « same padding », conserve la dimension spatiale. Sans padding (« valid padding »), la sortie est plus petite ( $O = I - F + 1$ ). Plus généralement, en notant  $I$  la taille d'entrée,  $F$  la taille du filtre,  $S$  le stride, et  $P$  le padding de chaque côté, la taille de la sortie est

$$O = \left\lfloor \frac{I - F + 2P}{S} \right\rfloor + 1,$$

Le paramètre *stride* détermine de combien de pixels le filtre se déplace à chaque pas. Par exemple, un stride de 2 décime pratiquement la taille spatiale (en sautant un pixel sur deux). En ajustant le padding et le stride, on contrôle ainsi finement la dimension en sortie et la quantité d'information conservée aux bords.

#### 7.5 Application aux images : exemple MNIST

Considérons l'exemple classique de classification des chiffres manuscrits (base MNIST) dont les images sont de taille  $28 \times 28$  pixels. Un CNN simple pour MNIST peut utiliser une première couche convolutionnelle avec, disons, 8 filtres de taille  $3 \times 3$ . Sans padding, cela transforme l'image  $28 \times 28$  en un volume  $26 \times 26 \times 8$  (car  $28 - 3 + 1 = 26$ ). On applique ensuite typiquement une couche de pooling (par exemple max pooling  $2 \times 2$ ) pour réduire à  $13 \times 13 \times 8$ , puis d'autres couches convolutionnelles et pooling. Finalement, on aplatis le volume et on passe à une ou plusieurs couches entièrement connectées pour la classification en 10 classes.

Dans ce contexte, on peut comparer le nombre de paramètres : un perceptron multi-couches traitant directement les 784 pixels de MNIST avec 100 neurones cachés aurait environ  $784 \times 100 \approx 7.8 \times 10^4$  poids. Le CNN exploitant les convolutions utilisera beaucoup moins de poids (par exemple  $3 \times 3$  filtres partagés) pour extraire des caractéristiques, puis quelques milliers de poids dans les couches finales. Cette efficacité paramétrique et la meilleure exploitation de la structure spatiale expliquent en partie la supériorité pratique des CNN sur les ensembles de données d'images.

## Conclusion

Les réseaux neuronaux convolutionnels constituent un outil fondamental en vision par ordinateur. Leur architecture, inspirée des procédés de traitement visuel biologiques, combine localité (champ réceptif) et partage de poids pour extraire automatiquement des représentations hiérarchiques d'images. Les CNN réussissent ainsi à capter des motifs invariants (bords, textures, formes) à différentes échelles et positions, rendant le modèle à la fois efficace en calcul et robuste. Les exemples, comme la classification MNIST, montrent qu'un CNN peut surpasser un perceptron classique en exploitant au mieux la structure des données visuelles.

## 8 Modèles CNN connus pour le computer vision et bonnes pratiques

Les réseaux de neurones convolutifs (CNN) occupent une place centrale dans le domaine de la vision par ordinateur. Conçus spécifiquement pour exploiter la structure locale des données, en particulier les images, ils permettent d'extraire automatiquement des caractéristiques discriminantes à différentes échelles. L'une des premières architectures marquantes, LeNet-5, a été développée par Yann LeCun à la fin des années 1990 pour la reconnaissance de chiffres manuscrits dans la base MNIST. Depuis, de nombreuses architectures ont été proposées, chacune introduisant des améliorations architecturales significatives. Cette évolution a été rendue possible par l'augmentation progressive de la puissance de calcul disponible, notamment grâce à l'avènement des processeurs graphiques (GPU), qui ont permis l'entraînement de réseaux toujours plus profonds sur des ensembles de données volumineux.

### 8.1 Architectures célèbres

- **LeNet-5** : premier CNN célèbre (MNIST), architecture simple.
- **AlexNet** : ImageNet 2012, popularisation du deep learning (première fois que le deep learning était plus performant que les méthodes alternatives).
- **VGG** : empilement de petits filtres  $3 \times 3$ , très profond.
- **ResNet** : introduction des connexions résiduelles (skip connections).
- **MobileNet** : optimisé pour les appareils embarqués.
- **EfficientNet** : équilibre entre largeur, profondeur et résolution.

### 8.2 Bonnes pratiques pour les CNN

- Normaliser les entrées (valeurs centrées, réduites)

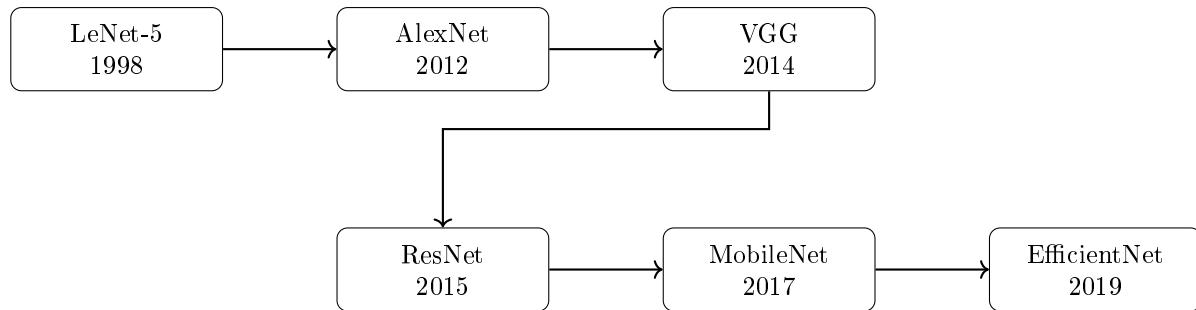


FIGURE 11 – Chronologie des principales architectures CNN utilisées en vision par ordinateur.

- Ajouter du dropout après les couches denses pour régulariser
- Utiliser le batch normalization
- Privilégier des kernels petits mais profonds
- Ajuster le learning rate avec un scheduler (fort au début et faible ensuite)

Une approche largement répandue dans les applications de vision par ordinateur consiste à utiliser un modèle préentraîné pour l'extraction automatique de caractéristiques (features), tout en ne réentraînant que la partie finale du réseau, généralement constituée de couches entièrement connectées dédiées à la classification. Dans ce cadre, le modèle préexistant agit principalement comme un extracteur de descripteurs visuels à partir des images d'entrée. La frontière entre les couches gelées (non modifiées) et les couches réentraînées peut cependant être modulée : il est fréquent de "dégeler" (unfreeze) les dernières couches convolutives afin d'adapter plus finement les représentations apprises aux spécificités du jeu de données ciblé. Cette stratégie permet de tirer profit des connaissances encapsulées dans des architectures entraînées sur de très grands corpus d'images (parfois plusieurs millions), tout en assurant une spécialisation efficace du modèle pour une tâche particulière. Elle illustre l'un des grands atouts du deep learning contemporain : la possibilité de transférer efficacement des représentations apprises dans un contexte générique vers des problématiques plus ciblées.

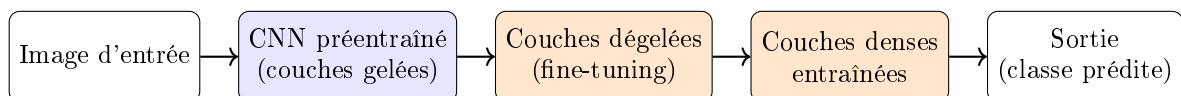


FIGURE 12 – Architecture typique avec transfert d'apprentissage : seules les couches finales sont réentraînées. Les premières couches convolutives restent gelées ou partiellement dégelées selon les besoins.

Nous vous invitons à explorer régulièrement le site [selvasystems.net](http://selvasystems.net), où nous documentons nos développements en cours. L'un des projets en cours consiste à concevoir un système de détection, de suivi et de pointage laser de cibles reposant sur le modèle YoLo, déployé à la fois sur un ordinateur monocarte de très petite taille (Orange Pi Zero 3) et sur un accélérateur TPU Google Coral (il est en cours de réalisation). Ce projet illustre qu'il est aujourd'hui possible, avec moins de 150 euros de matériel, de réaliser de l'acquisition vidéo, de l'inférence en temps réel à 20 Hz, et de piloter deux actionneurs pour orienter un laser avec une précision suffisante pour le pointage dynamique de cibles mobiles. Ces avancées techniques, qui rendent accessibles des inférences complexes à très bas coût, demeurent encore largement sous-exploitées dans le monde industriel aujourd'hui. L'activité de Selva Systems se concentre précisément sur ce type d'applications embarquées à forte valeur ajoutée technologique.

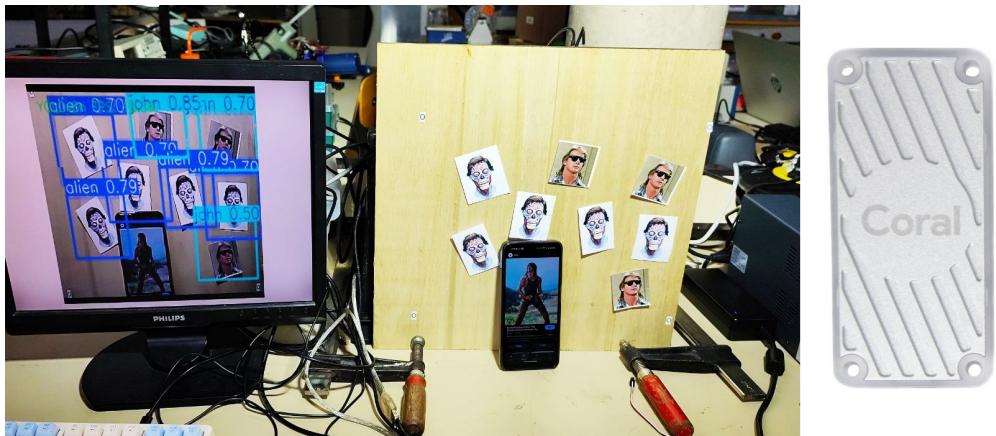


FIGURE 13 – Exemple d'identification des extraterrestres et du héros John Nada (clin d'œil au chef-d'œuvre du 7<sup>e</sup> art *They Live* de John Carpenter). Les inférences sont exécutées sur un Orange Pi Zero 3 équipé d'un Coral USB. Le modèle a été quantifié en UINT8 pour rendre son déploiement compatible avec l'utilisation du Coral USB. Nous avons atteind un taux d'inférence de 17 fps, ce qui est très bon pour une solution si légère.

#### En résumé

Les CNN permettent de capter efficacement les régularités locales dans des données structurées, qu'elles soient spatiales (images) ou temporelles (signaux). Ils sont au cœur des applications modernes de vision, reconnaissance d'activité, analyse audio, etc. On peut capitaliser sur les réseaux connus pour adresser des problèmes locaux.

## 9 Une utilisation très riche des CNN pour l'embarqué : l'analyse de séquences temporelles

Bien que les réseaux de neurones convolutifs (CNN) aient été conçus à l'origine pour le traitement d'images, ils sont aujourd'hui largement utilisés dans des contextes très variés, y compris pour l'analyse de données séquentielles. Cette généralisation est particulièrement utile en contexte embarqué, où les ressources sont limitées et où les CNN présentent des avantages décisifs par rapport à d'autres architectures.

### Applications typiques de l'analyse temporelle

Dans l'embarqué, de nombreuses applications consistent à traiter des séquences de mesures dans le temps :

- analyse de données accélérométriques pour reconnaître une activité physique (marche, course, repos, chute) ;
- diagnostic machine à partir de vibrations ou sons (maintenance prédictive) ;
- surveillance environnementale (température, pression, humidité) ;
- traitement audio (reconnaissance de mot clé, détection de bruits particuliers) ;
- séries temporelles multivariées (prévision météo, consommation d'énergie).

## Pourquoi utiliser des CNN ?

Contrairement aux réseaux spécialisés pour les séquences (RNN, LSTM, GRU, Transformers), les CNN offrent une exécution bien plus efficace sur des plateformes embarquées. Leur parallélisme naturel les rend compatibles avec des bibliothèques d'inférence optimisées, y compris sur microcontrôleurs (ex : TensorFlow Lite Micro, CMSIS-NN).

Il est possible de traiter une séquence temporelle multivariée comme une image 2D : chaque ligne correspond à un pas de temps, chaque colonne à une variable. On applique alors une convolution sur une seule ligne (convolution 1D généralisée) — ce qui revient à utiliser un kernel étroit et haut (par exemple  $(5 \times 1)$ ). Ce type de filtrage local dans le temps permet d'identifier des motifs caractéristiques sur de courtes fenêtres.

Plus on empile les couches convolutionnelles, plus on augmente le champ réceptif du réseau — autrement dit, la capacité à détecter des motifs complexes à différentes échelles temporelles (on détecte les motifs temporels, plus les motifs de motifs dans le couche 2, plus les motifs de motifs de motifs dans la couche suivante...). C'est ce principe qui permet aux CNN d'être très efficaces pour l'analyse de séquences, tout en restant légers en mémoire et rapides à l'exécution.

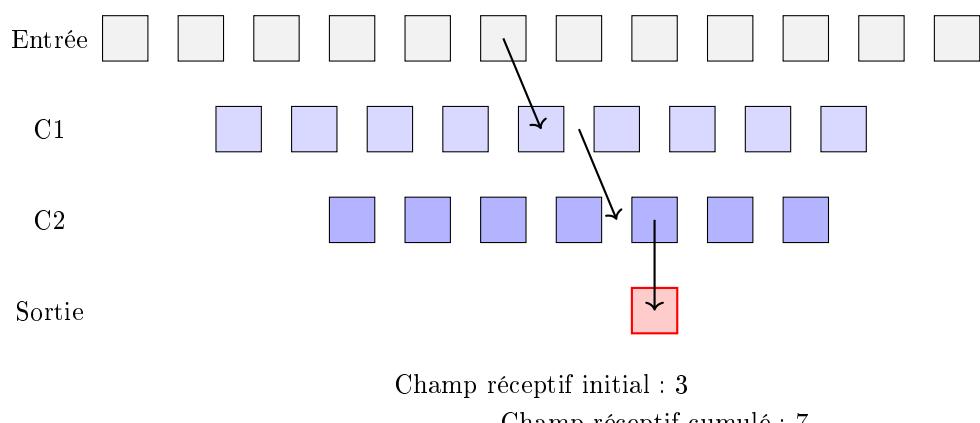


FIGURE 14 – Illustration du champ réceptif croissant dans un CNN 1D. À chaque couche, le réseau “voit” un contexte temporel plus large, ce qui permet de capturer des motifs plus complexes.

## Un exemple déployé sur ESP32

Nous avons appliqué cette méthode à la prévision de température à court terme à partir de séries météorologiques. Le modèle utilise des séquences de 12 heures comme entrée et prédit les 4 heures suivantes. Une seule couche convolutionnelle  $(5 \times 1)$  suffit pour capturer les variations locales. Le modèle, une fois quantifié et converti, est embarqué sur une carte ESP32. Il exécute une inférence complète en moins de 200 ms, démontrant la faisabilité de telles approches dans un environnement contraint.



FIGURE 15 – Exemple d'utilisation réelle d'un modèle CNN 1D embarqué pour la prévision de température. Le modèle est entraîné à partir de données publiques, converti au format **TensorFlow Lite**, puis déployé sur un microcontrôleur ESP32. Ce type d'application illustre la capacité des CNN à traiter efficacement des séquences temporelles multivariées dans un contexte à ressources limitées. Le boîtier embarqué intègre un capteur environnemental permettant de mesurer la température, la pression et l'humidité relative. La température de rosée est calculée localement à partir de ces données. Un module RTC (horloge temps réel) fournit l'heure et la date, deux variables clés dans la prévision météorologique. Plus d'informations sont disponibles sur [selvasystems.net/deep-learning](http://selvasystems.net/deep-learning).

Modèle	Taille	Vitesse	Déploiement	Performance sur tâche simple
CNN	★★★	★★★★★	★★★★★	★★★★★
LSTM	★★	★★	★	★★★★★
Transformer	★★★★★	★	Incompatible	★★★★★

FIGURE 16 – Comparatif des réseaux séquentiels pour le traitement embarqué. Les CNN sont souvent le meilleur compromis entre simplicité, efficacité et compatibilité matérielle.

### À retenir

Les CNN sont parfaitement adaptés à l'analyse de séquences temporelles embarquées. Bien que des architectures plus complexes existent (LSTM, Transformers), elles sont rarement justifiées dans ce contexte : plus lentes, plus lourdes, et souvent pas plus performantes. Des CNN simples à kernels étroits, profonds, et bien entraînés permettent déjà une détection efficace de motifs temporels dans des flux de données capteurs.

## 10 Keyword Spotting (KWS) et CNN

La qualité des systèmes de reconnaissance vocale repose en grande partie sur l'extraction de features, une étape essentielle qui transforme un RAW de signal audio en une représentation numérique adaptée au traitement du signal. Pendant des décennies, les MFCC (Mel-Frequency Cepstral Coefficients) ont été l'outil de prédilection pour cette tâche. Cependant, l'essor des réseaux de neurones a ouvert la voie à de nouvelles méthodes, notamment en utilisant les réseaux de neurones convolutifs.

La reconnaissance vocale est aujourd'hui omniprésente, du contrôle vocal des appareils domestiques aux assistants virtuels. Au cœur de ces technologies se trouve la détection de mots-clés, connue sous le nom de Keyword Spotting (KWS). Souvent, les solutions techniques font intervenir des dispositifs « en cascade », où le premier élément de la chaîne est un système embarqué léger. Par exemple, un microcontrôleur à faible consommation d'énergie reste en veille pour détecter l'apparition d'un mot-clé. Dès que ce mot-clé est détecté, il réactive l'intégralité du système, permettant un traitement plus complexe du signal.

### 10.1 Les MFCC : les premiers pas de la reconnaissance vocale

Les MFCC ont été introduits dans les années 1980, à une époque où la reconnaissance vocale était principalement fondée sur des modèles statistiques. L'idée derrière le MFCC est d'imiter la perception humaine du son, qui ne réagit pas linéairement aux fréquences. Notre oreille est plus sensible aux variations dans les basses fréquences qu'aux hautes fréquences, une caractéristique que le MFCC cherche à reproduire.

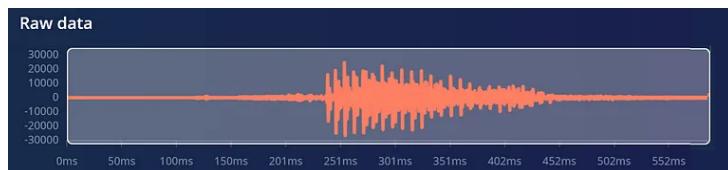


FIGURE 17 – Exemple d'un signal audio brut (*raw audio*) d'une seconde, acquis à 16 kHz, correspondant à l'enregistrement d'un mot.

Le processus d'obtention des MFCC commence par la division du signal audio en petites trames temporelles. Pour chaque trame, on applique une transformée de Fourier (FFT) pour obtenir le spectre fréquentiel. Ce spectre est ensuite filtré à travers une série de filtres triangulaires répartis selon une échelle de Mel, qui reflète la perception humaine des distances fréquentielles. L'étape finale consiste à appliquer une transformée en cosinus discrète (DCT) sur le spectre de Mel obtenu, ce qui génère les coefficients cepstraux.

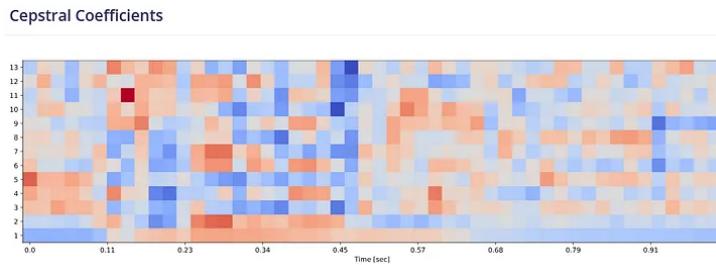


FIGURE 18 – Représentation cepstrale (cepstrum) correspondant à l'extrait audio présenté précédemment. On observe qu'un nombre limité de coefficients suffit à caractériser le signal : environ 650 paramètres indépendants sont nécessaires pour une seconde d'enregistrement.

Le cepstrum est une transformation qui consiste en quelque sorte à prendre le spectre d'un spectre. Lorsqu'on parle, les cordes vocales (situées dans le larynx) vibrent et produisent un son de base, souvent appelé son source ou excitation. Ce son source se compose d'une série d'harmoniques (fréquences multiples de la fréquence fondamentale). Ce son passe ensuite par le conduit vocal, qui agit comme un filtre, modifiant ce son brut en fonction de la forme et de la configuration du conduit à ce moment-là : position de la langue, ouverture de la bouche, forme du palais et d'autres éléments influencent ce filtrage. Ce processus de filtrage donne naissance aux formants, qui sont des pics dans le spectre de fréquences, caractérisant les sons de la parole (comme les voyelles et certaines consonnes). Le cepstrum permet de séparer ces deux composantes importantes du signal vocal : l'excitation et la partie filtrage. Cette séparation est cruciale en reconnaissance vocale, car elle permet de distinguer les caractéristiques liées au contenu linguistique (comment les sons sont produits et perçus) des caractéristiques liées à la source sonore elle-même.

Bien que les MFCC soient très efficaces, ils présentent certains inconvénients. Le processus est relativement complexe et exigeant en termes de calcul, ce qui peut poser des défis pour les applications embarquées. De plus, en résumant le signal audio en peu de coefficients, certaines nuances fines du spectre sonore sont inévitablement perdues, limitant ainsi la précision des systèmes de reconnaissance vocale. À titre d'exemple, une seconde d'audio, acquise à 16 kHz, correspond à environ 600 valeurs flottantes (avec 13 coefficients dans le filtre pour chaque trame de 20 ms).

## 10.2 L'Évolution des Réseaux de Neurones

L'évolution des réseaux de neurones, et en particulier l'essor des réseaux convolutionnels (CNN), a bouleversé la manière dont on appréhende de nombreux problèmes. Le point fort des réseaux de neurones réside dans leur capacité à fonctionner avec un grand nombre de données d'entrée. Cette caractéristique permet d'aborder le KWS différemment, en diminuant le coût calculatoire consacré à l'extraction de features et en contrepartie, en augmentant la taille du réseau de neurones chargé du traitement du son à partir des features extraites. Contrairement aux MFCC, qui nécessitent plusieurs étapes de transformation et de filtrage, les spectrogrammes offrent une représentation plus directe du signal audio. Un spectrogramme est essentiellement une série de spectres de Fourier empilés dans le temps, permettant de visualiser comment les fréquences d'un signal évoluent. Cette représentation est particulièrement adaptée aux réseaux de neurones, qui excellent à extraire des motifs complexes à partir de données brutes.

L'utilisation des spectrogrammes simplifie le pipeline de traitement du signal. Cependant, l'extraction de features génère beaucoup plus de paramètres. À titre d'exemple, une seconde d'audio à 16 kHz de fréquence d'échantillonnage génère plusieurs milliers de paramètres. Une telle quantité de données ne pourrait être exploitée sans les capacités récentes offertes par les réseaux de

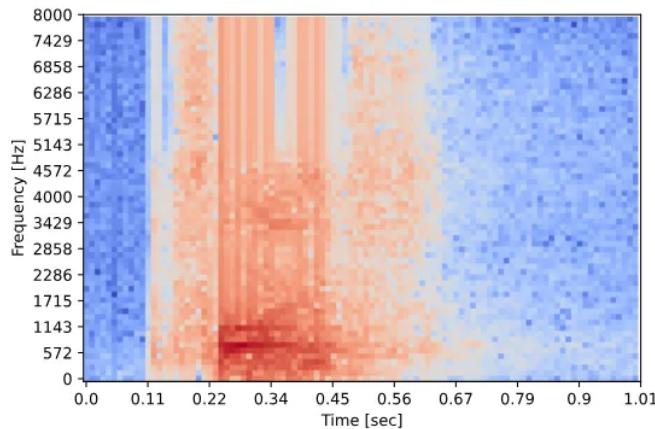


FIGURE 19 – Spectrogramme du même extrait audio. Cette représentation visuelle du signal est utilisée comme entrée pour l'entraînement de réseaux convolutifs. Elle s'adapte aussi très bien à la détection de bruits non vocaux, tels que des événements acoustiques spécifiques. Un exemple courant est l'utilisation de spectrogrammes pour des alarmes intelligentes capables de détecter le son caractéristique d'un bris de glace.

neurones, et en particulier les CNN : les capacités d'apprentissage des modèles complexes ont considérablement augmenté.

Il y a donc un compromis à trouver. Le but est de rendre le KWS le plus léger et rapide possible. Passer des MFCC aux spectrogrammes permet de gagner en temps de calcul pour l'extraction des features, mais au détriment du temps de calcul et de l'occupation mémoire du réseau de neurones, ainsi que de l'occupation mémoire en général, car il y a beaucoup plus de features à gérer. Cela est particulièrement vrai si l'on utilise les fonctions DSP du microcontrôleur, qui nécessitent souvent de recourir à des tampons pour stocker les vecteurs intermédiaires lors des calculs. Néanmoins, l'avantage semble pencher en faveur d'une méthode d'identification reposant sur les spectrogrammes. À titre d'exemple, sur un ESP32-S3 équipé d'un micro MEMS, pour détecter le mot « lumière », j'ai réduit le temps d'inférence d'un facteur supérieur à 10 en passant de la méthode basée sur les MFCC (proposée par Edge Impulse) à celle basée sur les spectrogrammes. Il est à noter que ce bon résultat est également permis par l'usage généralisée des fonctions DSP pour l'extraction des features.

### 10.3 Les Spectrogrammes Mel et Techniques Hybrides

Il paraît logique, même si ce n'est pas forcément la méthode la plus efficace, de tenter de générer un système dont la perception se rapproche au plus de celle de l'oreille humaine. C'était l'idée sous-jacente aux MFCC. Pour tirer parti des avantages des spectrogrammes tout en tenant compte des spécificités de la perception humaine, une approche intermédiaire a émergé : les spectrogrammes Mel. Cette technique conserve la richesse d'information et la simplicité calculatoire des spectrogrammes tout en appliquant une échelle de Mel au spectrogramme, offrant en quelque sorte un compromis entre les deux méthodes. En parallèle, d'autres techniques hybrides ont été explorées. Par exemple, certains systèmes combinent des features issues des MFCC avec des spectrogrammes pour enrichir la représentation du signal, capitalisant sur les points forts de chaque méthode. D'autres approches, telles que les Gammatone Cepstral Coefficients (GTCC) ou la Constant-Q Transform (CQT), offrent une alternative aux MFCC et aux spectrogrammes, en proposant des compromis adaptés aux contraintes spécifiques des systèmes embarqués.

## 10.4 Systèmes Embarqués et KWS : L'Art du Compromis

L'un des principaux défis des systèmes embarqués est de trouver un équilibre entre la complexité du traitement des features et les ressources limitées en termes de calcul, de mémoire et d'énergie. Les spectrogrammes, qui simplifient l'étape de prétraitement, peuvent sembler être une solution idéale. Cependant, la quantité de données qu'ils génèrent nécessite des réseaux neuronaux plus grands et plus profonds, ce qui peut être difficilement compatible avec les contraintes des systèmes embarqués.

Quoiqu'il en soit, on retrouve ici une dynamique propre à d'autres domaines du deep learning : l'approche « imitant » les perceptions humaines n'est pas toujours la meilleure, ou du moins, pas la seule voie possible. Ce principe s'observe également dans d'autres domaines du deep learning, comme la vision par ordinateur, où des méthodes non basées sur la perception humaine, telles que les CNN appliquées directement aux images, surpassent souvent les techniques basées sur des transformations perceptuelles, comme les filtres de détection de contours.

L'émergence des réseaux neuronaux a bouleversé le travail des ingénieurs. Autrefois, on cherchait à minimiser le nombre de paramètres influents sur un problème pour en proposer une modélisation pertinente. Aujourd'hui, avec les réseaux de neurones, il s'agit au contraire de fournir une grande quantité de données et de « faire confiance » à la méthode pour extraire les informations pertinentes et identifier les paramètres influents. Aujourd'hui, les approches hybrides, qui allient le meilleur des techniques basées sur la perception de l'oreille humaine et les capacités des réseaux CNN, représentent probablement l'avenir de cette discipline. Elles restent cependant un enjeu pour la recherche, car une bonne marge de progression semble encore possible pour rendre le KWS encore plus léger, rapide et précis.

## 10.5 Implémentation sur l'ESP32

Dans les exercices vous retrouverez un exemple complet d'implémentation de KWS sur l'ESP32 en temps réel. Il y a plusieurs défis :

- Gestion du temps réel avec double buffering. En réalité il y a 4 buffers audio de 250 ms qui sont remplis tour à tour. Cela permet d'exécuter toutes les 250 ms une inférence sur 1s d'audio. Là est la clef de la réactivité du système.
- Les fonctions DSP FFT de l'ESP32 sont mis à profit pour extraire les features d'entrée du modèle et construire les spectrogrammes
- Les spectrogrammes sont analysés à la manière d'une image par le réseau de neurone convolutif
- Le réseau fonctionne sous tensorflow lite et une librairie forké par Espressif pour bénéficier des optimisations matérielles propres à l'ESP32 (fonctions vectoriels SIMD, DSP).

Vous pouvez voir une démonstration du système en fonctionnement ici : [https://www.youtube.com/watch?v=bf5jzSzD\\_Q](https://www.youtube.com/watch?v=bf5jzSzD_Q). On peut observer les résultats d'inférence toutes les 250 ms avec la valeur des 3 neurones de sortie représentant la probabilité respectivement de bruit, de parole « non lumière » et du mot lumière.

## 11 Quantification et pruning : comment rendre un modèle compatible avec du matériel léger

L'un des principaux défis du déploiement de modèles de deep learning sur des dispositifs embarqués réside dans leurs contraintes matérielles : mémoire limitée, puissance de calcul réduite, consommation énergétique restreinte. Pour rendre un réseau de neurones compatible avec ces environnements, deux techniques complémentaires sont couramment utilisées : la **quantification** et le **pruning**.

### 11.1 Quantification

La quantification consiste à représenter les poids et/ou les activations du réseau avec une précision réduite, par exemple en utilisant des entiers sur 8 bits (int8) au lieu de nombres flottants en 32 bits (float32). Cela permet :

- de réduire la taille du modèle en mémoire ;
- d'accélérer les calculs, notamment sur des accélérateurs matériels comme le Coral TPU ou les microcontrôleurs dotés de SIMD ;
- de diminuer la consommation énergétique.

Des outils tels que *TensorFlow Lite*, *PyTorch Mobile* ou *ONNX Runtime* permettent d'automatiser cette conversion. Il existe plusieurs niveaux de quantification :

- **Post-training quantization** : la quantification est appliquée après l'entraînement sans modifier les poids ;
- **Quantization-aware training (QAT)** : le modèle est entraîné en intégrant la contrainte de quantification, ce qui préserve mieux les performances.

### 11.2 Pruning

Le pruning (élagage) vise à supprimer les connexions ou les neurones qui ont peu d'impact sur la sortie du réseau. Il en résulte un modèle plus léger, souvent plus rapide à exécuter. Deux approches principales existent :

- **Pruning non structuré** : suppression de poids isolés ; efficace mais peu compatible avec l'optimisation matérielle.
- **Pruning structuré** : suppression de neurones ou de filtres entiers ; mieux adapté au déploiement embarqué.

Le pruning est souvent suivi d'une étape de *fine-tuning* pour restaurer partiellement les performances perdues.

### 11.3 Combinaison des deux approches

La quantification et le pruning peuvent être combinés pour obtenir des gains significatifs en taille mémoire et en vitesse, tout en maintenant une précision acceptable. Ces techniques sont au cœur des stratégies d'optimisation pour l'embarqué et sont indispensables pour déployer des modèles performants sur des plateformes telles que l'ESP32, le Raspberry Pi, ou les accélérateurs Edge TPU.

### En résumé

La quantification et le pruning permettent de réduire fortement les ressources nécessaires à l'exécution d'un modèle de deep learning. Appliquées correctement, elles rendent possible le déploiement d'architectures avancées sur des dispositifs légers, ouvrant ainsi la voie à une intelligence artificielle véritablement embarquée.

Pour tout complément d'information, Selva Systems reste à votre disposition :  
[contact@selvasystems.net](mailto:contact@selvasystems.net) / 06 77 02 79 30.