CHAPITRE CNN · FORMATION DEEP LEARNING PAR LA PRATIQUE

Intuition sur les CNN

Comprendre pas à pas ce que fait un réseau convolutif

Ce document suit, étape par étape, un petit réseau convolutif (CNN) qui classe des images couleur de 32×32 pixels en 10 classes. Chaque étape est expliquée avec des exemples chiffrés, et trois questions qui viennent naturellement y sont intégrées :

- pourquoi la carte obtenue après une convolution est **plus petite** que l'image (étape 2) ;
- pourquoi on choisit **32 filtres** (étape 3) ;
- comment un réseau qui **perd de l'information** peut détecter des motifs **plus riches** (section 10).

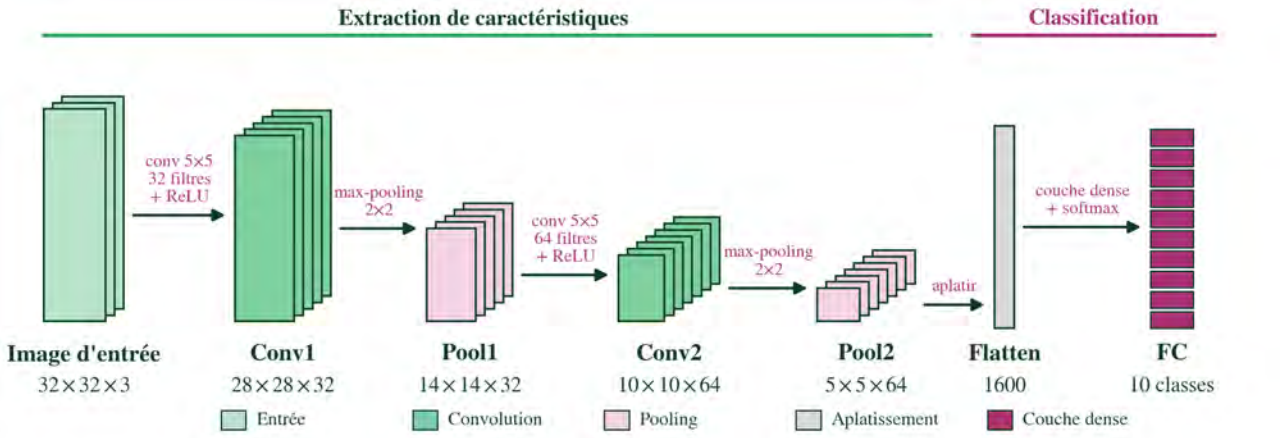

Figure 1 – Architecture du réseau étudié

> **Hypothèses du réseau** : filtres 5×5 de pas 1, sans remplissage ; max-pooling 2×2 de pas 2 ; ReLU après chaque convolution ; dernière couche dense à 10 sorties suivie d'une softmax.

Étape 0 – Ce que voit l'ordinateur

Une image couleur de 32×32 pixels est stockée sous la forme de **trois grilles de nombres** de 32×32 : une pour le rouge, une pour le vert, une pour le bleu. On écrit ses dimensions **32×32×3** : hauteur × largeur × canaux. Chaque nombre est l'intensité d'un pixel dans un canal (de 0 à 255, ramenée en général à une petite échelle).

Étape 1 – Pourquoi ne pas utiliser un réseau classique ?

Un réseau entièrement connecté doit aplatir l'image en 3 072 nombres. Il perd alors l'information « ce pixel est voisin de celui-là », et son nombre de paramètres explose. Pour produire les 28×28×32 = 25 088 valeurs de la première couche, il faudrait :

Couche	Paramètres
Couche dense : 3 072 entrées vers 25 088 sorties	77 095 424
Convolution : 32 filtres de 5×5×3	2 432

La convolution fait la même chose avec environ **31 700 fois moins** de paramètres, parce qu'elle regarde de petites zones et réutilise les mêmes poids partout.

Étape 2 – La convolution

2.1 Le principe

Un **filtre** est une petite grille de poids (par exemple 3×3). On le pose sur une zone de l'image, on multiplie chaque poids par le pixel situé en dessous, on additionne tout, et cela donne **un seul nombre**. On décale ensuite le filtre d'un pixel et on recommence : l'ensemble des résultats forme une **carte**.

2.2 Un exemple : détecter un bord vertical

Prenons une image 6×6 dont la moitié gauche est claire (valeur 10) et la moitié droite sombre (valeur 0), et le filtre ci-dessous, qui calcule « ce qu'il y a à gauche moins ce qu'il y a à droite ».

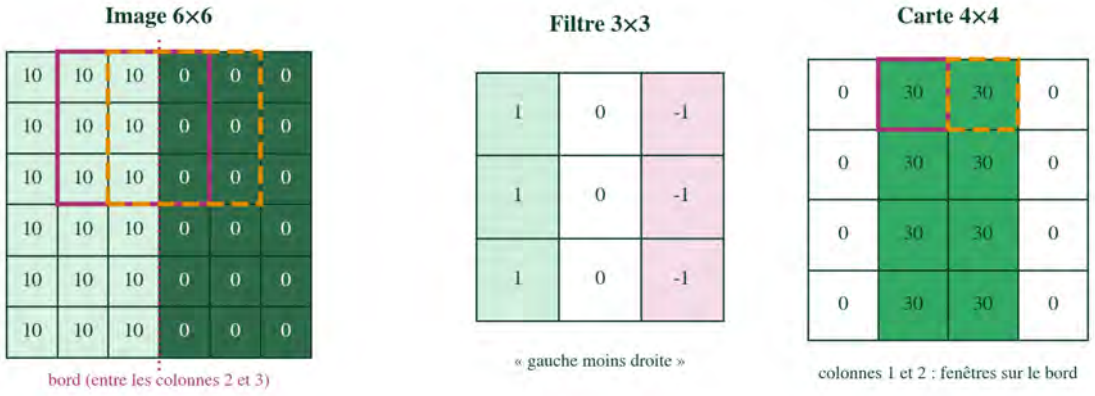

Figure 2 – Image, filtre et carte obtenue. Les deux fenêtres encadrées (magenta et orange) correspondent aux deux cases encadrées de la carte

Les valeurs fortes (30) apparaissent là où le filtre a rencontré un changement du clair au sombre. Un filtre est donc un **détecteur d'un motif précis**. Dans un CNN, ces poids ne sont pas choisis à la main : ils sont **appris** par l'entraînement.

2.3 Une case de la carte résume une fenêtre, pas un pixel

La carte (4×4) n'a pas la même taille que l'image (6×6), car chaque case de la carte correspond à une **fenêtre** de 3×3 pixels. La case de colonne j est calculée à partir des colonnes j, j+1 et j+2 de l'image. Voici la première ligne de l'image, `10 10 10 0 0 0`, vue fenêtre par fenêtre :

Case (colonne)	Colonnes de l'image	Valeurs vues	Résultat	Interprétation
0	0, 1, 2	10 10 10	0	Zone uniformément claire
1	1, 2, 3	10 10 0	30	La fenêtre contient le bord
2	2, 3, 4	10 0 0	30	La fenêtre contient le bord
3	3, 4, 5	0 0 0	0	Zone uniformément sombre

> **Formulation exacte** : les valeurs fortes (30) apparaissent pour les **fenêtres qui recouvrent le bord**, et 0 pour les fenêtres situées dans une zone uniforme. Le bord est entre les colonnes 2 et 3 de l'image : seules les fenêtres des colonnes 1 et 2 de la carte le recouvrent.

Pour retrouver une position dans l'image, on associe chaque case de la carte au **centre** de sa fenêtre : la case de colonne j correspond au pixel de colonne j + 1. Les deux cases à 30 sont donc centrées sur les colonnes 2 et 3, de part et d'autre du bord.

Pourquoi la carte est plus petite : le filtre doit rester entièrement dans l'image, donc il n'a que 6 − 3 + 1 = 4 positions possibles par ligne. Règle générale : **n − k + 1** (pas de 1, sans remplissage).

2.4 Garder la taille : le remplissage (padding)

En entourant l'image d'une bordure de zéros avant le calcul, le filtre peut être centré sur chaque pixel, y compris ceux des bords : la carte redevient 6×6 et la case (i, j) correspond directement au pixel (i, j). Sur l'exemple, on obtient (pour les lignes du milieu) :

```
-30   0  30  30   0   0
-30   0  30  30   0   0
-30   0  30  30   0   0
```

Les 30 sont maintenant aux colonnes **2 et 3**, c'est-à-dire de part et d'autre du bord, à la position même de l'image. Le −30 de la colonne 0 est un **effet de bord** : la bordure de zéros crée une frontière artificielle avec le début de l'image claire, que le filtre détecte (avec le signe opposé). En Keras, c'est `padding="same"` ; en PyTorch, `padding=2` pour un filtre 5×5. Le réseau étudié n'utilise **pas** de remplissage : c'est pourquoi la taille diminue à chaque convolution.

Étape 3 – La convolution sur une image couleur (Conv1)

3.1 Un filtre couvre tous les canaux

Le filtre doit couvrir les **trois canaux** : il est donc de taille **5×5×3**, soit **75 poids** (plus un biais). À chaque position, il multiplie ses 75 poids par les 75 pixels de la zone 5×5 (dans les trois canaux) et additionne le tout : **un seul nombre**. En balayant l'image, un filtre produit **une carte** de 28×28 (car 32 − 5 + 1 = 28). Avec **32 filtres différents**, chacun apprend un motif différent ; on empile les 32 cartes et on obtient **28×28×32**.

À ne pas confondre	Valeur	Signification
Canaux de l'entrée	3	Rouge, vert, bleu
Profondeur de chaque filtre	3	Elle est toujours égale au nombre de canaux de l'entrée
Nombre de filtres	32	C'est ce choix qui fixe la profondeur de la sortie

Paramètres de Conv1 : chaque filtre a 5×5×3 + 1 = 76 paramètres, donc 32 × 76 = **2 432**. Ce nombre ne dépend pas de la taille de l'image, parce que le même filtre sert à toutes les positions : c'est le **partage de paramètres**.

3.2 Pourquoi 32 filtres ?

Le nombre de filtres n'est pas dicté par une formule : c'est un **hyperparamètre**, un choix de conception, comme le nombre de neurones d'une couche dense. Chaque filtre apprend un motif différent ; avoir 32 filtres

revient à demander au réseau d'apprendre 32 détecteurs de motifs simples. Ce nombre fixe donc le « vocabulaire » de la couche et la quantité d'information transmise à la suivante.

Pas assez de filtres : le réseau ne distingue pas assez de motifs (sous-apprentissage). **Trop de filtres** : plus de paramètres et de calcul, risque de sur-apprentissage, filtres redondants, entraînement plus lent. Voici l'effet sur ce réseau (calculé en faisant varier les filtres de Conv1 et de Conv2) :

Filtres (Conv1, Conv2)	Paramètres	Multiplications-additions
(8, 16)	7 834	0,8 million
(16, 32)	22 058	2,2 millions
(32, 64) : notre réseau	69 706	7,0 millions
(64, 128)	241 802	24,2 millions

Doubler le nombre de filtres multiplie le calcul et les paramètres par environ 3,5. C'est un compromis entre capacité et coût : sur un CPU, un réseau plus petit s'entraîne plus vite.

Pourquoi plus de filtres en profondeur (32, puis 64) ? Deux raisons. D'abord, les motifs se multiplient : la première couche ne détecte que quelques motifs élémentaires (bords, couleurs), alors que les couches suivantes les **combinent** en formes bien plus nombreuses (coins, courbes, textures). Ensuite, la taille spatiale diminue avec le pooling (28×28, 14×14, 5×5) : on augmente le nombre de canaux pour conserver une capacité de représentation suffisante. Les architectures de référence suivent cette règle :

Réseau	Filtres des couches successives
LeNet-5 (1998)	6, puis 16
AlexNet (2012)	96, 256, 384, 384, 256
VGG16 (2014)	64, 128, 256, 512, 512
ResNet (2015)	64, puis 64, 128, 256, 512

Les puissances de 2 (16, 32, 64, 128…) sont une habitude : les bibliothèques de calcul les traitent un peu plus efficacement et on garde une progression simple (on double d'une couche à l'autre). Ce n'est pas une obligation.

> **Comment on choisit en pratique** : on part d'une valeur courante (32 ou 64 pour la première couche), on entraîne, puis on compare les courbes d'apprentissage et de validation. Si le modèle sous-apprend, on augmente le nombre de filtres ; s'il sur-apprend, on le réduit ou on régularise. Les comparaisons se font sur le jeu de **validation**, jamais sur le test.

Étape 4 – La ReLU

La ReLU s'applique à **chaque nombre** des 32 cartes : ReLU(x) = max(0, x). Les valeurs négatives deviennent 0, les valeurs positives restent inchangées. La forme ne change pas (28×28×32) et il n'y a aucun paramètre. Son rôle est d'introduire la **non-linéarité** : sans elle, les couches successives équivaudraient à une seule opération linéaire.

Étape 5 – Le pooling (Pool1)

On découpe chaque carte en blocs de 2×2 et on **garde la valeur maximale** de chaque bloc :

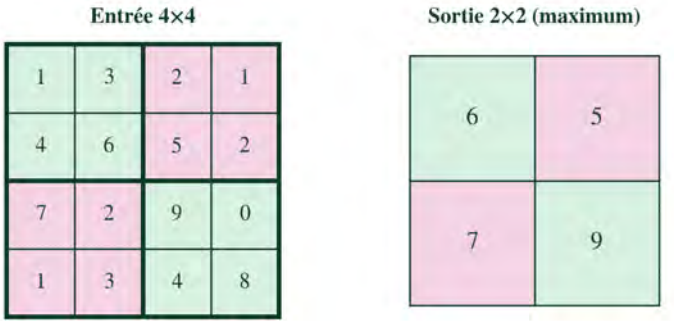

Figure 3 – Max-pooling 2×2 : on garde le maximum de chaque bloc

Chaque carte passe de 28×28 à **14×14**, et le nombre de cartes ne change pas (32) : le pooling agit **carte par carte**. Il n'a aucun paramètre. Il réduit le calcul (4 fois moins de valeurs) et rend le résultat peu sensible à un petit décalage : si un motif bouge d'un pixel, le maximum du bloc reste souvent le même. Sortie : **14×14×32**.

Étape 6 – Conv2 : des motifs de motifs

L'entrée de Conv2 n'est plus l'image, mais les **32 cartes** produites par Conv1 et réduites par Pool1. Chaque filtre doit donc couvrir ces 32 cartes : il est de taille **5×5×32**, soit 800 poids et 1 biais. Il y a **64 filtres**, donc 64 cartes de sortie, de taille 14 − 5 + 1 = 10 : **10×10×64**. Paramètres : 64 × (800 + 1) = **51 264**.

Conv1 détecte des motifs simples (bords, couleurs). Conv2 les **combine** pour détecter des motifs plus complexes (coins, textures, courbes). Chaque neurone de Conv2 « voit » une zone de **14×14 pixels** de l'image d'origine, car il regarde 5×5 cartes dont chaque case résume déjà une zone qui se chevauche. Plus on avance, plus les caractéristiques sont abstraites et plus la zone observée est large.

Étape 7 – Pool2

Même opération : 10×10×64 devient **5×5×64**. Chaque case de cette carte résume une zone de 16×16 pixels de l'image.

Étape 8 – Aplatissement, couche dense et softmax

- **Aplatissement (Flatten)** : les 5×5×64 = **1 600 nombres** sont mis bout à bout en un vecteur. Aucun calcul, aucun paramètre : seulement un changement de forme.
- **Couche dense** : 1 600 entrées vers 10 sorties (une par classe), soit 1 600 × 10 + 10 = **16 010** paramètres. Chaque sortie est un score.
- **Softmax** : transforme les 10 scores en 10 probabilités de somme 1. La classe prédite est la plus probable.

Récapitulatif

Étape	Forme après	Ce qui se passe	Paramètres
Image	32×32×3	Les données	0
Conv1 + ReLU	28×28×32	32 filtres 5×5×3 détectent des motifs simples	2 432
Pool1	14×14×32	On garde le maximum de chaque bloc 2×2	0
Conv2 + ReLU	10×10×64	64 filtres 5×5×32 combinent les motifs	51 264
Pool2	5×5×64	Idem	0
Flatten	1 600	Mise à plat	0
Dense + softmax	10	Scores puis probabilités	16 010
Total			69 706

10. Ce que le réseau perd, ce qu'il gagne

Une intuition légitime : avec les transformations successives et la réduction de dimension, on devrait **perdre** de l'information, et non détecter des motifs plus riches. Les deux affirmations sont vraies, parce qu'elles ne parlent pas de la même chose.

10.1 Oui, de l'information est perdue

Aucune transformation ne peut **ajouter** d'information sur l'entrée : au mieux on la conserve, le plus souvent on en perd (c'est l'inégalité du traitement des données). Dans ce réseau, les pertes viennent surtout :

- **du pooling** : sur chaque bloc 2×2, on garde 1 valeur sur 4. Les blocs `[[6,0],[0,0]]` et `[[0,0],[0,6]]` donnent tous deux 6 : on ne sait plus où se trouvait le maximum ;
- **de la ReLU** : toutes les valeurs négatives deviennent 0 ;
- **de la couche finale** : 1 600 nombres sont résumés en 10 scores.

L'aplatissement ne perd rien, et une convolution seule en perd peu.

10.2 Mais le nombre de valeurs ne diminue pas partout

Étape	Forme	Nombre de valeurs
Image	32×32×3	3 072
Conv1	28×28×32	25 088
Pool1	14×14×32	6 272
Conv2	10×10×64	6 400
Pool2	5×5×64	1 600
Sortie	10	10

Après Conv1, il y a **plus** de valeurs qu'au départ : la convolution à 32 filtres décrit la même image de 32 façons différentes. Le réseau ne « compresse » donc pas simplement l'image : il la **reformule**, puis il filtre.

10.3 Ce que veut dire « plus riche »

Chaque nombre de la première couche répond à une question simple : « y a-t-il un bord vertical ici ? ». Chaque nombre d'une couche plus profonde répond à une question plus élaborée : « y a-t-il un coin, une roue, un œil ? ». Cela se construit en combinant les réponses précédentes et en regardant une zone plus large de l'image (14×14 pixels pour Conv2, 16×16 après Pool2).

> **Une analogie** : le résumé d'un livre contient bien moins de mots que le livre, donc il a perdu énormément d'information. Mais ses phrases portent un sens plus abstrait (« un homme poursuit sa vengeance »), plus utile pour savoir de quoi parle le livre que les mots pris un à un. Passer des pixels aux concepts, c'est la même chose : « plus riche » signifie **plus abstrait et plus utile pour la tâche**, pas « plus d'information sur l'entrée ».

10.4 Pourquoi perdre est même souhaitable

Ce que le réseau cherche à garder n'est pas l'information sur l'**image**, mais l'information sur l'**étiquette** (« chat » ou « chien »). Beaucoup de détails sont inutiles, voire nuisibles : la position exacte de l'objet, l'éclairage, le bruit, l'arrière-plan. Deux photos de chats peuvent être très différentes pixel par pixel et pourtant devoir être classées pareil. Perdre ces détails rend leurs représentations proches, ce qui permet de **généraliser** à de nouvelles images.

Dans l'espace des pixels, les classes sont emmêlées. Après plusieurs couches, la représentation est organisée de façon que la couche finale n'ait besoin que d'un séparateur très simple : le réseau **trie** l'information, jette ce qui ne sert pas et met en avant ce qui distingue les classes.

10.5 Quand la perte devient un problème

Une réduction trop agressive peut effacer ce dont on a besoin : un **petit objet** peut disparaître après quelques pooling, et pour une tâche qui exige la **position précise** (segmentation, détection) perdre la localisation est gênant. Les architectures modernes gèrent ce compromis : elles réduisent la taille plus progressivement, ou ajoutent des **connexions de contournement** qui réinjectent l'information fine (connexions résiduelles de ResNet, liens entre couches symétriques de U-Net).

Opération	Ce qu'on perd	Ce qu'on gagne
Convolution	Peu (le détail brut des pixels)	Des cartes de motifs, à plusieurs filtres
ReLU	Les valeurs négatives	La non-linéarité
Pooling	La position exacte	L'invariance aux petits décalages, moins de calcul
Couches profondes	Les détails sans lien avec la classe	Des caractéristiques abstraites, une zone vue plus large
Couche finale	Presque tout, sauf les 10 scores	La décision

11. Les trois idées clés, reliées aux étapes

- **Connectivité locale** : un neurone de Conv1 ne regarde que 5×5×3 = 75 valeurs, et non les 3 072 pixels.
- **Partage de paramètres** : le même filtre est utilisé à toutes les positions (étape 3), d'où 2 432 paramètres au lieu de 77 millions.

- **Équivariance à la translation** : si le bord de l'exemple de l'étape 2 se décale de deux colonnes, le pic de 30 se décale de deux colonnes aussi : la détection suit l'objet. C'est une **équivariance**, et non une invariance ; une invariance partielle apparaît ensuite grâce au pooling.

12. Comment le réseau apprend

C'est l'entraînement habituel : passe avant, perte, rétropropagation, mise à jour. Les filtres démarrent avec des valeurs aléatoires et leurs poids sont corrigés pour que les motifs détectés aident à classer. Comme un même filtre est utilisé à 784 positions, son gradient est la **somme des contributions de toutes ces positions**. La rétropropagation traverse le pooling (le gradient revient à l'endroit du maximum) et la ReLU (il est nul là où la sortie valait 0).

13. Calcul contre paramètres

Conv1 n'a que 2 432 paramètres, mais elle effectue 28 × 28 × 32 × 75 = **1,88 million** de multiplications-additions, car chaque filtre est appliqué à 784 positions. Conv2 en effectue **5,12 millions**. Les convolutions ont peu de paramètres mais coûtent du calcul : c'est ce qui rend l'entraînement des CNN lent sur un CPU.

14. Pour le voir par soi-même

Ce code (PyTorch) affiche la forme après chaque couche :

```
import torch, torch.nn as nn
modele = nn.Sequential(nn.Conv2d(3, 32, 5), nn.ReLU(), nn.MaxPool2d(2),
                       nn.Conv2d(32, 64, 5), nn.ReLU(), nn.MaxPool2d(2),
                       nn.Flatten(), nn.Linear(64 * 5 * 5, 10))
x = torch.zeros(1, 3, 32, 32)      # une image : (lot, canaux, hauteur, largeur)
for couche in modele:
    x = couche(x)
    print(type(couche).__name__, tuple(x.shape))
```

Il affiche (1, 32, 28, 28), (1, 32, 28, 28), (1, 32, 14, 14), (1, 64, 10, 10), (1, 64, 10, 10), (1, 64, 5, 5), (1, 1600) puis (1, 10). PyTorch place les canaux avant la hauteur et la largeur ; Keras les place en dernier.

À retenir

1. Une convolution fait glisser un petit filtre sur l'image ; chaque case de la carte résume une **fenêtre**, d'où une carte plus petite que l'image.

2. Un filtre couvre **tous les canaux** de son entrée ; le **nombre de filtres** est un choix de conception qui fixe la profondeur de la sortie.

3. On augmente le nombre de filtres en profondeur (32, puis 64) parce que les motifs se combinent et que les cartes rétrécissent.

4. Le réseau **perd** de l'information sur l'image, mais en garde une version **plus abstraite et plus utile pour la tâche**.

5. Peu de paramètres (partage des poids), mais beaucoup de calcul.