Cours Machine Learning

Chapitre 07

Deep Learning : Réseaux de Neurones Convolutifs (CNN)

Objectifs d'apprentissage :

— Comprendre l'opération de convolution et son utilité pour les images
— Maîtriser les couches fondamentales : convolution, pooling, fully-connected
— Étudier les architectures classiques : LeNet, AlexNet, VGG, ResNet
— Appliquer le transfer learning et le fine-tuning
— Implémenter un CNN avec PyTorch/TensorFlow

Prérequis : Chapitre 06 (Réseaux de Neurones Fondamentaux)

Durée estimée : 8-10 heures

Notebooks : `07_demo_*.ipynb`

Cours ML - Sandbox-ML
Version 1.0 - 2026

Table des matières

1 Introduction aux CNN

1.1 Motivation : Limitations des MLP pour les images

Un MLP classique présente des problèmes majeurs pour traiter les images :

1. **Nombre de paramètres explosif** : Image 224×224 RGB → 150K inputs → MLP(512) nécessite 77M paramètres pour la première couche !
2. **Pas d'invariance spatiale** : Un chat en haut à gauche vs en bas à droite sont des patterns complètement différents
3. **Perte de structure 2D** : Aplatir l'image en vecteur 1D détruit les relations spatiales locales

Exemple

MNIST avec MLP vs CNN
— MLP (784 → 128 → 10) : 100K paramètres, 98% accuracy
— CNN simple (2 conv + pooling) : 10K paramètres, 99% accuracy
Le CNN est 10× plus compact et plus performant !

1.2 Principes des CNN

Les CNN exploitent trois idées clés :

1. **Connexions locales (local connectivity)** : Chaque neurone ne "regarde" qu'une petite région de l'image (champ récepteur)
2. **Partage de poids (weight sharing)** : Le même filtre est appliqué sur toute l'image → invariance par translation
3. **Hiérarchie de features** : Couches successives détectent des features de plus en plus complexes

1.3 Hiérarchie de représentations

— **Couche 1** : Détecte bords, contours, gradients (features bas niveau)
— **Couche 2-3** : Détecte textures, motifs simples (coins, cercles)
— **Couche 4-5** : Détecte parties d'objets (yeux, roues, fenêtres)
— **Couche finale** : Détecte objets complets (chat, voiture, visage)

2 Opération de Convolution

2.1 Convolution 2D

Définition

Convolution Discrète 2D Pour une image $\mathbf{I} \in \mathbb{R}^{H \times W}$ et un filtre (kernel) $\mathbf{K} \in \mathbb{R}^{k \times k}$, la

convolution produit une feature map **O** :

$$\mathbf{O}[i,j] = (\mathbf{I} * \mathbf{K})[i,j] = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \mathbf{I}[i+m, j+n] \cdot \mathbf{K}[m,n] \tag{1}$$

Interprétation : Le filtre "glisse" sur l'image et calcule un produit scalaire local à chaque position.

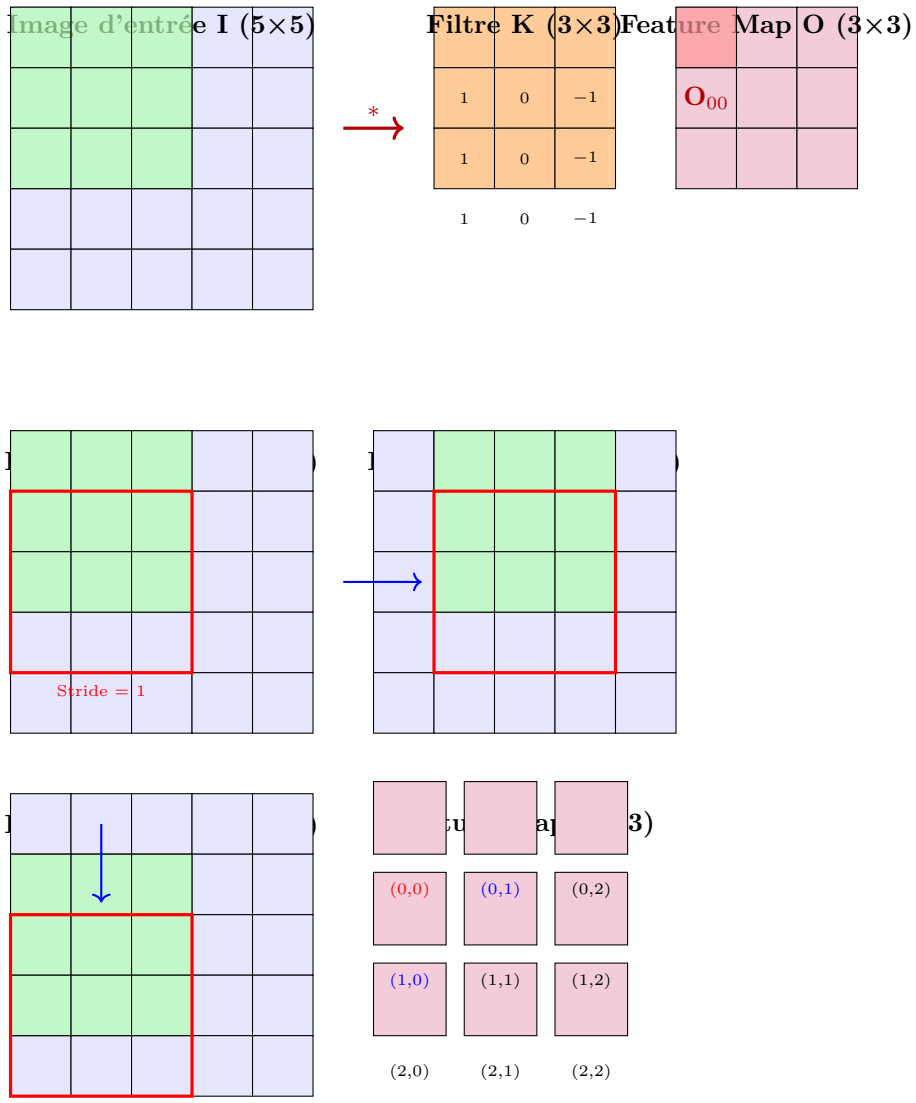

FIGURE 1 – Opération de convolution 2D avec sliding window (fenêtre glissante). **Haut :** Vue d'ensemble : le filtre 3×3 glisse sur l'image 5×5 pour produire une feature map 3×3. À chaque position, on calcule le produit scalaire entre le filtre et la zone correspondante de l'image (en vert). **Bas :** Détail du sliding window avec stride=1 : le filtre se déplace d'un pixel à la fois (horizontalement puis verticalement), calculant une valeur de sortie pour chaque position. Avec une image 5×5, un filtre 3×3, et sans padding, on obtient une sortie 3×3.

2.2 Exemple concret

Image 5×5 et filtre 3×3 :

$$\mathbf{I} = \begin{bmatrix} 1 & 2 & 3 & 0 & 1 \\ 0 & 1 & 2 & 3 & 0 \\ 1 & 0 & 1 & 2 & 3 \\ 2 & 1 & 0 & 1 & 2 \\ 3 & 2 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \text{ (détecteur de bord vertical)}$$

Calcul de $\mathbf{O}[0,0]$:

$$\begin{aligned} \mathbf{O}[0,0] &= 1 \cdot 1 + 2 \cdot 0 + 3 \cdot (-1) \\ &\quad + 0 \cdot 1 + 1 \cdot 0 + 2 \cdot (-1) \\ &\quad + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot (-1) \\ &= 1 + 0 - 3 + 0 + 0 - 2 + 1 + 0 - 1 = -4 \end{aligned}$$

2.3 Filtres classiques

Détecteur de bord vertical :

$$\mathbf{K}_{\text{vert}} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Détecteur de bord horizontal :

$$\mathbf{K}_{\text{horiz}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Filtre de flou (blur) :

$$\mathbf{K}_{\text{blur}} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Sobel (détection de contours) :

$$\mathbf{K}_{\text{Sobel}_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \mathbf{K}_{\text{Sobel}_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

2.4 Hyperparamètres de la convolution

2.4.1 Padding

Définition

Padding Ajout de zéros (ou autres valeurs) autour de l'image pour contrôler la taille de sortie.

- **Valid padding** (no padding) : $p = 0$
- **Same padding** : $p = \lfloor k/2 \rfloor$ (sortie même taille que l'entrée)

2.4.2 Stride

Définition

Stride Pas de déplacement du filtre. Stride $s = 1$: déplacement de 1 pixel. Stride $s = 2$: déplacement de 2 pixels (sous-échantillonnage).

2.4.3 Taille de sortie

Pour une entrée $H \times W$, filtre $k \times k$, padding p, stride s :

$$H_{out} = \left\lfloor \frac{H + 2p - k}{s} \right\rfloor + 1, \quad W_{out} = \left\lfloor \frac{W + 2p - k}{s} \right\rfloor + 1 \tag{2}$$

Exemple

Calcul de taille Entrée 32×32, filtre 5×5, padding 2, stride 1 :

$$H_{out} = \frac{32 + 2 \cdot 2 - 5}{1} + 1 = \frac{31}{1} + 1 = 32$$

Sortie : 32×32 (same padding)

2.5 Convolution multi-canal

Pour une image RGB (3 canaux) :

- Entrée : $\mathbf{I} \in \mathbb{R}^{H \times W \times C_{in}}$ ($C_{in} = 3$ pour RGB)
- Filtre : $\mathbf{K} \in \mathbb{R}^{k \times k \times C_{in} \times C_{out}}$
- Sortie : $\mathbf{O} \in \mathbb{R}^{H' \times W' \times C_{out}}$

Pour chaque canal de sortie c, on applique un filtre 3D :

$$\mathbf{O}[i, j, c] = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \sum_{c'=0}^{C_{in}-1} \mathbf{I}[i + m, j + n, c'] \cdot \mathbf{K}[m, n, c', c] + b_c \tag{3}$$

(i) Astuce

Un CNN apprend automatiquement les meilleurs filtres pendant l'entraînement, contrairement aux filtres manuels (Sobel, etc.) !

3 Couches d'un CNN

3.1 Couche de Convolution (Conv Layer)

Définition

Couche de Convolution Une couche Conv applique C_{out} filtres apprenables sur l'entrée pour produire C_{out} feature maps.

Nombre de paramètres :

$$\text{Params} = (k \times k \times C_{in} + 1) \times C_{out} \tag{4}$$

Le +1 correspond au biais par filtre.

Exemple

Conv2D(3, 64, kernel=3) Entrée RGB (3 canaux), 64 filtres de taille 3×3 :

$$\text{Params} = (3 \times 3 \times 3 + 1) \times 64 = 28 \times 64 = 1{,}792$$

3.2 Fonction d'Activation

Après chaque convolution, on applique une activation non-linéaire (typiquement ReLU) :

$$\mathbf{A} = \text{ReLU}(\mathbf{O}) = \max(0, \mathbf{O}) \tag{5}$$

3.3 Couche de Pooling

Définition

Pooling Opération de sous-échantillonnage qui réduit la dimension spatiale des feature maps.

3.3.1 Max Pooling

Prend le maximum dans chaque région :

$$\mathbf{O}[i, j] = \max_{m,n \in \text{pool}} \mathbf{I}[i \cdot s + m, j \cdot s + n] \tag{6}$$

Max Pooling 2×2 avec stride 2 :

$$\begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \xrightarrow{\text{MaxPool 2×2}} \begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}$$

3.3.2 Average Pooling

Prend la moyenne :

$$\mathbf{O}[i,j] = \frac{1}{k^2} \sum_{m,n \in \text{pool}} \mathbf{I}[i \cdot s + m, j \cdot s + n] \tag{7}$$

3.3.3 Avantages du Pooling

- ✓ Réduit la taille spatiale → moins de paramètres dans les couches suivantes
- ✓ Invariance locale par translation (petits déplacements)
- ✓ Augmente le champ récepteur
- ✓ Régularisation (réduit overfitting)

/!\ Attention

Le pooling **n'a pas de paramètres apprenables**. C'est une opération déterministe.

3.4 Couche Fully-Connected (FC)

En fin de réseau, on "aplatit" les feature maps et on applique un MLP classique :

$$\mathbf{x}_{\text{flat}} = \text{Flatten}(\mathbf{A}^{[L-1]}) \quad \in \mathbb{R}^d \tag{8}$$

$$\mathbf{z}^{[L]} = \mathbf{W}^{[L]}\mathbf{x}_{\text{flat}} + \mathbf{b}^{[L]} \tag{9}$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{[L]}) \tag{10}$$

3.5 Architecture typique

Un CNN classique suit le pattern :

$$\boxed{\text{INPUT}} \rightarrow \boxed{[\text{CONV} + \text{ReLU} + \text{POOL}] \times N} \rightarrow \boxed{\text{FC}} \rightarrow \boxed{\text{SOFTMAX}}$$

Exemple

CNN simple pour MNIST

Input : $28 \times 28 \times 1$

Conv1 : 32 filtres $3 \times 3 \rightarrow 28 \times 28 \times 32$

ReLU + MaxPool $2 \times 2 \rightarrow 14 \times 14 \times 32$

Conv2 : 64 filtres $3 \times 3 \rightarrow 14 \times 14 \times 64$

ReLU + MaxPool $2 \times 2 \rightarrow 7 \times 7 \times 64$

Flatten : $7 \times 7 \times 64 = 3{,}136$

FC : $3{,}136 \rightarrow 10$ (classes)

4 Backpropagation dans les CNN

4.1 Gradient de la convolution

Pour une convolution $\mathbf{O} = \mathbf{I} * \mathbf{K}$:

Gradient par rapport à l'entrée :

$$\frac{\partial L}{\partial \mathbf{I}} = \frac{\partial L}{\partial \mathbf{O}} * \mathbf{K}_{\text{rot180}} \tag{11}$$

où $\mathbf{K}_{\text{rot180}}$ est le filtre $\mathbf{K}$ tourné de 180°.

Gradient par rapport au filtre :

$$\frac{\partial L}{\partial \mathbf{K}} = \mathbf{I} * \frac{\partial L}{\partial \mathbf{O}} \tag{12}$$

4.2 Gradient du Max Pooling

Le gradient ne se propage qu'à travers l'élément qui était le maximum :

$$\frac{\partial L}{\partial \mathbf{I}[i,j]} = \begin{cases} \frac{\partial L}{\partial \mathbf{O}[i',j']} & \text{si } \mathbf{I}[i,j] = \max \text{ dans sa région} \\ 0 & \text{sinon} \end{cases} \tag{13}$$

(i) Astuce

En pratique, PyTorch et TensorFlow calculent automatiquement tous ces gradients grâce à l'autodifférentiation !

5 Architectures CNN Classiques

5.1 LeNet-5 (1998)

Auteurs : Yann LeCun et al. **Application :** Reconnaissance de chiffres manuscrits (MNIST)

Architecture :

Input : $32 \times 32 \times 1$
C1 : Conv $6@5 \times 5 \to 28 \times 28 \times 6$
S2 : AvgPool $2 \times 2 \to 14 \times 14 \times 6$
C3 : Conv $16@5 \times 5 \to 10 \times 10 \times 16$
S4 : AvgPool $2 \times 2 \to 5 \times 5 \times 16$
C5 : Conv $120@5 \times 5 \to 1 \times 1 \times 120$
F6 : FC $120 \to 84$
Output : FC $84 \to 10$

Paramètres : 60K **Fonction d'activation :** Tanh (à l'époque, ReLU n'était pas encore populaire)

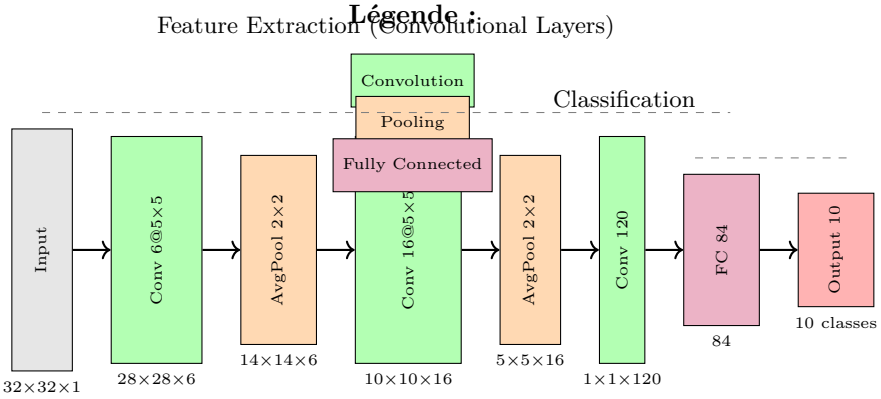

FIGURE 2 – Architecture LeNet-5 (LeCun et al., 1998). **Structure :** 2 blocs [Conv + AvgPool] pour l'extraction de features, suivis de 3 couches fully-connected pour la classification. LeNet-5 a été le premier CNN à réussir sur MNIST (reconnaissance de chiffres manuscrits) avec une précision >99%. **Innovations :** Partage de poids (convolution), pooling spatial, architecture hiérarchique. **Note :** À l'époque, la fonction d'activation était tanh ; aujourd'hui on utiliserait ReLU.

5.2 AlexNet (2012)

Auteurs : Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton **Dataset :** ImageNet (1.2M images, 1000 classes) **Impact :** Révolution du deep learning (top-5 error : 15.3% vs 26% avant)

Architecture :

$$\begin{aligned}
&\text{Input} : 227 \times 227 \times 3 \\
&\text{Conv1} : 96@11 \times 11, \text{stride } 4 \rightarrow 55 \times 55 \times 96 \\
&\text{MaxPool } 3 \times 3, \text{stride } 2 \rightarrow 27 \times 27 \times 96 \\
&\text{Conv2} : 256@5 \times 5 \rightarrow 27 \times 27 \times 256 \\
&\text{MaxPool } 3 \times 3, \text{stride } 2 \rightarrow 13 \times 13 \times 256 \\
&\text{Conv3} : 384@3 \times 3 \rightarrow 13 \times 13 \times 384 \\
&\text{Conv4} : 384@3 \times 3 \rightarrow 13 \times 13 \times 384 \\
&\text{Conv5} : 256@3 \times 3 \rightarrow 13 \times 13 \times 256 \\
&\text{MaxPool } 3 \times 3 \rightarrow 6 \times 6 \times 256 \\
&\text{FC6} : 9{,}216 \rightarrow 4{,}096 \\
&\text{FC7} : 4{,}096 \rightarrow 4{,}096 \\
&\text{FC8} : 4{,}096 \rightarrow 1{,}000
\end{aligned}$$

Paramètres : 60M **Innovations :**

- ReLU activation (au lieu de tanh)
- Dropout (0.5 dans les FC)
- Data augmentation (crop, flip, color jitter)
- GPU training (2× GTX 580)

5.3 VGGNet (2014)

Auteurs : Simonyan & Zisserman (Oxford) **Principe :** Utiliser des filtres 3×3 exclusivement, empiler beaucoup de couches

VGG-16 Architecture :

- **Block 1 :** 2× Conv(64, 3×3) + MaxPool
- **Block 2 :** 2× Conv(128, 3×3) + MaxPool
- **Block 3 :** 3× Conv(256, 3×3) + MaxPool
- **Block 4 :** 3× Conv(512, 3×3) + MaxPool
- **Block 5 :** 3× Conv(512, 3×3) + MaxPool
- **FC :** 4096 → 4096 → 1000

Paramètres : 138M (très lourd !) **Insight :** 2 convolutions 3×3 = champ récepteur 5×5, mais avec moins de paramètres et plus de non-linéarité

$$\text{Params}(5 \times 5) = 25C^2 \quad \text{vs} \quad \text{Params}(3 \times 3 \times 2) = 18C^2$$

5.4 ResNet (2015)

Auteurs : He et al. (Microsoft Research) **Innovation :** Residual connections (skip connections)

Problème des réseaux profonds : Degradation problem (réseaux très profonds difficiles à entraîner, même avec BN)

Définition

Residual Block Au lieu d'apprendre $\mathcal{H}(\mathbf{x})$, on apprend le résidu $\mathcal{F}(\mathbf{x}) = \mathcal{H}(\mathbf{x}) - \mathbf{x}$:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x} \tag{14}$$

La connexion $+\mathbf{x}$ est appelée "skip connection" ou "shortcut".

Architecture d'un bloc résiduel :

$$\begin{aligned} &\mathbf{x} \to \text{Conv } 3 \times 3 \to \text{BN} \to \text{ReLU} \\ &\quad \to \text{Conv } 3 \times 3 \to \text{BN} \to (+\mathbf{x}) \to \text{ReLU} \to \mathbf{y} \end{aligned}$$

Avantages :

- Permet d'entraîner des réseaux très profonds (50, 101, 152, voire 1000 couches)
- Gradient flow amélioré (évite vanishing gradient)
- Si nécessaire, le réseau peut "copier" l'identité en mettant $\mathcal{F}(\mathbf{x}) = 0$

ResNet-50 : 50 couches, 25M paramètres, top-5 error ImageNet : 3.6%

5.5 Autres architectures modernes

TABLE 1 – Comparaison des architectures CNN

Modèle	Année	Paramètres	Top-5 Error	Innovation
LeNet-5	1998	60K	-	Premier CNN
AlexNet	2012	60M	15.3%	ReLU, Dropout, GPU
VGG-16	2014	138M	7.3%	Filtres 3×3 profonds
ResNet-50	2015	25M	3.6%	Skip connections
Inception v3	2015	24M	3.5%	Multi-scale filters
EfficientNet	2019	5-66M	2.9%	Scaling optimal
Vision Transformer	2020	86M	2.3%	Self-attention

6 Implémentation avec PyTorch

6.1 CNN simple from scratch

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleCNN(nn.Module):
    def __init__(self):
```

```
        super(SimpleCNN, self).__init__()

        # Couches de convolution
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32,
                               kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,
                               kernel_size=3, padding=1)

        # Pooling
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Couches fully-connected
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

        # Dropout
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # Block 1: Conv + ReLU + Pool
        x = self.pool(F.relu(self.conv1(x)))  # 28x28x1 -> 14x14x32

        # Block 2: Conv + ReLU + Pool
        x = self.pool(F.relu(self.conv2(x)))  # 14x14x32 -> 7x7x64

        # Flatten
        x = x.view(-1, 64 * 7 * 7)  # (batch, 3136)

        # FC layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

# Instancier le mod le
model = SimpleCNN()
print(model)

# Compter les param tres
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params:,}")
```

Listing 1 – CNN simple pour MNIST

6.2 Entraînement

```
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

```

```python
# Dataset et DataLoader
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST('./data', train=True, download=True,
                               transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Loss et optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)

num_epochs = 10

for epoch in range(num_epochs):
    model.train()
    total_loss = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        # Forward
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)

        # Backward
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}")

print("Training complete!")
```

Listing 2 – Training loop

6.3 VGG-like architecture

```python
class VGGBlock(nn.Module):
    def __init__(self, in_channels, out_channels, num_convs):
        super(VGGBlock, self).__init__()
        layers = []
```

```
        for _ in range(num_convs):
            layers.append(nn.Conv2d(in_channels, out_channels,
                                    kernel_size=3, padding=1))
            layers.append(nn.ReLU(inplace=True))
            in_channels = out_channels
        layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
        self.block = nn.Sequential(*layers)

    def forward(self, x):
        return self.block(x)

class TinyVGG(nn.Module):
    def __init__(self, num_classes=10):
        super(TinyVGG, self).__init__()

        self.features = nn.Sequential(
            VGGBlock(3, 64, 2),   # 2x Conv64 + Pool
            VGGBlock(64, 128, 2), # 2x Conv128 + Pool
            VGGBlock(128, 256, 3) # 3x Conv256 + Pool
        )

        self.classifier = nn.Sequential(
            nn.Linear(256 * 4 * 4, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

Listing 3 – VGG-style CNN

6.4 ResNet block

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels,
                               kernel_size=3, stride=stride, padding=1,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection (shortcut)
```

```
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)  # Skip connection
        out = F.relu(out)
        return out

# Utilisation
class TinyResNet(nn.Module):
    def __init__(self, num_classes=10):
        super(TinyResNet, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=
    False)
        self.bn1 = nn.BatchNorm2d(64)

        self.layer1 = self._make_layer(64, 64, 2, stride=1)
        self.layer2 = self._make_layer(64, 128, 2, stride=2)
        self.layer3 = self._make_layer(128, 256, 2, stride=2)

        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(256, num_classes)

    def _make_layer(self, in_channels, out_channels, num_blocks, stride)
    :
        layers = []
        layers.append(ResidualBlock(in_channels, out_channels, stride))
        for _ in range(1, num_blocks):
            layers.append(ResidualBlock(out_channels, out_channels, 1))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.avg_pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

Listing 4 – Residual Block

7 Transfer Learning et Fine-Tuning

7.1 Principe

Idée : Utiliser un réseau pré-entraîné sur ImageNet (1.2M images) comme point de départ pour une nouvelle tâche.

Pourquoi ça marche ?

— Les features bas niveau (bords, textures) sont universelles
— Pré-entraîner sur un grand dataset capture ces features
— On peut réutiliser ces features pour une nouvelle tâche (même avec peu de données)

7.2 Stratégies

7.2.1 Feature Extraction (Frozen Backbone)

1. Charger un modèle pré-entraîné (ResNet, VGG, etc.)
2. **Geler** toutes les couches convolutionnelles
3. Remplacer la dernière couche FC par une nouvelle (taille = nb de classes)
4. Entraîner uniquement la nouvelle couche FC

Avantages : Très rapide, peu de données nécessaires **Inconvénients :** Features pas adaptées à la nouvelle tâche

7.2.2 Fine-Tuning

1. Charger modèle pré-entraîné
2. Remplacer dernière couche FC
3. **Entraîner tout le réseau** avec un learning rate très faible
4. Optionnel : dégeler progressivement les couches (shallow → deep)

Avantages : Meilleure performance **Inconvénients :** Nécessite plus de données, risque d'overfitting

7.3 Implémentation PyTorch

```
import torchvision.models as models

# Charger ResNet-18 pr -entra n
model = models.resnet18(pretrained=True)

# Option 1 : Feature Extraction (geler toutes les couches)
for param in model.parameters():
    param.requires_grad = False

# Remplacer la derni re couche FC
num_classes = 10  # Notre nouvelle t che
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, num_classes)

```

```python
# Seule la derni re couche sera entra n e
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)

# ----------------------------------------

# Option 2 : Fine-Tuning (tout entra ner)
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, num_classes)

# Entra ner tout le r seau avec un petit LR
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# ----------------------------------------

# Option 3 : Fine-Tuning progressif
model = models.resnet18(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, num_classes)

# Geler d'abord toutes les couches sauf FC
for param in model.parameters():
    param.requires_grad = False
for param in model.fc.parameters():
    param.requires_grad = True

# Entra ner FC pendant quelques epochs
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
# ... train ...

# Puis d geler tout et fine-tune avec petit LR
for param in model.parameters():
    param.requires_grad = True
optimizer = optim.Adam(model.parameters(), lr=1e-5)
# ... train ...
```

Listing 5 – Transfer Learning avec ResNet

7.4 Data Augmentation

Pour éviter l'overfitting avec peu de données :

```python
from torchvision import transforms

train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224),      # Crop al atoire
    transforms.RandomHorizontalFlip(),      # Flip horizontal
    transforms.ColorJitter(                 # Perturbations couleur
        brightness=0.2,
        contrast=0.2,
        saturation=0.2
    ),
    transforms.RandomRotation(15),          # Rotation   15
```

```
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

val_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

Listing 6 – Data Augmentation

8 Visualisation et Interprétation

8.1 Visualiser les filtres

```
import matplotlib.pyplot as plt

# Extraire les poids de la premi re couche conv
conv1_weights = model.conv1.weight.data.cpu()
# Shape: (out_channels, in_channels, k, k)

# Visualiser les 16 premiers filtres
fig, axes = plt.subplots(4, 4, figsize=(10, 10))
for i, ax in enumerate(axes.flat):
    if i < conv1_weights.shape[0]:
        # Si RGB, prendre le premier canal
        if conv1_weights.shape[1] == 3:
            filter_img = conv1_weights[i, 0, :, :]
        else:
            filter_img = conv1_weights[i, 0, :, :]
        ax.imshow(filter_img, cmap='gray')
        ax.axis('off')
plt.suptitle('Filtres Conv1')
plt.show()
```

Listing 7 – Visualiser les filtres Conv1

8.2 Visualiser les feature maps

```
def visualize_feature_maps(model, image, layer_name='conv1'):
    # Hook pour capturer l'output d'une couche
    activations = {}
    def hook_fn(module, input, output):
        activations['feature_maps'] = output

    # Enregistrer le hook
    layer = dict(model.named_modules())[layer_name]
```

```
    hook = layer.register_forward_hook(hook_fn)

    # Forward pass
    model.eval()
    with torch.no_grad():
        _ = model(image.unsqueeze(0))

    # Supprimer le hook
    hook.remove()

    # Visualiser
    feature_maps = activations['feature_maps'][0].cpu()
    num_maps = min(16, feature_maps.shape[0])

    fig, axes = plt.subplots(4, 4, figsize=(12, 12))
    for i, ax in enumerate(axes.flat):
        if i < num_maps:
            ax.imshow(feature_maps[i], cmap='viridis')
            ax.set_title(f'Map {i}')
            ax.axis('off')
    plt.suptitle(f'Feature Maps - {layer_name}')
    plt.show()
```

Listing 8 – Visualiser les activations

8.3 Grad-CAM (Class Activation Mapping)

Technique pour visualiser quelles régions de l'image influencent la prédiction.

```
def grad_cam(model, image, target_class):
    model.eval()
    image.requires_grad = True

    # Forward
    output = model(image.unsqueeze(0))
    class_score = output[0, target_class]

    # Backward
    model.zero_grad()
    class_score.backward()

    # R cup rer gradients de la derni re conv
    gradients = image.grad.data

    # Calculer importance (moyenne des gradients)
    weights = torch.mean(gradients, dim=(2, 3), keepdim=True)

    # Combiner avec feature maps
    cam = torch.sum(weights * image, dim=1, keepdim=True)
    cam = F.relu(cam)  # ReLU pour garder activations positives

```

```
    # Normaliser
    cam = (cam - cam.min()) / (cam.max() - cam.min())

    return cam.squeeze().cpu().numpy()
```

Listing 9 – Grad-CAM simplifié

9 Applications des CNN

9.1 Computer Vision

1. **Classification d'images**
   - ImageNet : 1000 classes (chiens, chats, avions, etc.)
   - Diagnostic médical : détection cancer, COVID-19 sur radiographies
2. **Détection d'objets** (Object Detection)
   - YOLO, Faster R-CNN, SSD
   - Applications : voitures autonomes, surveillance
3. **Segmentation sémantique**
   - U-Net, Mask R-CNN
   - Applications : imagerie médicale, édition photo
4. **Reconnaissance faciale**
   - FaceNet, DeepFace
   - Applications : déverrouillage téléphone, sécurité
5. **Génération d'images**
   - GANs (Generative Adversarial Networks)
   - Style Transfer, Super-Resolution

9.2 Au-delà de la vision

Les CNN peuvent aussi traiter d'autres données spatiales :

- **Traitement du signal audio** : spectrogrammes (convolution 1D ou 2D)
- **Séries temporelles** : Temporal CNN (TCN)
- **Texte** : CNN 1D pour classification de textes (moins utilisé que Transformers)
- **Graphes** : Graph Convolutional Networks (GCN)

10 Bonnes Pratiques

10.1 Architecture

- Utiliser des filtres 3×3 (standard moderne)
- Doubler les canaux quand on divise la résolution spatiale par 2
- Batch Normalization après chaque Conv (avant ReLU)
- Utiliser Global Average Pooling au lieu de FC massifs (réduit overfitting)
- Préférer des réseaux profonds mais avec skip connections (ResNet-style)

10.2 Entraînement

— **Optimizer :** Adam ou SGD avec momentum (0.9)
— **Learning Rate :** 1e-3 pour Adam, 1e-1 pour SGD
— **LR Scheduling :** ReduceLROnPlateau ou Cosine Annealing
— **Batch Size :** 32-128 (compromis vitesse/généralisation)
— **Régularisation :** L2 (1e-4), Dropout (0.5), Data Augmentation
— **Early Stopping :** Patience de 10-20 epochs

10.3 Transfer Learning

(i) Astuce

Règle générale :
— **Peu de données (< 1000)** : Feature extraction
— **Données moyennes (1K-10K)** : Fine-tuning des dernières couches
— **Beaucoup de données ($> 10K$)** : Fine-tuning complet ou entraînement from scratch

11 Avantages et Limites

11.1 Avantages

— ✓ Exploitation de la structure spatiale des images
— ✓ Invariance par translation (weight sharing)
— ✓ Beaucoup moins de paramètres qu'un MLP équivalent
— ✓ Hiérarchie automatique de features (bas niveau → haut niveau)
— ✓ State-of-the-art en vision par ordinateur
— ✓ Transfer learning très efficace

11.2 Limites

— × Nécessite beaucoup de données (ou transfer learning)
— × Pas invariant aux rotations/échelles (nécessite data augmentation)
— × Sensible aux adversarial examples (perturbations imperceptibles)
— × Coût computationnel élevé (GPU indispensable)
— × Difficile à interpréter (boîte noire)
— × Pas optimal pour données non-spatiales (tableaux, graphes)

12 Résumé du Chapitre

12.1 Points Clés

— **Convolution** : Produit scalaire local avec partage de poids → invariance translation
— **Pooling** : Sous-échantillonnage pour réduire dimension et augmenter champ récepteur
— **Architecture** : [Conv + ReLU + Pool] × N → Flatten → FC → Softmax
— **LeNet** (1998) : Premier CNN (MNIST)
— **AlexNet** (2012) : Révolution deep learning (ImageNet)

— **VGG** (2014) : Empilage profond de Conv 3×3
— **ResNet** (2015) : Skip connections → réseaux très profonds
— **Transfer Learning** : Réutilisation de modèles pré-entraînés
— **Data Augmentation** : Flip, crop, rotation, color jitter

12.2 Formules Essentielles

Formules à retenir

Convolution 2D :

$$\mathbf{O}[i,j] = \sum_{m=0}^{k-1}\sum_{n=0}^{k-1} \mathbf{I}[i+m, j+n] \cdot \mathbf{K}[m,n] + b$$

Taille de sortie :

$$H_{out} = \left\lfloor \frac{H + 2p - k}{s} \right\rfloor + 1$$

Nombre de paramètres Conv :

$$\text{Params} = (k \times k \times C_{in} + 1) \times C_{out}$$

Residual Block :

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}$$

13 Exercices

13.1 Questions de compréhension

1. Pourquoi un CNN a-t-il moins de paramètres qu'un MLP pour traiter des images ?
2. Expliquer l'intuition derrière le max pooling.
3. Quelle est la différence entre valid padding et same padding ?
4. Pourquoi VGG utilise exclusivement des filtres 3×3 ?
5. Comment les skip connections de ResNet aident-elles à entraîner des réseaux profonds ?
6. Quelle est la différence entre feature extraction et fine-tuning ?

13.2 Exercices pratiques

1. **CNN pour CIFAR-10**
   — Implémenter un CNN from scratch pour CIFAR-10
   — Architecture libre, objectif : $> 75\%$ accuracy
   — Utiliser data augmentation
2. **Transfer Learning**
   — Charger ResNet-18 pré-entraîné
   — Fine-tuner sur un petit dataset (ex : Cats vs Dogs)
   — Comparer feature extraction vs fine-tuning
3. **Visualisation**
   — Visualiser les filtres de Conv1
   — Visualiser les feature maps pour différentes couches

   — Implémenter Grad-CAM
4. **Architecture ResNet**
   — Implémenter un ResNet-18 from scratch
   — Comparer avec un CNN classique de même profondeur (sans skip connections)

Solutions disponibles dans `07_exercices.ipynb` *(solutions intégrées dans le notebook)*

14 Pour Aller Plus Loin

14.1 Lectures Recommandées

— LeCun et al. (1998) - "Gradient-based learning applied to document recognition" (LeNet)
— Krizhevsky et al. (2012) - "ImageNet Classification with Deep CNNs" (AlexNet)
— Simonyan & Zisserman (2014) - "Very Deep Convolutional Networks" (VGG)
— He et al. (2015) - "Deep Residual Learning for Image Recognition" (ResNet)
— Szegedy et al. (2015) - "Going Deeper with Convolutions" (Inception)

14.2 Ressources en Ligne

— CS231n Stanford : http://cs231n.stanford.edu/
— PyTorch Tutorials : https://pytorch.org/tutorials/
— Papers With Code : https://paperswithcode.com/
— Distill.pub : Articles interactifs sur les CNN

14.3 Architectures Avancées

— **EfficientNet** : Scaling optimal (width, depth, resolution)
— **MobileNet** : CNN légers pour mobile/edge devices
— **Vision Transformers (ViT)** : Alternatives aux CNN basées sur attention
— **YOLO, Faster R-CNN** : Détection d'objets
— **U-Net, Mask R-CNN** : Segmentation

14.4 Prochaines Étapes

Chapitre suivant recommandé : **Chapitre 08 - Deep Learning : RNN et Transformers**

Les RNN et Transformers sont des architectures pour les données séquentielles (texte, séries temporelles, audio).

Références

1. LeCun, Y., et al. (1998). "Gradient-based learning applied to document recognition". *Proceedings of the IEEE*, 86(11), 2278-2324.
2. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). "ImageNet classification with deep convolutional neural networks". *NIPS*.
3. Simonyan, K., & Zisserman, A. (2014). "Very deep convolutional networks for large-scale image recognition". *arXiv :1409.1556*.

4. He, K., et al. (2015). "Deep residual learning for image recognition". *CVPR*.
5. Szegedy, C., et al. (2015). "Going deeper with convolutions". *CVPR*.
6. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.