

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

Table des matières

1	Introduction aux CNN	2
1.1	Motivation : Limitations des MLP pour les images	2
1.2	Principes des CNN	2
1.3	Hiérarchie de représentations	2
2	Opération de Convolution	2
2.1	Convolution 2D	2
2.2	Exemple concret	3
2.3	Filtres classiques	3
2.4	Hyperparamètres de la convolution	4
2.4.1	Padding	4
2.4.2	Stride	4
2.4.3	Taille de sortie	4
2.5	Convolution multi-canal	4
3	Couches d'un CNN	5
3.1	Couche de Convolution (Conv Layer)	5
3.2	Fonction d'Activation	5
3.3	Couche de Pooling	5
3.3.1	Max Pooling	5
3.3.2	Average Pooling	6
3.3.3	Avantages du Pooling	6
3.4	Couche Fully-Connected (FC)	6
3.5	Architecture typique	6
4	Backpropagation dans les CNN	7
4.1	Gradient de la convolution	7
4.2	Gradient du Max Pooling	7
5	Architectures CNN Classiques	7
5.1	LeNet-5 (1998)	7
5.2	AlexNet (2012)	8
5.3	VGGNet (2014)	8
5.4	ResNet (2015)	9
5.5	Autres architectures modernes	9
6	Implémentation avec PyTorch	9
6.1	CNN simple from scratch	9
6.2	Entraînement	10
6.3	VGG-like architecture	11
6.4	ResNet block	12
7	Transfer Learning et Fine-Tuning	14
7.1	Principe	14
7.2	Stratégies	14

7.2.1	Feature Extraction (Frozen Backbone)	14
7.2.2	Fine-Tuning	14
7.3	Implémentation PyTorch	14
7.4	Data Augmentation	15
8	Visualisation et Interprétation	16
8.1	Visualiser les filtres	16
8.2	Visualiser les feature maps	16
8.3	Grad-CAM (Class Activation Mapping)	17
9	Applications des CNN	18
9.1	Computer Vision	18
9.2	Au-delà de la vision	18
10	Bonnes Pratiques	18
10.1	Architecture	18
10.2	Entraînement	19
10.3	Transfer Learning	19
11	Avantages et Limites	19
11.1	Avantages	19
11.2	Limites	19
12	Résumé du Chapitre	19
12.1	Points Clés	19
12.2	Formules Essentielles	20
13	Exercices	20
13.1	Questions de compréhension	20
13.2	Exercices pratiques	20
14	Pour Aller Plus Loin	21
14.1	Lectures Recommandées	21
14.2	Ressources en Ligne	21
14.3	Architectures Avancées	21
14.4	Prochaines Étapes	21

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 224x224 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 10x 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 \mathbf{O} :

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

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

2.2 Exemple concret

Image 5x5 et filtre 3x3 :

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

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 :

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

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 :

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

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 :

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

Le +1 correspond au biais par filtre.

Exemple : Conv2D(3)

Entrée RGB (3 canaux), 64 filtres de taille 3x3 :

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

3.2 Fonction d'Activation

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

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

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 :

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

Max Pooling 2x2 avec stride 2 :

$$\begin{bmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 7 & 8 \\ 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix} \xrightarrow{\text{MaxPool 2x2}} \begin{bmatrix} 6 & 8 \\ 3 & 4 \end{bmatrix}$$

3.3.2 Average Pooling

Prend la moyenne :

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

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 :

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

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

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

3.5 Architecture typique

Un CNN classique suit le pattern :



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 :

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

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

Gradient par rapport au filtre :

$$\frac{\partial L}{\partial \mathbf{K}} = \mathbf{I} * \frac{\partial L}{\partial \mathbf{O}} \quad (12)$$

4.2 Gradient du Max Pooling

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

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

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 × 32 × 1
C1 : Conv 6@5 × 5 → 28 × 28 × 6
S2 : AvgPool 2 × 2 → 14 × 14 × 6
C3 : Conv 16@5 × 5 → 10 × 10 × 16
S4 : AvgPool 2 × 2 → 5 × 5 × 16
C5 : Conv 120@5 × 5 → 1 × 1 × 120
F6 : FC 120 → 84
Output : FC 84 → 10

```

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

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 :

```

Input : 227 × 227 × 3
Conv1 : 96@11 × 11, stride 4 → 55 × 55 × 96
MaxPool 3 × 3, stride 2 → 27 × 27 × 96
Conv2 : 256@5 × 5 → 27 × 27 × 256
MaxPool 3 × 3, stride 2 → 13 × 13 × 256
Conv3 : 384@3 × 3 → 13 × 13 × 384
Conv4 : 384@3 × 3 → 13 × 13 × 384
Conv5 : 256@3 × 3 → 13 × 13 × 256
MaxPool 3 × 3 → 6 × 6 × 256
FC6 : 9,216 → 4,096
FC7 : 4,096 → 4,096
FC8 : 4,096 → 1,000

```

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 $\mathcal{F}(\mathbf{x}) = \mathcal{H}(\mathbf{x}) - \mathbf{x}$:

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

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

Architecture d'un bloc résiduel :

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

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 3x3 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

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class SimpleCNN(nn.Module):
6     def __init__(self):

```

```

7     super(SimpleCNN, self).__init__()
8
9     # Couches de convolution
10    self.conv1 = nn.Conv2d(in_channels=1, out_channels=32,
11                          kernel_size=3, padding=1)
12    self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,
13                          kernel_size=3, padding=1)
14
15    # Pooling
16    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
17
18    # Couches fully-connected
19    self.fc1 = nn.Linear(64 * 7 * 7, 128)
20    self.fc2 = nn.Linear(128, 10)
21
22    # Dropout
23    self.dropout = nn.Dropout(0.5)
24
25 def forward(self, x):
26     # Block 1: Conv + ReLU + Pool
27     x = self.pool(F.relu(self.conv1(x)))  # 28x28x1 -> 14x14x32
28
29     # Block 2: Conv + ReLU + Pool
30     x = self.pool(F.relu(self.conv2(x)))  # 14x14x32 -> 7x7x64
31
32     # Flatten
33     x = x.view(-1, 64 * 7 * 7)  # (batch, 3136)
34
35     # FC layers
36     x = F.relu(self.fc1(x))
37     x = self.dropout(x)
38     x = self.fc2(x)
39
40     return x
41
42 # Instancier le modèle
43 model = SimpleCNN()
44 print(model)
45
46 # Compter les paramètres
47 total_params = sum(p.numel() for p in model.parameters())
48 print(f"Total parameters: {total_params:,}")

```

Listing 1 – CNN simple pour MNIST

6.2 Entraînement

```

1 import torch.optim as optim
2 from torch.utils.data import DataLoader
3 from torchvision import datasets, transforms
4

```

```

5 # Dataset et DataLoader
6 transform = transforms.Compose([
7     transforms.ToTensor(),
8     transforms.Normalize((0.1307,), (0.3081,)))
9 ])
10
11 train_dataset = datasets.MNIST('./data', train=True, download=True,
12                                 transform=transform)
13 train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
14
15 # Loss et optimizer
16 criterion = nn.CrossEntropyLoss()
17 optimizer = optim.Adam(model.parameters(), lr=0.001)
18
19 # Training loop
20 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
21 model = model.to(device)
22
23 num_epochs = 10
24
25 for epoch in range(num_epochs):
26     model.train()
27     total_loss = 0
28
29     for batch_idx, (data, target) in enumerate(train_loader):
30         data, target = data.to(device), target.to(device)
31
32         # Forward
33         optimizer.zero_grad()
34         output = model(data)
35         loss = criterion(output, target)
36
37         # Backward
38         loss.backward()
39         optimizer.step()
40
41         total_loss += loss.item()
42
43     avg_loss = total_loss / len(train_loader)
44     print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}")
45
46 print("Training complete!")

```

Listing 2 – Training loop

6.3 VGG-like architecture

```

1 class VGGBlock(nn.Module):
2     def __init__(self, in_channels, out_channels, num_convs):
3         super(VGGBlock, self).__init__()
4         layers = []

```

```

5     for _ in range(num_convs):
6         layers.append(nn.Conv2d(in_channels, out_channels,
7                               kernel_size=3, padding=1))
8         layers.append(nn.ReLU(inplace=True))
9         in_channels = out_channels
10    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
11    self.block = nn.Sequential(*layers)
12
13    def forward(self, x):
14        return self.block(x)
15
16 class TinyVGG(nn.Module):
17     def __init__(self, num_classes=10):
18         super(TinyVGG, self).__init__()
19
20         self.features = nn.Sequential(
21             VGGBlock(3, 64, 2),      # 2x Conv64 + Pool
22             VGGBlock(64, 128, 2),   # 2x Conv128 + Pool
23             VGGBlock(128, 256, 3)  # 3x Conv256 + Pool
24         )
25
26         self.classifier = nn.Sequential(
27             nn.Linear(256 * 4 * 4, 512),
28             nn.ReLU(inplace=True),
29             nn.Dropout(0.5),
30             nn.Linear(512, num_classes)
31         )
32
33     def forward(self, x):
34         x = self.features(x)
35         x = x.view(x.size(0), -1)
36         x = self.classifier(x)
37         return x

```

Listing 3 – VGG-style CNN

6.4 ResNet block

```

1 class ResidualBlock(nn.Module):
2     def __init__(self, in_channels, out_channels, stride=1):
3         super(ResidualBlock, self).__init__()
4
5         self.conv1 = nn.Conv2d(in_channels, out_channels,
6                               kernel_size=3, stride=stride, padding=1,
7                               bias=False)
8         self.bn1 = nn.BatchNorm2d(out_channels)
9         self.conv2 = nn.Conv2d(out_channels, out_channels,
10                           kernel_size=3, padding=1, bias=False)
11         self.bn2 = nn.BatchNorm2d(out_channels)
12
13     # Skip connection (shortcut)

```

```

14     self.shortcut = nn.Sequential()
15     if stride != 1 or in_channels != out_channels:
16         self.shortcut = nn.Sequential(
17             nn.Conv2d(in_channels, out_channels,
18                     kernel_size=1, stride=stride, bias=False),
19             nn.BatchNorm2d(out_channels)
20         )
21
22     def forward(self, x):
23         out = F.relu(self.bn1(self.conv1(x)))
24         out = self.bn2(self.conv2(out))
25         out += self.shortcut(x)  # Skip connection
26         out = F.relu(out)
27         return out
28
29 # Utilisation
30 class TinyResNet(nn.Module):
31     def __init__(self, num_classes=10):
32         super(TinyResNet, self).__init__()
33
34         self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=False)
35         self.bn1 = nn.BatchNorm2d(64)
36
37         self.layer1 = self._make_layer(64, 64, 2, stride=1)
38         self.layer2 = self._make_layer(64, 128, 2, stride=2)
39         self.layer3 = self._make_layer(128, 256, 2, stride=2)
40
41         self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
42         self.fc = nn.Linear(256, num_classes)
43
44     def _make_layer(self, in_channels, out_channels, num_blocks, stride):
45         layers = []
46         layers.append(ResidualBlock(in_channels, out_channels, stride))
47         for _ in range(1, num_blocks):
48             layers.append(ResidualBlock(out_channels, out_channels, 1))
49         return nn.Sequential(*layers)
50
51     def forward(self, x):
52         x = F.relu(self.bn1(self.conv1(x)))
53         x = self.layer1(x)
54         x = self.layer2(x)
55         x = self.layer3(x)
56         x = self.avg_pool(x)
57         x = x.view(x.size(0), -1)
58         x = self.fc(x)
59         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

```

1 import torchvision.models as models
2
3 # Charger ResNet-18 pré-entraîné
4 model = models.resnet18(pretrained=True)
5
6 # Option 1 : Feature Extraction (geler toutes les couches)
7 for param in model.parameters():
8     param.requires_grad = False
9
10 # Remplacer la dernière couche FC
11 num_classes = 10 # Notre nouvelle tâche
12 num_features = model.fc.in_features
13 model.fc = nn.Linear(num_features, num_classes)
14

```

```

15 # Seule la dernière couche sera entraînée
16 optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
17
18 # -----
19
20 # Option 2 : Fine-Tuning (tout entraîner)
21 model = models.resnet18(pretrained=True)
22 model.fc = nn.Linear(model.fc.in_features, num_classes)
23
24 # Entrainer tout le réseau avec un petit LR
25 optimizer = optim.Adam(model.parameters(), lr=1e-4)
26
27 # -----
28
29 # Option 3 : Fine-Tuning progressif
30 model = models.resnet18(pretrained=True)
31 model.fc = nn.Linear(model.fc.in_features, num_classes)
32
33 # Geler d'abord toutes les couches sauf FC
34 for param in model.parameters():
35     param.requires_grad = False
36 for param in model.fc.parameters():
37     param.requires_grad = True
38
39 # Entrainer FC pendant quelques epochs
40 optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
41 # ... train ...
42
43 # Puis dégeler tout et fine-tune avec petit LR
44 for param in model.parameters():
45     param.requires_grad = True
46 optimizer = optim.Adam(model.parameters(), lr=1e-5)
47 # ... train ...

```

Listing 5 – Transfer Learning avec ResNet

7.4 Data Augmentation

Pour éviter l'overfitting avec peu de données :

```

1 from torchvision import transforms
2
3 train_transform = transforms.Compose([
4     transforms.RandomResizedCrop(224),           # Crop aléatoire
5     transforms.RandomHorizontalFlip(),            # Flip horizontal
6     transforms.ColorJitter(                      # Perturbations couleur
7         brightness=0.2,
8         contrast=0.2,
9         saturation=0.2
10    ),
11    transforms.RandomRotation(15),                # Rotation ±15°

```

```

12     transforms.ToTensor(),
13     transforms.Normalize(mean=[0.485, 0.456, 0.406],
14                           std=[0.229, 0.224, 0.225])
15   ])
16
17 val_transform = transforms.Compose([
18   transforms.Resize(256),
19   transforms.CenterCrop(224),
20   transforms.ToTensor(),
21   transforms.Normalize(mean=[0.485, 0.456, 0.406],
22                       std=[0.229, 0.224, 0.225])
23 ])

```

Listing 6 – Data Augmentation

8 Visualisation et Interprétation

8.1 Visualiser les filtres

```

1 import matplotlib.pyplot as plt
2
3 # Extraire les poids de la première couche conv
4 conv1_weights = model.conv1.weight.data.cpu()
5 # Shape: (out_channels, in_channels, k, k)
6
7 # Visualiser les 16 premiers filtres
8 fig, axes = plt.subplots(4, 4, figsize=(10, 10))
9 for i, ax in enumerate(axes.flat):
10    if i < conv1_weights.shape[0]:
11        # Si RGB, prendre le premier canal
12        if conv1_weights.shape[1] == 3:
13            filter_img = conv1_weights[i, 0, :, :]
14        else:
15            filter_img = conv1_weights[i, 0, :, :]
16        ax.imshow(filter_img, cmap='gray')
17        ax.axis('off')
18 plt.suptitle('Filtres Conv1')
19 plt.show()

```

Listing 7 – Visualiser les filtres Conv1

8.2 Visualiser les feature maps

```

1 def visualize_feature_maps(model, image, layer_name='conv1'):
2     # Hook pour capturer l'output d'une couche
3     activations = {}
4     def hook_fn(module, input, output):
5         activations['feature_maps'] = output
6
7     # Enregistrer le hook
8     layer = dict(model.named_modules())[layer_name]

```

```

9    hook = layer.register_forward_hook(hook_fn)
10
11    # Forward pass
12    model.eval()
13    with torch.no_grad():
14        _ = model(image.unsqueeze(0))
15
16    # Supprimer le hook
17    hook.remove()
18
19    # Visualiser
20    feature_maps = activations['feature_maps'][0].cpu()
21    num_maps = min(16, feature_maps.shape[0])
22
23    fig, axes = plt.subplots(4, 4, figsize=(12, 12))
24    for i, ax in enumerate(axes.flat):
25        if i < num_maps:
26            ax.imshow(feature_maps[i], cmap='viridis')
27            ax.set_title(f'Map {i}')
28            ax.axis('off')
29    plt.suptitle(f'Feature Maps - {layer_name}')
30    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.

```

1 def grad_cam(model, image, target_class):
2     model.eval()
3     image.requires_grad = True
4
5     # Forward
6     output = model(image.unsqueeze(0))
7     class_score = output[0, target_class]
8
9     # Backward
10    model.zero_grad()
11    class_score.backward()
12
13    # Récupérer gradients de la dernière conv
14    gradients = image.grad.data
15
16    # Calculer importance (moyenne des gradients)
17    weights = torch.mean(gradients, dim=(2, 3), keepdim=True)
18
19    # Combiner avec feature maps
20    cam = torch.sum(weights * image, dim=1, keepdim=True)
21    cam = F.relu(cam) # ReLU pour garder activations positives
22

```

```

23 # Normaliser
24 cam = (cam - cam.min()) / (cam.max() - cam.min())
25
26 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 :

- **Traitements 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 3x3 (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

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 3x3
- **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 3x3 ?
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.