

PyTorch & Réseaux de Neurones Convolutifs

Jour 3 — Après-midi

Julien Rolland

Formation M2 Développement Fullstack

Jour 3

Plan du module

- 1 Vers PyTorch
- 2 Tenseurs
- 3 torch.nn
- 4 Optimisation & Data
- 5 Convolutions
- 6 Pooling
- 7 Architecture CNN
- 8 Training Loop
- 9 Récapitulatif

Pourquoi passer à PyTorch ?

Limite du « fait main » (J3-AM)

- Notre classe `Value` est **pédagogique** mais lente
- Python pur : quelques milliers d'opérations/s
- Un vrai réseau : **milliards** d'opérations par forward

Même principe, autre échelle

Même graphe de calcul, même `backward()`

⇒ mais sur des **tenseurs** en C++/CUDA

La solution industrielle

- Calcul vectorisé** : matrices entières en une opération
- Backend C++/CUDA** : parallélisme GPU
- Écosystème riche** :
 - `torchvision` — Vision par ordinateur
 - `torchaudio` — Audio
 - `transformers` — NLP / LLM

Définition

Généralisation des matrices à n dimensions

scalaire ⊂ vecteur ⊂ matrice ⊂ tenseur

Propriétés clés

- `shape` : géométrie du tenseur
- `device` : CPU ou GPU (`cuda`)
- `grad` : stockage du gradient
⇒ notre `self.grad` de ce matin
- `requires_grad` : active l'autograd

Objet	Shape	Cas d'usage
Scalaire	[]	Perte \mathcal{L}
Vecteur	[n]	Biais, sortie
Matrice	[n, m]	Poids W
Batch	[B, n]	Mini-batch
Image	[B, C, H, W]	Batch d'images

Exemple : batch d'images RGB

[128, 3, 224, 224]

128 images · 3 canaux · 224 × 224 pixels

Le Module : brique de base

- On ne gère **plus** W et b manuellement
- nn.Linear, nn.Conv2d... gèrent leurs paramètres
- `__init__` : définit les couches
- `forward()` : flux de données

Avantage

`model.parameters()` retourne tous les poids \Rightarrow prêts pour l'optimiseur

```
1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 class MLP(nn.Module):
5     def __init__(self, n_in, n_h, n_out):
6         super().__init__()
7         self.fc1 = nn.Linear(n_in, n_h)
8         self.fc2 = nn.Linear(n_h, n_out)
9
10    def forward(self, x):
11        x = F.relu(self.fc1(x))
12        return self.fc2(x)
13
14 model = MLP(784, 128, 10)
15 # Nombre de paramètres :
16 n = sum(p.numel() for p in model.parameters())
17 print(f"{n} paramètres") # 101770
```

torch.optim : plus de $w \leftarrow w - lr * grad$

- **SGD** : descente de gradient classique
- **Adam** : adaptatif, converge rapidement
- **RMSprop** : stable pour les RNN

Dataset & DataLoader

- Dataset : comment accéder à un exemple
- DataLoader : gère le **batching**,
le **shuffling** et le **multi-threading**
- Le GPU ne doit pas attendre le CPU

```
1 import torch.optim as optim
2 from torch.utils.data import DataLoader
3
4 # Optimiseur Adam
5 opt = optim.Adam(model.parameters(), lr=1e
6 -3)
7
8 # DataLoader
9 loader = DataLoader(
10     dataset,
11     batch_size=32,
12     shuffle=True,
13     num_workers=4,      # threads CPU
14 )
15
16 for X, y in loader:
17     # X.shape = [32, ...]
18     pass
```

Pourquoi le MLP Échoue sur les Images ?

Problème 1 : Explosion des paramètres

- Image $1024 \times 1024 \Rightarrow 10^6$ entrées
- Couche cachée de 512 neurones
- $10^6 \times 512 = 5 \times 10^8$ poids
- Rien que pour la **première couche !**

Solution : CNN

- **Localité** : chaque neurone voit une petite zone
- **Partage des poids** : même filtre sur toute l'image
- **Hiérarchie** : bords → formes → objets

Problème 2 : Perte de topologie

- Flatten ⇒ perte de la **structure 2D**
- Un décalage d'1 pixel change tout le vecteur
- Le réseau ne voit pas la **proximité spatiale**

	MLP	CNN
Params (1 ^{re} couche)	500M	~ 1K
Invariance translation	Non	Oui
Localité spatiale	Non	Oui

Le Kernel (Filtre)

- Petite fenêtre (ex: 3×3) qui **glisse** sur l'image
- **Localité** : chaque neurone ne voit qu'une zone
- **Partage des poids** : même filtre partout
- \Rightarrow beaucoup moins de paramètres

champ récepteur

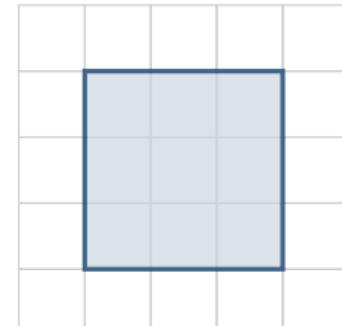
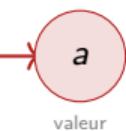


Image 5×5

Kernel 3×3

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9



Ce que le réseau apprend

Couche 1 : bords, contrastes

Couche 2 : coins, textures

Couche n : visages, objets...

Max Pooling 2×2

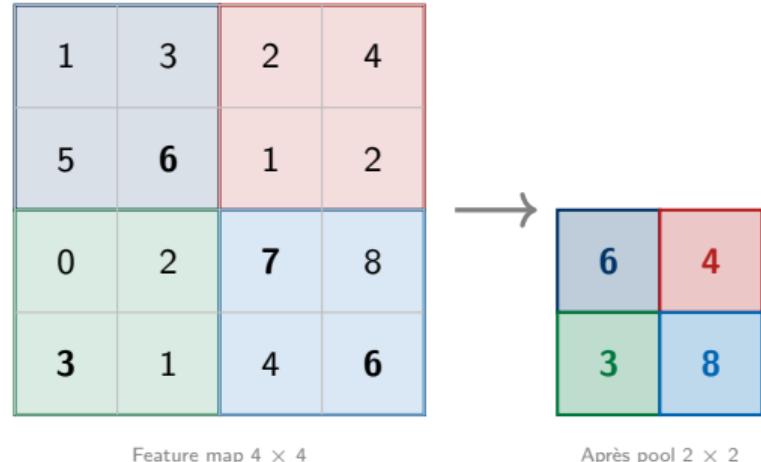
On garde la valeur **maximale** dans chaque zone 2×2

Objectifs

- ① **Réduire** la résolution spatiale (moins de calculs)
- ② **Invariance** aux petites translations
- ③ **Élargir** le champ de vision des couches suivantes

Résultat

Pool 2×2 : $H \times W \rightarrow \frac{H}{2} \times \frac{W}{2}$
 $\Rightarrow 4 \times$ moins de valeurs



Architecture Globale d'un CNN

Partie 1 : Extraction de features

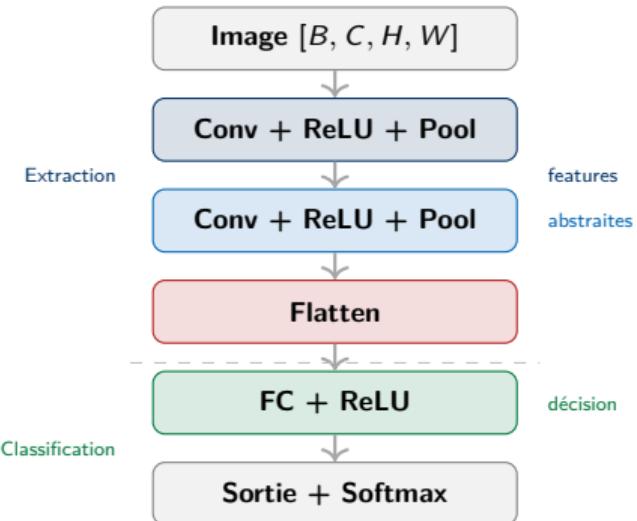
- **Conv + ReLU + Pool** $\times N$
- Image \rightarrow cartes de caractéristiques abstraites
- Taille spatiale \downarrow , profondeur (canaux) \uparrow

Partie 2 : Classification

- **Flatten** : cartes \rightarrow vecteur 1D
- MLP classique (J2) pour la décision finale
- Sortie : logits \rightarrow softmax \rightarrow classes

MNIST : $[B, 1, 28, 28] \rightarrow [B, 10]$

1 canal, 28×28 pixels, 10 classes



La Training Loop Standard

Les 5 étapes rituelles

- ➊ `zero_grad()` : efface les anciens gradients
- ➋ `model(x)` : forward pass
- ➌ `criterion(out, y)` : calcul de la perte
- ➍ `loss.backward()` : autograd
- ➎ `optimizer.step()` : mise à jour des poids

Piège classique

Oublier `zero_grad()` \Rightarrow gradients accumulés \Rightarrow divergence

```
1  criterion = nn.CrossEntropyLoss()
2  optimizer = optim.Adam(model.parameters())
3
4  for epoch in range(n_epochs):
5      for X, y in train_loader:
6          # 1. Zero gradients
7          optimizer.zero_grad()
8
9          # 2. Forward pass
10         output = model(X)
11
12         # 3. Compute loss
13         loss = criterion(output, y)
14
15         # 4. Backward (autograd)
16         loss.backward()
17
18         # 5. Update weights
19         optimizer.step()
```

Concepts maîtrisés

- **Autograd** : graphe de calcul, chain rule, backward
- **Tenseurs** : généralisation vectorisée de Value
- **torch.nn** : couches, paramètres, forward
- **Convolutions** : localité, partage des poids
- **Pooling** : réduction, invariance
- **Training loop** : les 5 étapes rituelles

Transition : MNIST → Vision complexe

- MNIST (28×28 , 10 classes) : CNN simple
- ImageNet (224×224 , 1000 classes) : ResNet, VGG
- Détection d'objets : YOLO, Faster R-CNN
- Segmentation : U-Net

Demain : Séquences & Attention

De la vision spatiale aux données **séquentielles**
⇒ RNN, LSTM, Transformers