PyTorch & Réseaux de Neurones Convolutifs

Jour 3 — Après-midi

Julien Rolland

Formation M2 Développement Fullstack

Jour 3

Plan du module

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()`
$\Rightarrow$ 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

Le Concept de Tenseur (`torch.Tensor`)

Définition

Généralisation des matrices à n dimensions

scalaire $\subset$ vecteur $\subset$ matrice $\subset$ tenseur

Propriétés clés

- `shape` : géométrie du tenseur
- `device` : CPU ou GPU (`cuda`)
- `grad` : stockage du gradient
  $\Rightarrow$ 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 ⇒ prêts pour l'optimiseur

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

class MLP(nn.Module):
    def __init__(self, n_in, n_h, n_out):
        super().__init__()
        self.fc1 = nn.Linear(n_in, n_h)
        self.fc2 = nn.Linear(n_h, n_out)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)

model = MLP(784, 128, 10)
# Nombre de parametres :
n = sum(p.numel() for p in model.parameters())
print(f"{n} parametres")  # 101770
```

`torch.optim` : plus de `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

```python
import torch.optim as optim
from torch.utils.data import DataLoader

# Optimiseur Adam
opt = optim.Adam(model.parameters(), lr=1e
    -3)

# DataLoader
loader = DataLoader(
    dataset,
    batch_size=32,
    shuffle=True,
    num_workers=4,    # threads CPU
)

for X, y in loader:
    # X.shape = [32, ...]
    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
- $10^6 \times 512 = \mathbf{5 \times 10^8}$ poids
- Rien que pour la **première couche** !

Problème 2 : Perte de topologie

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

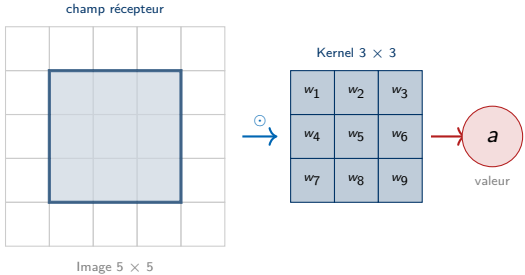
Solution : CNN

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

	MLP	**CNN**
Params (1re couche)	500M	$\sim$ 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



Ce que le réseau apprend

Couche 1 : bords, contrastes
Couche 2 : coins, textures
Couche n : visages, objets...

Max Pooling 2×2

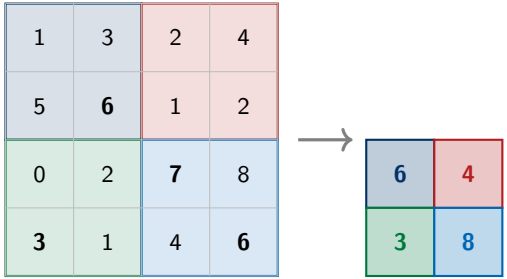
On garde la valeur **maximale** dans chaque zone 2×2

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



Feature map 4×4 — Après pool 2×2

Partie 1 : Extraction de features

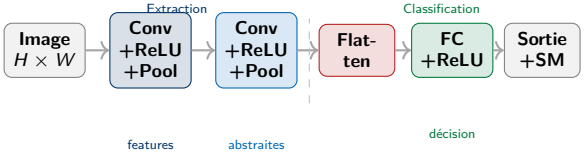
- **Conv + ReLU + Pool** $\times N$
- Image $\rightarrow$ cartes de caractéristiques abstraites
- Taille spatiale $\downarrow$, profondeur (canaux) $\uparrow$

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

1. `zero_grad()` : efface les anciens gradients
2. `model(x)` : forward pass
3. `criterion(out, y)` : calcul de la perte
4. `loss.backward()` : autograd
5. `optimizer.step()` : mise à jour des poids

Piège classique

Oublier `zero_grad()` ⇒ gradients **accumulés** ⇒ divergence

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

for epoch in range(n_epochs):
    for X, y in train_loader:
        # 1. Zero gradients
        optimizer.zero_grad()

        # 2. Forward pass
        output = model(X)

        # 3. Compute loss
        loss = criterion(output, y)

        # 4. Backward (autograd)
        loss.backward()

        # 5. Update weights
        optimizer.step()
```

Récapitulatif : De J3-AM à la Vision

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 $\rightarrow$ 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**
$\Rightarrow$ RNN, LSTM, Transformers