

TP4. Réseaux de neurones convolutionnels.

UE apprentissage statistique et réseaux de neurones.

Master mathématiques appliquées, statistique - parcours data science.

Frédéric Richard, AMU, 2025.

Antoine Legendre

L'objectif de ce TP est d'apprendre à mettre en oeuvre des réseaux de neurones convolutionnels pour effectuer des tâches de classification. Dans un premier temps, on s'intéresse à la classification des chiffres manuscrits de la base de données MNIST. Puis, on travaille sur un problème de détection de fissures sur des images de matériaux.

Déclaration des librairies et méthodes utiles au TP:

```
In [2]: # Importations nécessaires
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from torchvision.datasets import MNIST
from torch.optim import SGD, Adam
from torch import device as torch_device, manual_seed, no_grad
from torch.backends import cudnn
from torchsummary import summary
import os
import glob
from imageio.v3 import imread
from PIL import Image
import numpy as np
import zipfile
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

1. Classification des chiffres manuscrits.

1.1. Préparation des données.

On commence par charger la base de données MNIST et à la préparer pour le traitement. Les images de la base sont normalisées. On définit un générateur de batchs (mini-lots) pour l'apprentissage et le test, qui intègre des transformations sur les images.

```
In [3]: # Fonction de chargement des données MNIST
def load_mnist(batch_size=64):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
    train_dataset = MNIST(root='./data', train=True, download=True, transform=transform)
    test_dataset = MNIST(root='./data', train=False, download=True, transform=transform)

    return DataLoader(train_dataset, batch_size=batch_size, shuffle=True), \
           DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

train_loader, test_loader = load_mnist()
```

```
In [4]: # Affichage d'un exemple de batch
images, labels = next(iter(train_loader))
print(f"Taille du batch : {images.shape}")
print(f"Labels : {labels[:10]}")
```

```
Taille du batch : torch.Size([64, 1, 28, 28])
Labels : tensor([6, 7, 0, 9, 0, 8, 8, 4, 8, 4])
```

1.2. Régression multinomiale.

Tout d'abord, on effectue la classification des images mnist à partir d'une régression multinomiale.

Définition du réseau : On définit le réseau de neurones qui permet de faire de la régression multinomiale sur la base MNIST.

```
In [5]: # Modèle de régression multinomiale
class MultinomialRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(28 * 28, 10)

    def forward(self, x):
        x = torch.flatten(x, 1)
        return self.linear(x)
```

Apprentissage du réseau : On minimise un critère d'entropie croisée avec un algorithme de gradient stochastique par mini-lots.

On prépare l'apprentissage.

```
In [6]: # Initialisation des paramètres de PyTorch
```

```
manual_seed(12)
cudnn.deterministic = True
device = torch_device("cuda" if torch.cuda.is_available() else "cpu")
```

On définit le modèle, la fonction de perte et l'optimiseur.

```
In [7]: # Entraînement du modèle
def train_model(model, train_loader, nb_epochs=3, lr=0.01):
    optimizer = SGD(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(nb_epochs):
        running_loss = 0.0
        for X, Y in train_loader:
            optimizer.zero_grad()
            outputs = model(X)
            loss = criterion(outputs, Y)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"[epoch + 1] loss = {running_loss / len(train_loader):
```

```
In [8]: # Instanciation et entraînement du modèle de régression
model = MultinomialRegression().to(device)
train_model(model, train_loader)
```

```
[1] loss = 0.468
```

```
[2] loss = 0.333
```

```
[3] loss = 0.312
```

Evaluation. On évalue le modèle sur la base de test en calculant la précision du modèle (pourcentage d'images correctement classées).

On définit une fonction pour calculer une erreur de classification (*accuracy*).

```
In [9]: # Evaluation de la précision
def evaluate(model, data_loader):
    model.eval()
    correct, total = 0, 0
    with no_grad():
        for images, labels in data_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total * 100
```

```
In [10]: # Précision d'entraînement et test
train_acc = evaluate(model, train_loader)
test_acc = evaluate(model, test_loader)
print(f"Précision (train) : {train_acc:.2f} %")
print(f"Précision (test) : {test_acc:.2f} %")
```

```
Précision (train) : 91.50 %
```

```
Précision (test) : 91.68 %
```

```
In [11]: # Affichage de la complexité du modèle
summary(model, (1, 28, 28))
```

```
-----
              Layer (type)              Output Shape              Param #
=====
              Linear-1                  [-1, 10]                  7,850
=====
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
-----
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.03
Estimated Total Size (MB): 0.03
-----
```

1.3. Réseaux de neurones convolutionnels.

1. En vous inspirant du modèle de régression logistique, créer le réseau de neurones convolutionnel (CNN 1) suivant:

Couche	Spécifications	Dimension de la sortie	Nombre de paramètres
Input	-	(28, 28)	-
Convolution 2D	32 filtres de taille 3 x 3, activation ReLU
Max Pooling	Taille de fenêtre 2 x 2 et sous-échantillonnage de pas 2
Vectorisation	-
Dense	10 cellules

2. Faire l'apprentissage du CNN 1.
3. Comparer le CNN 1 au modèle de régression logistique en termes de complexité (nombre de paramètres) et de précision.
4. Mêmes questions avec le réseau de neurones convolutionnel (CNN 2):

Couche	Spécifications	Dimension de la sortie	Nombre de paramètres
Input	-	(28, 28)	-
Convolution 2D n°1	32 filtres de taille 3 x 3, activation ReLU
Max Pooling	Taille de fenêtre 2 x 2 et sous-échantillonnage de pas 2
Convolution 2D n°2	64 filtres de taille 3 x 3, activation ReLU
	Taille de fenêtre 2 x 2 et sous-		

Max Pooling	échantillonnage de pas 2
Vectorisation	-
Dense n°1	128 cellules
Dense n°2	10 cellules

```
In [12]: # Modèle CNN1
class CNN1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(32 * 14 * 14, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = torch.flatten(x, 1)
        return self.fc1(x)
```

```
In [13]: # Entraînement et évaluation pour CNN1
model = CNN1().to(device)
train_model(model, train_loader)

train_acc = evaluate(model, train_loader)
test_acc = evaluate(model, test_loader)
print(f"Précision (train) : {train_acc:.2f} %")
print(f"Précision (test) : {test_acc:.2f} %")
```

```
[1] loss = 0.328
[2] loss = 0.173
[3] loss = 0.130
Précision (train) : 96.95 %
Précision (test) : 96.97 %
```

```
In [14]: # Affichage de la complexité du modèle CNN1
summary(model, (1, 28, 28))
```

```
-----
Layer (type)                Output Shape          Param #
=====
      Conv2d-1              [-1, 32, 28, 28]         320
      MaxPool2d-2          [-1, 32, 14, 14]           0
      Linear-3              [-1, 10]              62,730
=====
Total params: 63,050
Trainable params: 63,050
Non-trainable params: 0
-----
Input size (MB): 0.00
Forward/backward pass size (MB): 0.24
Params size (MB): 0.24
Estimated Total Size (MB): 0.48
-----
```

Modèle de régression logistique :

- Entrée : $28 \times 28 = 784$ pixels
- Nombre total de paramètres : $784 \times 10 + 10 = 7850$ (car 10 classes de sortie)

Modèle CNN 1 :

- Entrée : $28 \times 28 = 784$ pixels
- Couche convolutionnelle : $1 \times 32 \times 3 \times 3 + 32 = 320$ (1 canal d'entrée et 32 filtres de sortie de taille 3×3)
- Couche fully connected : $32 \times 14 \times 14 \times 10 + 10 = 62730$ (32 canaux d'entrée d'une image réduite à 14×14 pixels et 10 classes de sortie)
- Nombre total de paramètres : $62730 + 320 = 63050$

Le modèle CNN 1 est donc beaucoup plus complexe, il a plus de 8 fois plus de paramètres que la régression logistique mais il est plus performant, 97.0% de précision contre 91.7% pour le modèle de régression logistique.

```
In [15]: # Modèle CNN2
class CNN2(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(7 * 7 * 64, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = torch.flatten(x, 1)
        return self.fc2(F.relu(self.fc1(x)))
```

```
In [16]: # Entraînement et évaluation pour CNN2
model = CNN2().to(device)
train_model(model, train_loader)

train_acc = evaluate(model, train_loader)
test_acc = evaluate(model, test_loader)
print(f"Précision (train) : {train_acc:.2f} %")
print(f"Précision (test) : {test_acc:.2f} %")
```

```
[1] loss = 0.505
[2] loss = 0.155
[3] loss = 0.101
Précision (train) : 97.36 %
Précision (test) : 97.48 %
```

```
In [17]: # Affichage de la complexité du modèle CNN2
summary(model, (1, 28, 28))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 28, 28]	320
MaxPool2d-2	[-1, 32, 14, 14]	0
Conv2d-3	[-1, 64, 14, 14]	18,496
MaxPool2d-4	[-1, 64, 7, 7]	0
Linear-5	[-1, 128]	401,536
Linear-6	[-1, 10]	1,290
Total params: 421,642		
Trainable params: 421,642		
Non-trainable params: 0		
Input size (MB): 0.00		
Forward/backward pass size (MB): 0.36		
Params size (MB): 1.61		
Estimated Total Size (MB): 1.97		

Modèle CNN 2 :

- Entrée : $28 \times 28 = 784$ pixels
- Couche convolutionnelle 1 : $1 \times 32 \times 3 \times 3 + 32 = 320$ (1 canal d'entrée et 32 filtres de sortie de taille 3×3)
- Couche convolutionnelle 2 : $32 \times 64 \times 3 \times 3 + 64 = 18,496$ (32 canaux d'entrée et 64 filtres de sortie de taille 3×3)
- Couche fully connected 1 : $64 \times 7 \times 7 \times 128 + 128 = 401,536$ (64 canaux d'entrée d'une image réduite à 7×7 pixels et 128 neurones de sortie)
- Couche fully connected 2 : $128 \times 10 + 10 = 1,290$ (128 neurones d'entrée et 10 classes de sortie)
- Nombre total de paramètres : $320 + 18496 + 401536 + 1290 = 421,642$

Le modèle CNN 2 est donc environ 7 fois plus complexe que le modèle CNN 1 et environ 54 fois plus complexe que le modèle de régression logistique. En revanche, il est plus précis que ces deux modèles : 97.5% de précision contre 97.0% pour le modèle CNN 1 et 91.7% pour le modèle de régression logistique.

1. Détection des fissures sur des images de matériaux.

1.1. Présentation des données.

Les données se trouvent sur le site [surface crack detection](#) de Kaggle. Elles sont constituées d'images de matériaux dont certains comportent des fissures et d'autres non. L'objectif est de construire un réseau de neurones qui permet de détecter ces fissures.

1.2. Préparation des données.

On commence par mettre à disposition les données. Plusieurs solutions sont possibles. Lorsqu'on travaille sur Kaggle, on peut mettre à disposition les données dans le répertoire de travail `/kaggle/working/` en utilisant les outils dédiés de kaggle.

```
In [18]: # Préparation des données d'images
with zipfile.ZipFile("archive.zip", "r") as zip_ref:
    zip_ref.extractall("archive")
```

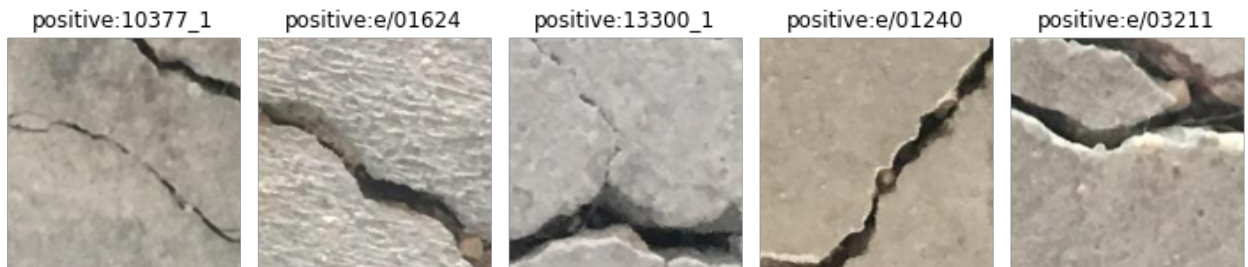
On fait ensuite la liste des accès physiques vers les images en distinguant celles qui ont des fissures (dans le répertoire Positive) de celles qui n'en ont pas (dans le répertoire Negative). On crée également un vecteur d'étiquettes pour associer la classe à chaque image.

```
In [19]: pos_images = glob.glob("archive/Positive/*.jpg")
neg_images = glob.glob("archive/Negative/*.jpg")
images = pos_images + neg_images
labels = np.array([1] * len(pos_images) + [0] * len(neg_images))
```

On peut visualiser quelques images pour se rendre compte des données.

```
In [20]: # Affichage d'images
def Affiche_Image(path, cnt, titre):
    img = imread(path)
    plt.subplot(2, 5, cnt)
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.title(titre)

plt.figure(figsize=(10, 10))
cnt = 1
for path in neg_images[:5]:
    Affiche_Image(path, cnt, "negative:" + path[-9:-4])
    cnt += 1
for path in pos_images[:5]:
    Affiche_Image(path, cnt, "positive:" + path[-11:-4])
    cnt += 1
plt.tight_layout()
plt.show()
```



Ensuite, on répartie les images en trois sous-ensembles des sous-ensembles qui seront utilisé pour l'entraînement, la validation et le test. Pour cela, on peut utiliser une méthode de scikit-learn qui permet de constituer des sous-ensembles équilibrés entre les deux classes (positif et négatif).

```
In [21]: # Séparation des données en train, validation et test
images_reste, images_test, y_reste, labels_test = train_test_split(images, y, test_size=0.1, random_state=42)
images_train, images_val, labels_train, labels_val = train_test_split(images_reste, y_reste, test_size=0.1, random_state=42)

print(f"Taille de la base d'entraînement: {len(images_train)}")
print(f"Taille de la base de validation: {len(images_val)}")
print(f"Taille de la base de test: {len(images_test)}")
```

```
Taille de la base d'entraînement: 24000
Taille de la base de validation: 8000
Taille de la base de test: 8000
```

Puis on personnalise une classe Dataset pour gérer les données en pytorch et de manipuler les mini-lots.

```
In [22]: # Définition du Dataset pour les fissures
class Dataset_fissures(Dataset):
    def __init__(self, img_path, img_labels):
        self.img_path = img_path
        self.img_labels = torch.Tensor(img_labels).view(-1, 1)
        self.transforms = transforms.Compose([transforms.Grayscale(), transforms.RandomCrop(100, 100)])

    def __getitem__(self, index):
        img = Image.open(self.img_path[index]).convert("L")
        return self.transforms(img), self.img_labels[index]

    def __len__(self):
        return len(self.img_path)
```

On spécifie ensuite les trois sous-ensembles (apprentissage, validation et test).

```
In [23]: train_dataset = Dataset_fissures(images_train, labels_train)
val_dataset = Dataset_fissures(images_val, labels_val)
test_dataset = Dataset_fissures(images_test, labels_test)
```

Pour finir, on crée les générateurs de patches sur les ensembles de données qui seront utilisés pour l'apprentissage, la validation et le test.

```
In [24]: # DataLoader
batch_size = 16
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Pour l'apprentissage, on utilisera comme fonction de perte, l'entropie croisée binaire, adaptée pour le cas où il y a deux classes. Celle-ci s'appelle [BCEWithLogitsLoss](#). Ce critère nécessite que la dernière couche du réseau ne comporte qu'une seule cellule. Comme optimiseur, on utilisera l'algorithme *Adam* en spécifiant un `learning_rate` à $1e-3$.

Exercice 2.

1. Construire un réseau de neurones convolutionnel pour classer les images, en indiquant sa complexité.
2. Faire l'apprentissage de ce réseau en l'évaluant sur des données de validation à chaque époque.
3. Evaluer la précision de ce réseau sur des données de test.
4. Tester les méthodes de dropout ou de batch normalization

```
In [25]: # Modèle de détection des fissures
class CNN_FissureDetection(nn.Module):
    def __init__(self):
        super(CNN_FissureDetection, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten_dim = (32 * (227 // 4) * (227 // 4))
        self.fc1 = nn.Linear(self.flatten_dim, 64)
        self.fc2 = nn.Linear(64, 1)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        return x
```

```
In [26]: # Création du modèle et affichage de sa complexité
```

```
model = CNN_FissureDetection().to(device)
summary(model, (1, 227, 227))
```

```
-----
Layer (type)                Output Shape              Param #
=====
Conv2d-1                    [-1, 16, 227, 227]        160
MaxPool2d-2                 [-1, 16, 113, 113]        0
Conv2d-3                    [-1, 32, 113, 113]       4,640
MaxPool2d-4                 [-1, 32, 56, 56]         0
Linear-5                    [-1, 64]                 6,422,592
Linear-6                    [-1, 1]                  65
=====
Total params: 6,427,457
Trainable params: 6,427,457
Non-trainable params: 0
-----
Input size (MB): 0.20
Forward/backward pass size (MB): 11.73
Params size (MB): 24.52
Estimated Total Size (MB): 36.45
-----
```

```
In [ ]: # Entraînement du modèle de détection des fissures
criterion = nn.BCELoss()
optimizer = Adam(model.parameters(), lr=1e-3)

for epoch in range(3):
    running_loss = 0.0
    for X, Y in train_loader:
        optimizer.zero_grad()
        outputs = model(X)
        loss = criterion(outputs, Y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"[epoch + 1] loss = {running_loss / len(train_loader):.3f}")

[1] loss = 0.130
```

```
In [ ]: # Evaluation de la précision
def evaluate_model(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for X, Y in test_loader:
            outputs = model(X)
            predictions = (torch.sigmoid(outputs) > 0.5).float()
            correct += (predictions == Y).sum().item()
            total += Y.size(0)

    return correct / total * 100
```

```
In [ ]: # Evaluation sur les données de test
accuracy = evaluate_model(model, test_loader, device)
print(f"Précision du modèle : {accuracy:.2f}%")
```

Complexité :

- Entrée : $227 \times 227 = 51,529$ pixels
- Couche convolutionnelle 1 : $1 \times 16 \times 3 \times 3 + 16 = 160$ (1 canal d'entrée et 16 filtres de sortie de taille 3×3)
- Couche convolutionnelle 2 : $16 \times 32 \times 3 \times 3 + 32 = 4,640$ (16 canaux d'entrée et 32 filtres de sortie de taille 3×3)
- Couche fully connected 1 : $32 \times 56 \times 56 \times 64 + 64 = 6,422,592$ (32 canaux d'entrée d'une image réduite à 56×56 pixels et 64 neurones de sortie)
- Couche fully connected 2 : $64 \times 1 + 1 = 65$ (64 neurones d'entrée et 1 classe de sortie)
- Nombre total de paramètres : $160 + 4,640 + 6,422,592 + 65 = 6,427,457$

Le modèle est donc assez complexe mais également assez précis (% de précision sur les données de test).

```
In [ ]: # Modèle de détection des fissures avec batch normalisation
class CNN_FissureDetectionBN(nn.Module):
    def __init__(self):
        super(CNN_FissureDetectionBN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.flatten_dim = (32 * (227 // 4) * (227 // 4))
        self.fc1 = nn.Linear(self.flatten_dim, 64)
        self.fc2 = nn.Linear(64, 1)

    def forward(self, x):
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool2(F.relu(self.bn2(self.conv2(x))))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        return x
```

```
In [ ]: # Création du modèle et affichage de sa complexité
model = CNN_FissureDetectionBN().to(device)
summary(model, (1, 227, 227))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 16, 227, 227]	160
BatchNorm2d-2	[-1, 16, 227, 227]	32
MaxPool2d-3	[-1, 16, 113, 113]	0
Conv2d-4	[-1, 32, 113, 113]	4,640
BatchNorm2d-5	[-1, 32, 113, 113]	64
MaxPool2d-6	[-1, 32, 56, 56]	0
Linear-7	[-1, 64]	6,422,592
Linear-8	[-1, 1]	65

Total params: 6,427,553
 Trainable params: 6,427,553
 Non-trainable params: 0

Input size (MB): 0.20
 Forward/backward pass size (MB): 21.14
 Params size (MB): 24.52
 Estimated Total Size (MB): 45.86

```
In [ ]: # Entraînement du modèle de détection des fissures avec batch normalisation
criterion = nn.BCELoss()
optimizer = Adam(model.parameters(), lr=1e-3)

for epoch in range(3):
    running_loss = 0.0
    for X, Y in train_loader:
        optimizer.zero_grad()
        outputs = model(X)
        loss = criterion(outputs, Y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"[epoch + 1] loss = {running_loss / len(train_loader):.3f}")

[1] loss = 0.934
```

```
In [ ]: # Evaluation sur les données de test
accuracy = evaluate_model(model, test_loader, device)
print(f"Précision du modèle : {accuracy:.2f}%")
```

Complexité :

- Entrée : $227 \times 227 = 51,529$ pixels
- Couche convolutionnelle 1 : $1 \times 16 \times 3 \times 3 + 16 = 160$ (1 canal d'entrée et 16 filtres de sortie de taille 3×3)
- Batch normalisation 1 : 32
- Couche convolutionnelle 2 : $16 \times 32 \times 3 \times 3 + 32 = 4,640$ (16 canaux d'entrée et 32 filtres de sortie de taille 3×3)
- Batch normalisation 2 : 64
- Couche fully connected 1 : $32 \times 56 \times 56 \times 64 + 64 = 6,422,592$ (32 canaux d'entrée d'une image réduite à 56×56 pixels et 64 neurones de sortie)
- Couche fully connected 2 : $64 \times 1 + 1 = 65$ (64 neurones d'entrée et 1 classe de

sortie)

- Nombre total de paramètres :

$$160 + 32 + 4,640 + 64 + 6,422,592 + 65 = 6,427,553$$

On remarque que la batch normalisation augmente de manière négligeable la complexité du réseau de neurones (96 paramètres en plus sur plus de 6 millions de paramètres) et qu'elle permet de lutter contre le sur-apprentissage et rend donc le modèle plus précis (% de précision avec la batch normalisation contre % sans).

Normalement le code fonctionne mais malgré toutes mes tentatives le kernel python de jupyter crashe sur mon ordi, je vous renvoie donc le code sans les pourcentages de précision des modèles (pourcentage qui était normalement assez bon sur le test que j'ai pu réaliser ce week-end). Veuillez m'excuser pour le retard de 30 minutes sur le rendu, j'ai essayé de régler ce bug avant de vous renvoyer le TP.