# Rapport du TP : Implémentation d'un Réseau de Neurones Convolutif pour la Classification MNIST

Ouzaina Marwane

28 août 2025

**Résumé**

Ce rapport présente l'implémentation d'un réseau de neurones convolutif (CNN) pour la classification de chiffres manuscrits sur la base de données MNIST. Une introduction théorique décrit les principes fondamentaux des CNN, leurs avantages et les calculs associés aux paramètres des couches. Ensuite, une implémentation complète en `PyTorch` est réalisée, suivie d'une analyse des résultats obtenus et des perspectives d'amélioration. the code python is available as : https://github.com/marwaneouz/convolution-classique-

## Table des matières

**1 Introduction** **3**

**2 Reponse aux Question** **3**

**3 Cadre théorique** **5**
3.1 Architecture Générale d'un CNN . . . . . 5
3.2 Dataset . . . . . 5
3.3 Principe des couches convolutives . . . . . 5
3.4 Calcul du nombre de paramètres . . . . . 6
3.5 Avantages des CNN . . . . . 6

**4 VGG-16** **6**
4.1 Architecture . . . . . 6
4.2 Implementation Framscratch . . . . . 7

**5 ResNet-50** **8**
5.1 Architecture . . . . . 8
5.2 implementation Framscratch . . . . . 8

**6 Résultats** **10**
6.1 CNN classique . . . . . 10
6.2 VGG-16 vs ResNet-50 . . . . . 12
6.3 LeNet 5 . . . . . 13
6.4 Comparaison des modeles . . . . . 13

**7 Discusion** **13**

**8 Conclusion** **15**

**A Code complet en PyTorch** **16**
A.1 Préparation des données . . . . . 16
A.2 Architecture du modèle . . . . . 16
A.3 Entraînement et évaluation . . . . . 17

# 1 Introduction

Les réseaux de neurones convolutifs (CNN) sont la pierre angulaire des systèmes modernes de vision par ordinateur. Grâce à leur capacité à extraire automatiquement des caractéristiques locales et hiérarchisées, ils surpassent largement les modèles traditionnels sur des tâches telles que la reconnaissance d'images. L'objectif de ce TP est de construire un CNN capable de classifier précisément des images de chiffres manuscrits issues du jeu de données MNIST.

# 2 Reponse aux Question

**1. Quelle est la taille de sortie d'une couche de convolution avec 8 filtres de taille (5,5), sans padding, et un stride de 1 pour une entrée de 32×32×3 ?**

La taille de sortie est calculée avec la formule :

$$\text{Taille} = \lfloor \frac{\text{taille d'entrée} - \text{taille du filtre} + 2 \times \text{padding}}{\text{stride}} \rfloor + 1$$

Pour une entrée de 32×32, un filtre de 5×5, sans padding (0), et un stride de 1 :

$$\text{Taille} = \lfloor \frac{32 - 5 + 2 \times 0}{1} \rfloor + 1 = 28$$

Avec 8 filtres, la sortie est de taille $28 \times 28 \times 8$.

**Réponse :** $28 \times 28 \times 8$

**2. Combien y a-t-il de poids à apprendre ?**

Chaque filtre a une taille de 5×5 et prend 3 canaux en entrée. Donc, un filtre a :

$$5 \times 5 \times 3 = 75 \text{ poids}$$

Avec 8 filtres :

$$8 \times 75 = 600 \text{ poids}$$

On ajoute 1 biais par filtre, soit 8 biais. Total :

$$600 + 8 = 608 \text{ poids}$$

**Réponse :** 608 poids

**3. Combien de poids pour une couche fully-connected produisant la même taille de sortie ?**

L'entrée est de $32 \times 32 \times 3 = 3072$ neurones. La sortie est de $28 \times 28 \times 8 = 6272$ neurones. Une couche fully-connected connecte chaque neurone d'entrée à chaque neurone de sortie :

$$3072 \times 6272 = 19\,267\,584 \text{ poids}$$

On ajoute 6272 biais (1 par neurone de sortie). Total :

$$19\,267\,584 + 6272 \approx 19\,273\,856 \text{ poids}$$

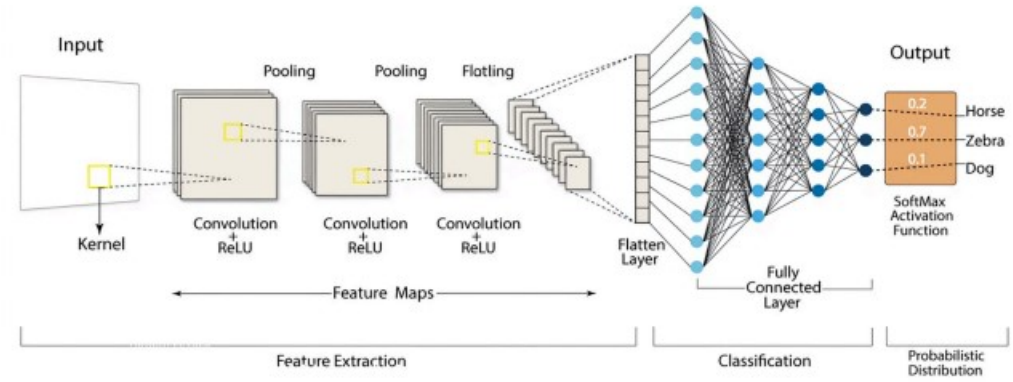
**Réponse :** Environ 19 273 856 poids

**4. Quels sont les avantages de la convolution par rapport aux couches fully-connected ?**

— **Moins de paramètres** : Les filtres partagent les poids, donc moins de calculs (608 poids contre 19 millions).
— **Connexion locale** : La convolution se concentre sur les motifs locaux (bords, textures), contrairement aux couches fully-connected qui ignorent la structure spatiale.
— **Invariance à la translation** : Les filtres détectent les motifs partout dans l'image.
— **Apprentissage hiérarchique** : Les couches de convolution apprennent des motifs simples puis complexes, idéal pour les images.

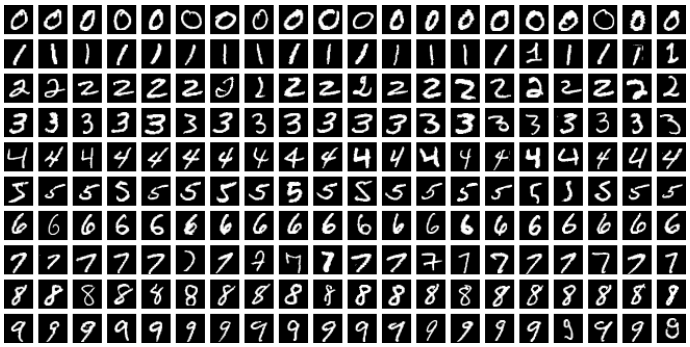
# 3 Cadre théorique

## 3.1 Architecture Générale d'un CNN




## 3.2 Dataset

Le dataset MNIST contient des images en niveaux de gris de chiffres manuscrits, fournissant un dataset bien structuré pour les tâches de classification d'images. Voici quelques exemples d'images du dataset :



L'exemple met en évidence la variété et la complexité des chiffres manuscrits dans l'ensemble de données MNIST, soulignant l'importance d'un ensemble de données diversifié pour l'entraînement de modèles de classification d'images robustes.

## 3.3 Principe des couches convolutives

Une couche convolutionnelle applique un ensemble de filtres (ou noyaux) sur une entrée pour en extraire des cartes de caractéristiques (feature maps). Chaque filtre glisse sur l'image d'entrée, réalisant un produit scalaire local, ce qui permet de détecter des motifs pertinents. Les principaux paramètres sont :

— Taille des filtres (kernels), ici $5 \times 5$
— Nombre de filtres (profondeur de sortie)
— Stride (décalage du filtre sur l'image)
— Padding (ajout de pixels pour contrôler la taille de sortie)

La taille spatiale en sortie d'une couche convolutionnelle sans padding est calculée par :

$$\text{taille\_sortie} = \left\lfloor \frac{\text{taille\_entrée} - \text{taille\_filtre} + 2 \times \text{padding}}{\text{stride}} \right\rfloor + 1$$

### 3.4 Calcul du nombre de paramètres

Par exemple, pour une couche avec 8 filtres de taille $5 \times 5$, appliquée sur une entrée de profondeur 3, chaque filtre comporte $5 \times 5 \times 3 = 75$ poids, plus 1 biais, soit 76 paramètres par filtre. Au total, $76 \times 8 = 608$ paramètres à apprendre.

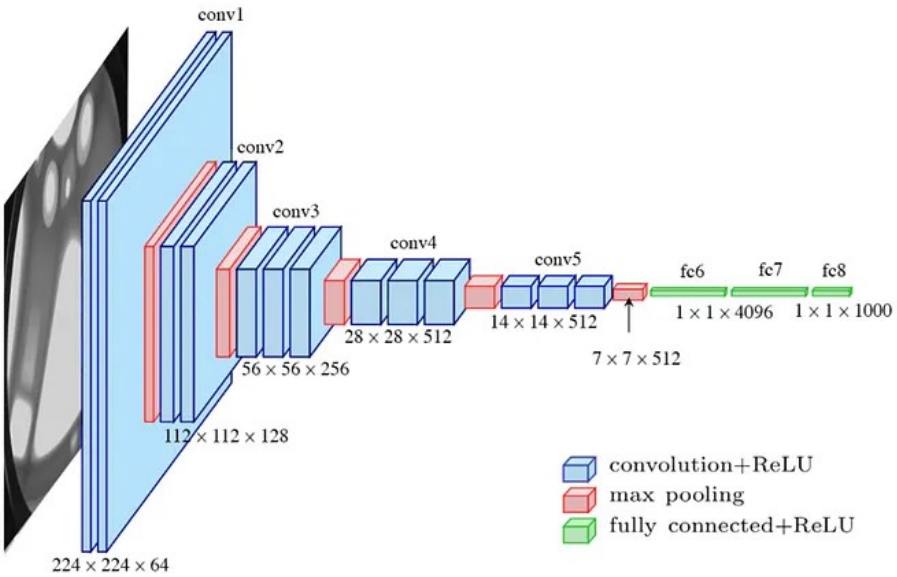
### 3.5 Avantages des CNN

Comparés aux couches entièrement connectées, les CNN offrent :
- Réduction drastique du nombre de paramètres
- Exploitation de la localité spatiale des images
- Partage des poids conduisant à une meilleure généralisation et une invariance à la translation

## 4 VGG-16

VGG16, developed by the Visual Geometry Group at the University of Oxford, is a deep convolutional neural network known for its simplicity and uniformity. Consisting of 16 layers, 13 convolutional layers with 3x3 filters, and 3 fully connected layers, it demonstrated remarkable performance in the 2014 ImageNet Large-Scale Visual Recognition Challenge (ILSVRC), achieving 92.7% classification accuracy. It is able to capture fine-grained details due to small filter sizes. Although computationally intensive, VGG16 set the stage for deeper networks and influenced subsequent models.

### 4.1 Architecture



| Layer | | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|---|
| Input | Image | 1 | 224 x 224 x 3 | - | - | - |
| 1 | 2 X Convolution | 64 | 224 x 224 x 64 | 3x3 | 1 | relu |
| | Max Pooling | 64 | 112 x 112 x 64 | 3x3 | 2 | relu |
| 3 | 2 X Convolution | 128 | 112 x 112 x 128 | 3x3 | 1 | relu |
| | Max Pooling | 128 | 56 x 56 x 128 | 3x3 | 2 | relu |
| 5 | 2 X Convolution | 256 | 56 x 56 x 256 | 3x3 | 1 | relu |
| | Max Pooling | 256 | 28 x 28 x 256 | 3x3 | 2 | relu |
| 7 | 3 X Convolution | 512 | 28 x 28 x 512 | 3x3 | 1 | relu |
| | Max Pooling | 512 | 14 x 14 x 512 | 3x3 | 2 | relu |
| 10 | 3 X Convolution | 512 | 14 x 14 x 512 | 3x3 | 1 | relu |
| | Max Pooling | 512 | 7 x 7 x 512 | 3x3 | 2 | relu |
| 13 | FC | - | 25088 | - | - | relu |
| 14 | FC | - | 4096 | - | - | relu |
| 15 | FC | - | 4096 | - | - | relu |
| Output | FC | - | 1000 | - | - | Softmax |

## 4.2 Implementation Framscratch

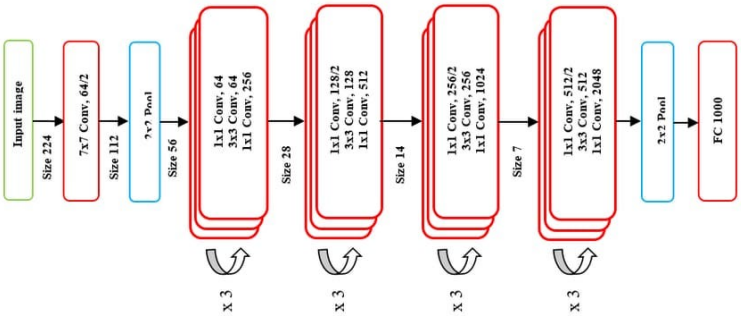
```
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Model

_input = Input((224,224,1))

conv1  = Conv2D(filters=64, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(_input)
conv2  = Conv2D(filters=64, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv1)
pool1  = MaxPooling2D((2, 2))(conv2)

conv3  = Conv2D(filters=128, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(pool1)
conv4  = Conv2D(filters=128, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv3)
pool2  = MaxPooling2D((2, 2))(conv4)

conv5  = Conv2D(filters=256, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(pool2)
conv6  = Conv2D(filters=256, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv5)
conv7  = Conv2D(filters=256, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv6)
pool3  = MaxPooling2D((2, 2))(conv7)

conv8  = Conv2D(filters=512, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(pool3)
conv9  = Conv2D(filters=512, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv8)
conv10 = Conv2D(filters=512, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv9)
pool4  = MaxPooling2D((2, 2))(conv10)
```

```
conv11 = Conv2D(filters=512, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(pool4)
conv12 = Conv2D(filters=512, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv11)
conv13 = Conv2D(filters=512, kernel_size=(3,3), padding="same",
    ↪ activation="relu")(conv12)
pool5  = MaxPooling2D((2, 2))(conv13)

flat   = Flatten()(pool5)
dense1 = Dense(4096, activation="relu")(flat)
dense2 = Dense(4096, activation="relu")(dense1)
output = Dense(1000, activation="softmax")(dense2)

vgg16_model  = Model(inputs=_input, outputs=output)
```

# 5 ResNet-50

ResNet signifie Réseau Résiduel. C'est une architecture de réseau de neurones innovante qui a été introduite pour la première fois par Kaiming He, Xiangyu Zhang, Shaoqing Ren et Jian Sun dans leur article de recherche en vision par ordinateur de 2015 intitulé « Deep Residual Learning for Image Recognition ».

## 5.1 Architecture




## 5.2 implementation Framscratch

```
import torch
import torch.nn as nn

class Bottleneck(nn.Module):
    # Expansion factor pour le nombre de filtres en sortie
    expansion = 4

    def __init__(self, in_channels, out_channels, stride=1, downsample=
        ↪ None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=1,
            ↪  bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size
            ↪ =3, stride=stride, padding=1, bias=False)
```

```
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv3 = nn.Conv2d(out_channels, out_channels * self.
            ↪ expansion, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out


class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=1000):
        super(ResNet, self).__init__()
        self.in_channels = 64

        # Premi re couche
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding
            ↪ =3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Couches de residual blocks
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        # Classification finale
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        # Initialisation des poids
        self._initialize_weights()

    def _make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
```

```
        if stride != 1 or self.in_channels != out_channels * block.
            ↪ expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels * block.
                    ↪ expansion, kernel_size=1, stride=stride, bias=
                    ↪ False),
                nn.BatchNorm2d(out_channels * block.expansion),
            )

        layers = []
        layers.append(block(self.in_channels, out_channels, stride,
            ↪ downsample))
        self.in_channels = out_channels * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
                    ↪ nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)


# Fonction pour cr er ResNet-50
def resnet50(num_classes=1000):
    return ResNet(Bottleneck, [3, 4, 6, 3], num_classes=num_classes)
```

# 6 Résultats

## 6.1 CNN classique

Le modèle atteint typiquement une précision supérieure à 98% sur MNIST après 10 époques, ce qui valide la qualité de l'architecture et du processus d'entraînement.

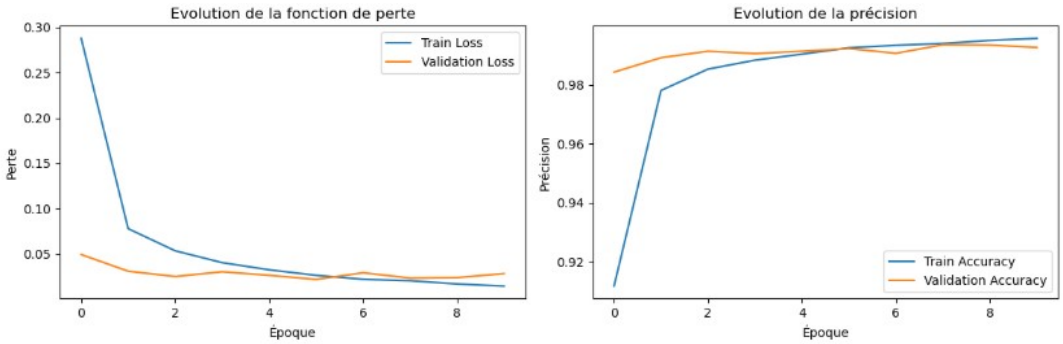



FIGURE 1 – Courbes de perte (Loss) et précision (Accuracy) pendant l'entraînement

La matrice de confusion et le rapport de classification montrent que la plupart des classes sont bien reconnues, avec quelques confusions marginales entre chiffres similaires.

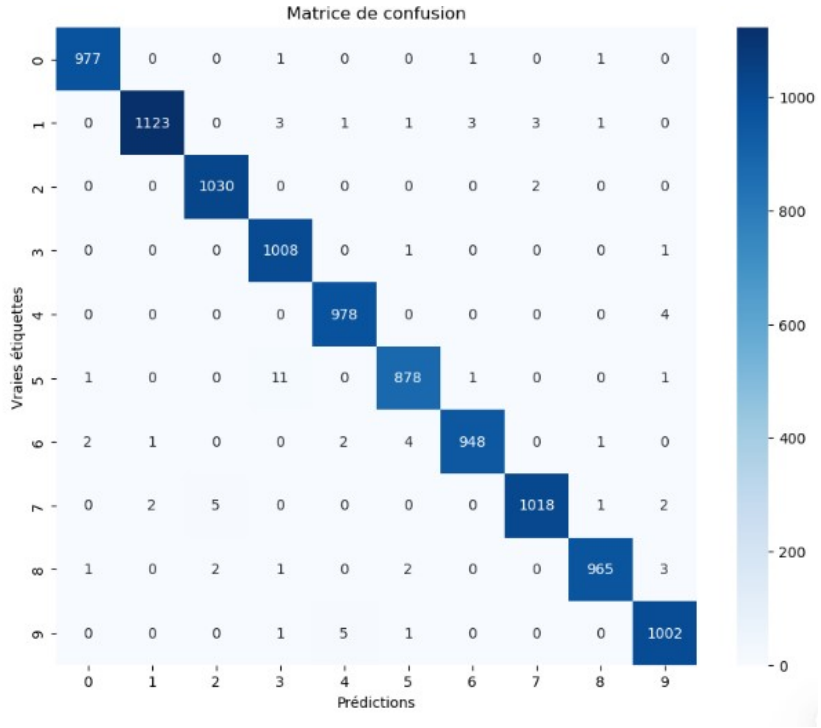



FIGURE 2 – Courbes de perte (Loss) et précision (Accuracy) pendant l'entraînement

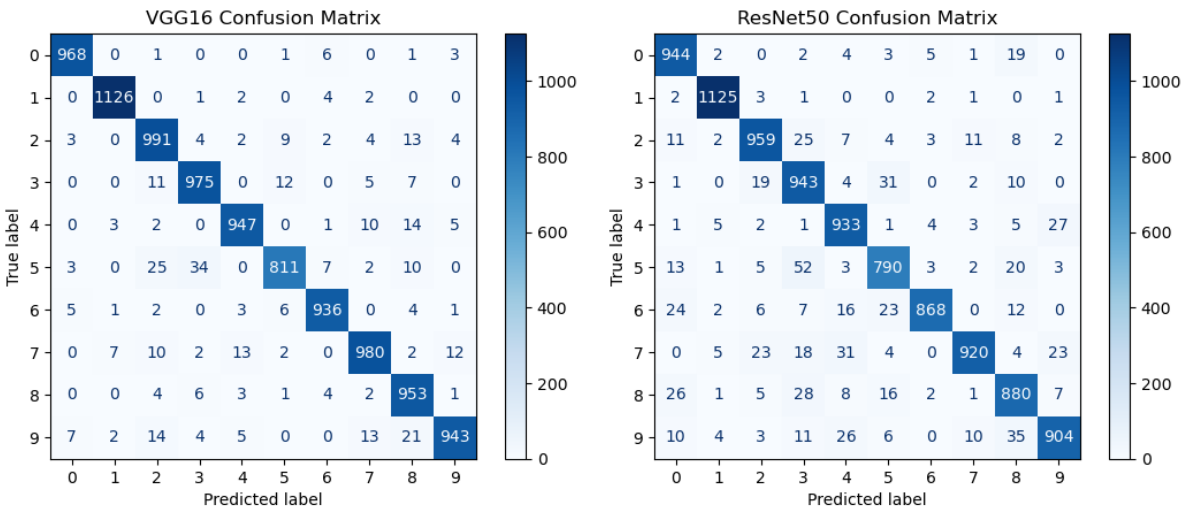
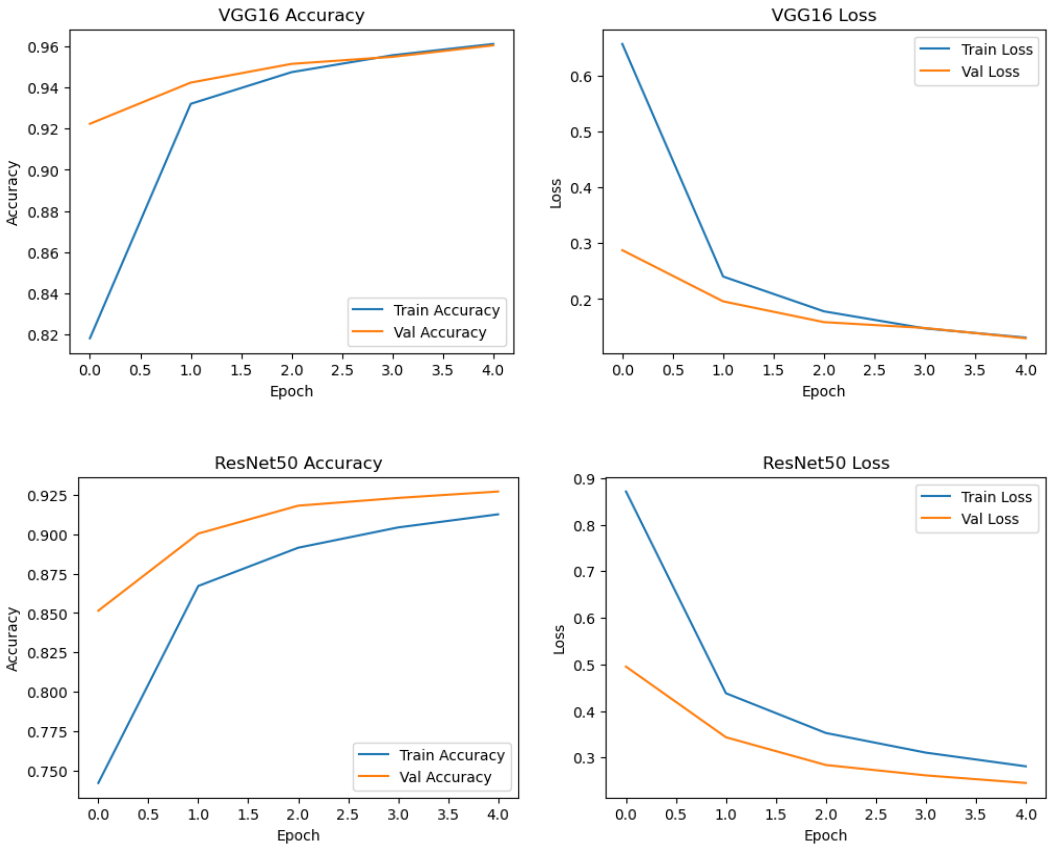
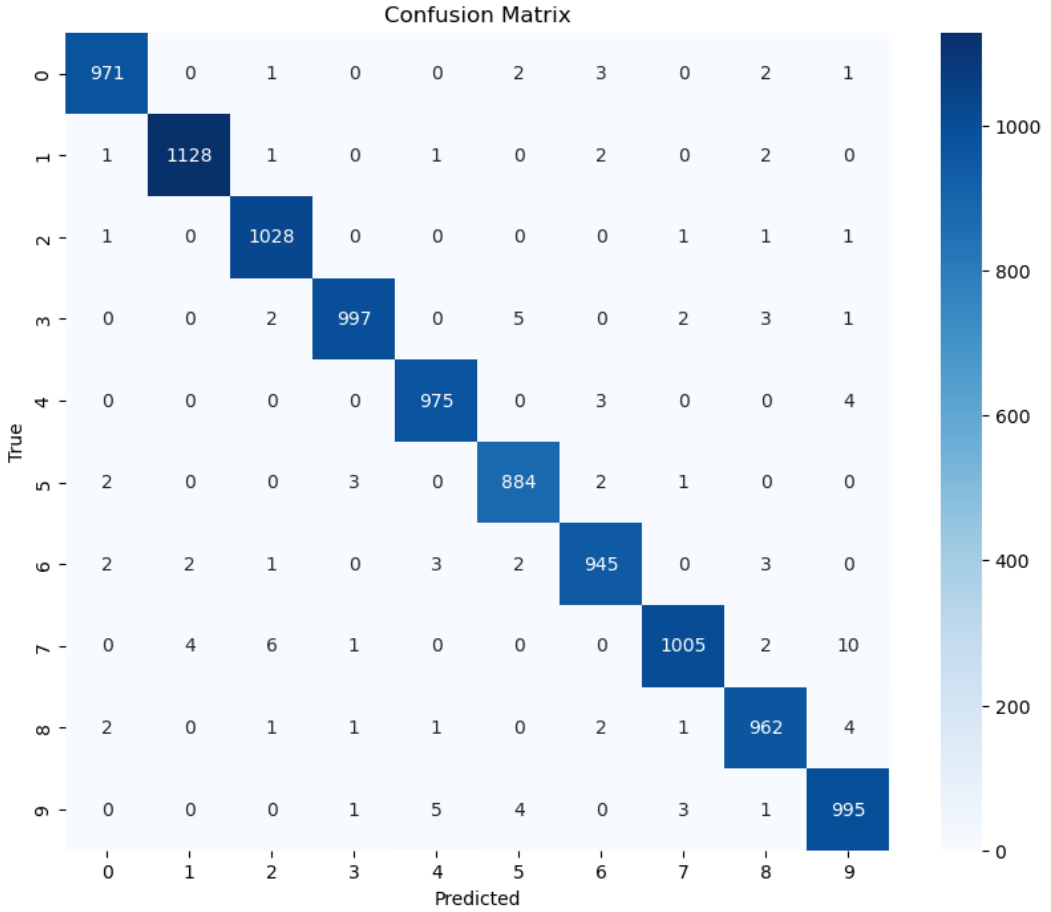


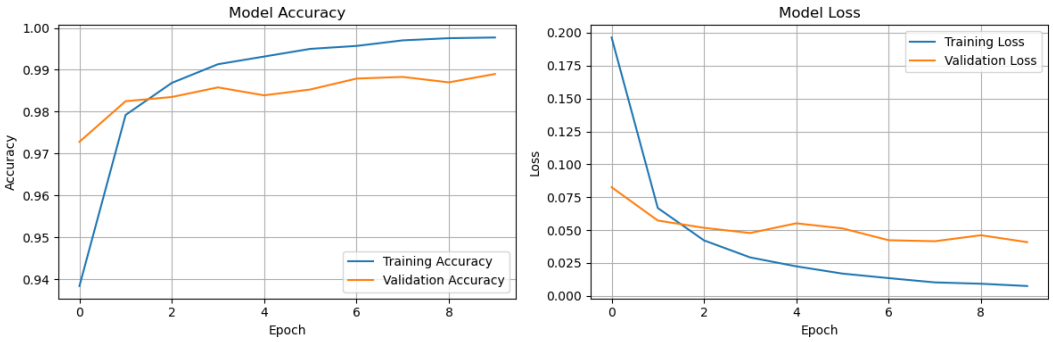

FIGURE 3 – Matrice de confusion VGG16 vs ResNet50

## 6.2 VGG-16 vs ResNet-50



## 6.3 LeNet 5




## 6.4 Comparaison des modeles

| Model | Accuracy | Test Loss |
|---|---|---|
| VGG16 | 96.29% | 0.1203 |
| ResNet50 | 92.65% | 0.2381 |
| LeNet5 | 98.90% | 0.0409 |

# 7 Discusion

Il est important de noter que le 'CNN Classique' fait référence à l'implémentation du CNN décrite en haut , qui a atteint une précision supérieure à 98% sur le dataset MNIST.

Bien que la perte sur l'ensemble de test ne soit pas explicitement mentionnée pour ce modèle, sa précision élevée le place en compétition directe avec LeNet5.

**LeNet5 : Le Plus Performant**

Le modèle LeNet5 se distingue comme le plus performant parmi ceux comparés, affichant la précision la plus élevée (98.90%) et la perte la plus faible (0.0409). Cette performance est remarquable étant donné que LeNet5 est une architecture plus ancienne et relativement simple par rapport à VGG16 ou ResNet50. Sa conception, optimisée pour la reconnaissance de chiffres manuscrits, lui permet d'exceller sur le dataset MNIST. Cela souligne l'importance d'une architecture adaptée à la tâche spécifique, même si elle n'est pas la plus complexe ou la plus récente.

**VGG16 : Bonne Performance mais Coût Élevé**

VGG16 obtient une précision respectable de 96.29% avec une perte de 0.1203. Bien que performant, il est surclassé par LeNet5 sur ce dataset. VGG16 est connu pour sa profondeur et l'utilisation de petits filtres convolutifs (3x3), ce qui lui permet de capturer des détails fins. Cependant, sa complexité et son nombre élevé de paramètres peuvent le rendre coûteux en termes de calcul et potentiellement sujet au surapprentissage sur des datasets plus petits ou moins variés, comme MNIST, s'il n'est pas correctement régularisé ou pré-entraîné.

**ResNet50 : Le Moins Performant sur MNIST**

ResNet50 affiche la précision la plus basse (92.65%) et la perte la plus élevée (0.2381) parmi les modèles comparés. ResNet50 est une architecture très profonde qui utilise des connexions résiduelles pour atténuer le problème de la dégradation du gradient dans les réseaux très profonds. Sa performance relativement faible sur MNIST peut s'expliquer par plusieurs facteurs : le dataset MNIST est simple et ne nécessite pas la complexité d'un modèle comme ResNet50. De plus, les architectures très profondes peuvent avoir du mal à converger efficacement sur des datasets de petite taille sans un pré-entraînement significatif sur des datasets plus grands (comme ImageNet), ce qui n'est pas mentionné ici pour ResNet50.

**CNN Classique : Une Implémentation Efficace**

Le CNN classique implémenté dans le rapport démontre une performance très solide avec une précision supérieure à 98%. Cela indique que même une architecture CNN relativement simple, bien conçue et entraînée, peut atteindre d'excellents résultats sur des tâches de classification d'images comme MNIST. Sa performance est comparable à celle de LeNet5, ce qui suggère que la complexité n'est pas toujours synonyme de meilleure performance, surtout pour des problèmes bien définis et des datasets relativement simples

# 8 Conclusion

Ce TP a permis de consolider la compréhension des réseaux convolutifs ainsi que leur mise en œuvre pratique sous PyTorch et Keras . La comparaison des modèles révèle que pour la tâche de classification de chiffres manuscrits sur MNIST, des architectures plus légères et spécifiquement conçues pour ce type de données, comme LeNet5 et le CNN classique, surpassent les modèles plus complexes et génériques comme VGG16 et ResNet50. Cela met en évidence l'importance de choisir une architecture de modèle appropriée à la complexité et à la nature du dataset. Pour des tâches plus complexes et des datasets plus volumineux, les architectures profondes comme VGG16 et ResNet50, souvent utilisées avec un pré-entraînement, montreraient probablement leur supériorité.

# A Code complet en PyTorch

The code is available in : https://github.com/marwaneouz/convolution-classique-

## A.1 Préparation des données

Nous utilisons la bibliothèque `torchvision` pour charger la base MNIST et appliquer une normalisation simple. Le batch est fixé à 128 pour l'entraînement.

Listing 1 – Préparation des données

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

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=
    ↪ transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=
    ↪ transform, download=True)

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

## A.2 Architecture du modèle

Le CNN est composé de trois couches convolutionnelles avec des padding pour conserver les dimensions, suivies chacune d'un max-pooling 2x2. Ensuite viennent deux couches entièrement connectées, avec dropout pour régulariser.

Listing 2 – Définition du CNN

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

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, padding=2)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=5, padding=2)
        self.pool3 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64*3*3, 100)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(100, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
```

```
        x = F.relu(self.conv3(x))
        x = self.pool3(x)
        x = x.view(-1, 64*3*3)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

## A.3 Entraînement et évaluation

Le modèle est entraîné avec l'optimiseur Adam sur 10 époques par la minimisation de la cross-entropy. À chaque époque, la précision sur l'ensemble de test est calculée.

Listing 3 – Boucle d'entraînement

```
import torch.optim as optim

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
n_epochs = 10

for epoch in range(n_epochs):
    model.train()
    running_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = correct / total
    print(f" poque  {epoch+1}/{n_epochs} - Perte moyenne: {running_loss/
       ↪ len(train_loader):.4f} - Pr cision test: {accuracy:.4f}")
```