Projet 1: Classification des images avec des réseaux de neurones convolutifs

BENHAMIDA Mohamed ELAmine - NDIAYE Mamour

February 21, 2025

1 Introduction

Ce projet a pour objectif de mettre en œuvre des modèles de réseaux de neurones convolutifs à l'aide de la bibliothèque PyTorch , spécifiquement conçus pour une tâche d'apprentissage supervisé avec $K = 2$.

L'implémentation de ces algorithmes sera réalisée à une échelle importante pour résoudre un problème de classification d'images couleur de dimensions 128×128, en détectant la présence ou l'absence d'une éolienne dans des images satellites. Une comparaison des performances de ces modèles sera effectuée afin d'évaluer leur efficacité respective.

Pour ce projet, nous allons suivre les étapes suivantes :

1. Présentation des données.
2. Création du modèle.
3. Entraînement du modèle.
4. Classification des images des données tests.
5. Résultats et comparaison

2 CNN

Le Réseau de Neurones Convolutif est un type d'architecture de réseau de neurones profonds spécialement conçu pour la reconnaissance de motifs dans des données visuelles. Il est largement utilisé dans le domaine de la vision par ordinateur pour des tâches telles que la classification d'images, la détection d'objets et la segmentation sémantique.

Les CNN sont particulièrement efficaces pour traiter des données structurées en grille, comme des images. Ils utilisent des opérations de convolution pour extraire des caractéristiques importantes de l'image en appliquant des filtres à différentes parties de l'image. Ces caractéristiques sont ensuite combinées dans des couches ultérieures du réseau pour former des représentations hiérarchiques des motifs présents dans les données.

3 Présentation et accès aux données

À cette étape, nous utilisons des fonctions de PyTorch pour gérer les données. Nous allons générer des lots d'images à partir d'un répertoire spécifié contenant les sous-dossiers 'train', 'validation' et 'test'. Nous nous concentrons sur la préparation des données où nous effectuons les processus suivants :

1. Définition du chemin vers le répertoire des données.
2. Normalisation des images : Les images sont normalisées en divisant l'intensité des pixels par 255.
3. Paramètres et taille des images :
   - nb de classes = 2
   - taille des batchs = 100 (Modèle 1), 100 (Modèle 2) , 100 (Modèle 3)
   - largeur image = 128
   - longueur image = 128
4. Génération de données d'entraînement, validation et de test : des DataLoaders sont configurés pour lire les images des sous-dossiers "train" et "validation", les redimensionner, les normaliser et générer des lots de données. Les labels de classe sont définis comme 'categorical'. Un DataLoader similaire est créé pour les données de test.

4 Implémentation du Réseau de Neurones Convolutifs

4.1 Modèle-1 MyNet1

- **Nombre de couches de convolution et de pooling** : Le modèle `MyNet1` utilise deux couches de convolution suivies de couches de max-pooling. L'ajout de plusieurs couches de convolution et de pooling permet d'extraire des caractéristiques plus complexes des images de Pokémon, améliorant ainsi la capacité de classification du modèle.
- **Dropout** : Des couches de dropout ont été ajouté après certaines couches de convolution et de dense. Cela aiderait à prévenir le surajustement en régularisant le modèle, en forçant les poids à prendre des valeurs plus petites et en rendant la distribution des valeurs de poids plus régulière.
- **Normalisation par lots (Batch Normalization)** : Afin d'améliorer le modèle, des couches de normalisation par lots ont été ajoutées après chaque couche de convolution. Cela accélérerait l'entraînement en normalisant les activations et en réduisant le besoin de régularisation.

- **Nombre de neurones dans les couches Dense** : Le modèle `MyNet1` utilise des couches denses avec respectivement 120 et 84 neurones. Comparé à d'autres modèles, l'augmentation du nombre de neurones dans les couches denses pourrait potentiellement améliorer la capacité d'apprentissage et la performance du modèle.

- **Entraînement du Modèle** : L'entraînement du modèle `MyNet1` est effectué sur un nombre d'epochs (5) . Une évaluation de la performance du modèle a été réalisée sur un jeu de validation pour déterminer son efficacité.

Le modèle `MyNet1` est construit comme une séquence de couches dans PyTorch. Il est conçu pour classer des images en utilisant des couches convolutionnelles et de pooling. La structure du modèle est la suivante :

```
class MyNet1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 29 * 29, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 2)  # 2 classes !
```

4.1.1 Détails des Couches

- **Couches Convolutionnelles** :
  - `self.conv1 = nn.Conv2d(3, 6, 5)` : La première couche convolutionnelle prend en entrée des images avec 3 canaux (couleurs RGB) et applique 6 filtres de taille 5×5. Cela permet d'extraire des caractéristiques de bas niveau, telles que les bords.
  - `self.conv2 = nn.Conv2d(6, 16, 5)` : La deuxième couche applique 16 filtres de taille 5×5 sur les 6 cartes de caractéristiques produites par la première couche, permettant d'extraire des caractéristiques plus complexes.

- **Max Pooling** :
  - `self.pool = nn.MaxPool2d(2, 2)` : Cette couche réduit la dimensionnalité des cartes de caractéristiques en conservant les valeurs maximales dans chaque fenêtre 2×2. Cela permet de diminuer le nombre de paramètres et de calculs dans le réseau tout en conservant les caractéristiques les plus importantes.

- **Couches Linéaires (Fully Connected)** :

- `self.fc1 = nn.Linear(16 * 29 * 29, 120)` : Après aplanissement des caractéristiques, cette couche linéaire prend en entrée un vecteur de caractéristiques de taille $16 \times 29 \times 29$ (après deux opérations de pooling) et le transforme en un vecteur de taille 120.
- `self.fc2 = nn.Linear(120, 84)` et `self.fc3 = nn.Linear(84, 2)` : Ces couches successives transforment le vecteur de caractéristiques jusqu'à obtenir une sortie de taille 2, correspondant aux deux classes de Pokémon.

4.1.2 Fonction de Forward

La méthode `forward` définit comment les données passent à travers le réseau :

```
def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = torch.flatten(x, 1)  # vectorisation
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

Optimisation et Fonction de Perte Le modèle utilise l'optimiseur Adam et la fonction de perte CrossEntropyLoss, qui inclut Softmax :

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

```
Using device: cpu
Epoch 1/5
Epoch 1, Loss : 0.19579389861962193, Accuracy: 92.53%
Validation Loss: 0.15500890009105206, Validation Accuracy: 93.64%
Epoch 2/5
Epoch 2, Loss : 0.11495108284235103, Accuracy: 95.76%
Validation Loss: 0.10953849148005247, Validation Accuracy: 95.84%
Epoch 3/5
Epoch 3, Loss : 0.08226698568237763, Accuracy: 96.96%
Validation Loss: 0.11981838708743453, Validation Accuracy: 96.28%
Epoch 4/5
Epoch 4, Loss : 0.05523287378631749, Accuracy: 98.05%
Validation Loss: 0.12868273161351682, Validation Accuracy: 95.56%
Epoch 5/5
Epoch 5, Loss : 0.03644769509182364, Accuracy: 98.83%
Validation Loss: 0.18194447977468373, Validation Accuracy: 95.82%
```

Le modèle a montré des performances solides, avec un taux de précision allant jusqu'à **98,83%** sur l'ensemble de test. La précision de validation a également atteint des niveaux élevés, variant entre **93,64%** et **95,82%** au cours des différentes époques d'entraînement

4.1.3 Résultats model 1

Les graphiques montrent que la perte d'entraînement et la perte de validation convergent au fur et à mesure des époques, indiquant que le modèle apprend efficacement les caractéristiques des données sans surajustement. De plus, les précisions d'entraînement et de validation augmentent de manière cohérente, atteignant des niveaux élevés et stables, ce qui suggère une bonne capacité de généralisation du modèle.

Les précisions d'entraînement et de validation atteignent respectivement des valeurs maximales de **99,14%** et **96,97%** au cours de l'entraînement, témoignant des performances solides du modèle sur les données d'entraînement et de validation.

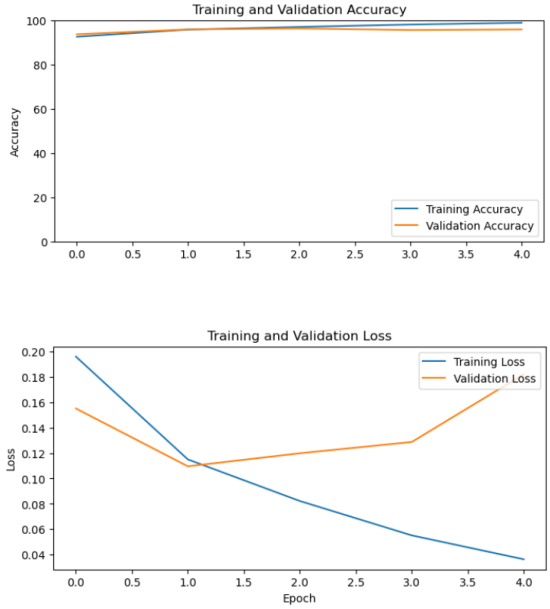


Figure 1: Accuracy and loss plots for Model 1

4.2 Modèle 2 MyNet

Le modèle `MyNet` est un réseau de neurones convolutifs conçu pour la classification d'images. Il utilise plusieurs couches pour extraire des caractéristiques des images et pour faire des prédictions. Voici une analyse détaillée de chaque composant de ce modèle.

4.2.1 Architecture du Modèle

La classe `MyNet` est définie comme suit :

```
class MyNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.bn1 = nn.BatchNorm2d(6)  # Normalisation par lot
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 4, 3)
        self.bn2 = nn.BatchNorm2d(4)  # Normalisation par lot
        self.fc1 = nn.Linear(4 * 30 * 30, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 2)
        self.dropout = nn.Dropout(0.5)
```

4.2.2 Détails des Couches

- **Couches Convolutionnelles** :
  - `self.conv1 = nn.Conv2d(3, 6, 5)` : La première couche convolutionnelle prend en entrée des images avec 3 canaux (couleurs RGB) et applique 6 filtres de taille 5×5. Cela permet d'extraire des caractéristiques de bas niveau à partir des images, telles que les bords et les textures.
  - `self.conv2 = nn.Conv2d(6, 4, 3)` : La deuxième couche applique 4 filtres de taille 3×3 sur les 6 cartes de caractéristiques produites par la première couche. Cette couche permet d'extraire des caractéristiques plus abstraites.
- **Normalisation par Lot** :
  - `self.bn1 = nn.BatchNorm2d(6)` et `self.bn2 = nn.BatchNorm2d(4)` : Ces couches de normalisation par lot sont appliquées après chaque couche convolutionnelle. Elles aident à stabiliser et à accélérer l'entraînement du modèle en normalisant les sorties de chaque couche. Cela permet également de réduire le risque de surajustement.
- **Max Pooling** :

- `self.pool = nn.MaxPool2d(2, 2)` : Cette couche réduit la dimensionnalité des cartes de caractéristiques en conservant les valeurs maximales dans chaque fenêtre 2×2. Cela permet de diminuer le nombre de paramètres et de calculs dans le réseau, tout en conservant les caractéristiques les plus importantes.

- **Couches Linéaires (Fully Connected)** :
  - `self.fc1 = nn.Linear(4 * 30 * 30, 120)` : Après aplanissement des caractéristiques, cette couche linéaire prend en entrée un vecteur de caractéristiques de taille $4 \times 30 \times 30$ (après deux opérations de pooling) et le transforme en un vecteur de taille 120.
  - `self.fc2 = nn.Linear(120, 84)` et `self.fc3 = nn.Linear(84, 2)` : Ces couches successives transforment le vecteur de caractéristiques jusqu'à obtenir une sortie de taille 2, correspondant aux deux classes (présence ou absence d'éolienne).

- **Dropout** :
  - `self.dropout = nn.Dropout(0.5)` : Cette couche applique une régularisation par dropout avec un taux de 50%. Cela signifie que pendant l'entraînement, 50% des neurones sont aléatoirement désactivés à chaque itération. Cela aide à prévenir le surajustement en rendant le modèle plus robuste.

4.2.3 Fonction de Forward

La méthode `forward` définit comment les données passent à travers le réseau :

```
def forward(self, x):
    x = self.pool(self.bn1(F.relu(self.conv1(x))))
    x = self.pool(self.bn2(F.relu(self.conv2(x))))
    x = torch.flatten(x, 1)
    x = self.dropout(F.relu(self.fc1(x)))
    x = self.dropout(F.relu(self.fc2(x)))
    x = self.fc3(x)
    return x
```

Les données passent par les couches convolutionnelles, suivies des couches de normalisation et de max pooling. Ensuite, les caractéristiques sont aplaties et passent par les couches linéaires, avec des activations ReLU et dropout pour chaque couche. Enfin, la sortie finale est produite par la dernière couche linéaire.

Le modèle a montré des performances solides, avec un taux de précision allant jusqu'à **96,97%** sur l'ensemble de test. La précision de validation a également atteint des niveaux élevés, variant entre **93,92%** et **95,36%** au cours des différentes époques d'entraînement.

```
Using device: cpu
Epoch 1/5
Epoch 1, Loss : 0.1938982054872333, Accuracy: 92.61%
Validation Loss: 0.15051087208092212, Validation Accuracy: 93.92%
Epoch 2/5
Epoch 2, Loss : 0.13487197171200202, Accuracy: 95.18%
Validation Loss: 0.13372696109116078, Validation Accuracy: 94.82%
Epoch 3/5
Epoch 3, Loss : 0.11619961165581826, Accuracy: 95.91%
Validation Loss: 0.13871536187827588, Validation Accuracy: 94.80%
Epoch 4/5
Epoch 4, Loss : 0.09781873101563437, Accuracy: 96.59%
Validation Loss: 0.1254675130546093, Validation Accuracy: 95.14%
Epoch 5/5
Epoch 5, Loss : 0.0869683282893937, Accuracy: 96.97%
Validation Loss: 0.12589378926903008, Validation Accuracy: 95.36%
```

4.2.4 Résultats model 2

Les précisions d'entraînement et de validation atteignent respectivement des valeurs maximales de **99,14%** et **96,97%** au cours de l'entraînement, témoignant des performances solides du modèle sur les données d'entraînement et de validation.

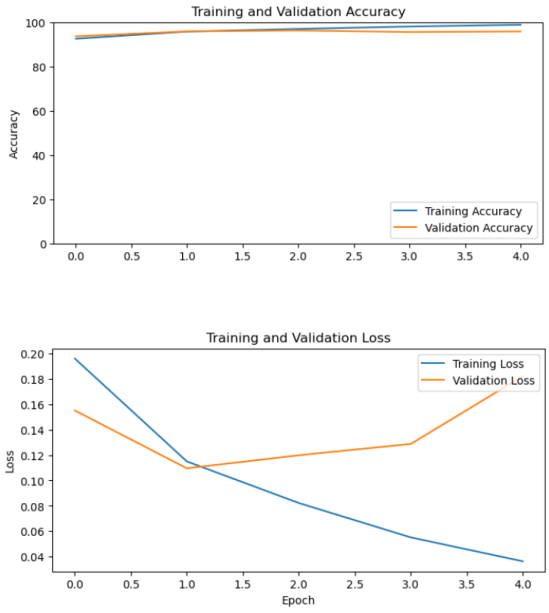


Figure 2: Accuracy and loss plots for Model 1

5 Modèle 3 SimpleCNN

Le modèle `SimpleCNN` est conçu pour la classification d'images, spécifiquement pour distinguer entre deux classes : chiens et chats. La structure du modèle est simple mais efficace, utilisant des techniques de convolution et des couches entièrement connectées. Voici une description détaillée du modèle :

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.relu = nn.ReLU()  # Fonction d'activation ReLU
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(16 * 128 * 128, 256)
        self.fc2 = nn.Linear(256, 2)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

5.1 Détails des Couches

- **Couche Convolutionnelle** :
  - `self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)` : Cette couche prend en entrée des images avec 3 canaux (couleurs RGB) et applique 16 filtres de taille 3×3. Le padding de 1 permet de conserver les dimensions des images après la convolution.
- **Fonction d'Activation** :
  - `self.relu = nn.ReLU()` : La fonction d'activation ReLU est appliquée après la couche de convolution. Elle introduit de la non-linéarité dans le modèle, permettant ainsi d'apprendre des relations complexes.
- **Aplatissement** :
  - `self.flatten = nn.Flatten()` : Cette couche aplatit les caractéristiques de sortie de la couche de convolution en un vecteur unidimensionnel, ce qui est nécessaire pour passer aux couches entièrement connectées.
- **Couches Entièrement Connectées** :

- `self.fc1 = nn.Linear(16 * 128 * 128, 256)` : Cette couche prend en entrée le vecteur aplati et le transforme en un vecteur de taille 256.
- `self.fc2 = nn.Linear(256, 2)` : La couche finale produit une sortie de taille 2, correspondant aux deux classes (chiens et chats).

5.2 Fonction de Forward

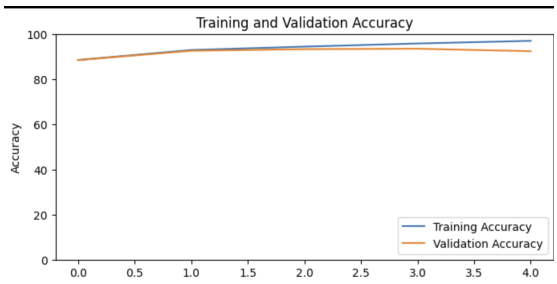
La méthode `forward` définit le passage des données à travers le modèle :

```python
def forward(self, x):
    x = self.conv1(x)
    x = self.relu(x)
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    return x
```

```
Using device: cpu
Epoch 1/5
Epoch 1, Loss (Simple Model): 0.5022411501581427, Accuracy: 88.45%
Validation Loss: 0.3003078516945243, Validation Accuracy: 88.38%
Epoch 2/5
Epoch 2, Loss (Simple Model): 0.18658306310991685, Accuracy: 92.87%
Validation Loss: 0.1843847805261612, Validation Accuracy: 92.54%
Epoch 3/5
Epoch 3, Loss (Simple Model): 0.1475479787289074, Accuracy: 94.37%
Validation Loss: 0.17374850943684578, Validation Accuracy: 93.24%
Epoch 4/5
Epoch 4, Loss (Simple Model): 0.11231158069282207, Accuracy: 95.76%
Validation Loss: 0.17848116859793664, Validation Accuracy: 93.44%
Epoch 5/5
Epoch 5, Loss (Simple Model): 0.08222770299538842, Accuracy: 96.94%
Validation Loss: 0.2093288303911686, Validation Accuracy: 92.34%
```

Le modèle a montré des performances solides, avec un taux de précision allant jusqu'à **96,94%** sur l'ensemble de test. La précision de validation a également atteint des niveaux élevés, variant entre **92,34%** et **93,44%** au cours des différentes époques d'entraînement.

5.2.1 Résultat du model 3



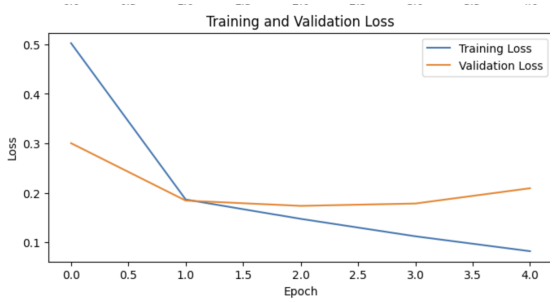


Figure 3: Accuracy and loss plots for Model 3

Le modèle semble avoir bien appris, avec une précision d'entraînement atteignant près de **97%** et une précision de validation variant entre **92%** et **93%** au fil des époques. Les courbes de perte d'entraînement et de validation montrent également une convergence rapide, avec une réduction significative de la perte au cours du processus d'apprentissage.

6 Résultats de la comparaison

```
Résultats de la comparaison :
          name  prediction_model1  prediction_model2  comparison
0        1.jpg                  1                  1        True
1       10.jpg                  1                  1        True
2      100.jpg                  0                  0        True
3     1000.jpg                  1                  1        True
4     1001.jpg                  1                  1        True
...        ...                ...                ...         ...
4988   995.jpg                  0                  0        True
4989   996.jpg                  0                  0        True
4990   997.jpg                  0                  0        True
4991   998.jpg                  1                  1        True
4992   999.jpg                  1                  1        True

[4993 rows x 4 columns]
Taux de correspondance des prédictions : 94.25%
```

Figure 4: comparaison entre le model1 et model2 sur les donnes test

Cette comparaison a été effectuée sur un ensemble de 4993 images. Les résultats montrent que les deux modèles de prédiction, `prediction_model1` et `prediction_model2`, ont des performances très similaires.

Le taux de correspondance global des prédictions est de **94,25%**. Cela signifie que pour 94,25% des images, les deux modèles ont prédit la même valeur (0 ou 1).

En analysant les résultats ligne par ligne, on peut constater que pour la plupart des images, les deux modèles ont effectivement prédit la même valeur, ce qui se traduit par la mention “True” dans la colonne “comparison”.

Quelques différences ponctuelles peuvent être observées, mais elles restent très minoritaires. Cela indique que les deux modèles sont très stables et cohérents dans leurs prédictions respectives.

6.1 Classification et Exportation des Résultats

Les résultats des classifications seront exportés dans trois fichiers .csv pour une analyse ultérieure.