Transfert Learning CNN

Réalisé par : Mathis Aulagnier
Cours : Vision Artificielle (8INF804)

1 Objectif du TP

L'objectif de ce travail pratique est d'explorer les réseaux de neurones convolutifs (CNN) à travers deux approches distinctes :

- **Transfer Learning** : utilisation d'un modèle CNN pré-entrainé, tel que ResNet ou VGG16, et ajustement de ses paramètres pour effectuer une nouvelle tâche de classification sur un autre ensemble de données.
- **Création d'une architecture personnalisée** : conception d'un modèle CNN sur mesure, entraîné de zéro sur le même ensemble de données, afin de comparer ses performances avec celles du modèle utilisant le transfer learning.

Pour accomplir ce projet, j'ai utilisé de nombreux extraits de code issus du cours, en particulier pour définir les fonctions d'entraînement et évaluer les performances. Ces mêmes fonctions ont été abordées dans le tutoriel 5, consacré au *transfer learning* et au *fine-tuning*, qui m'a servi de référence.

1.1 Le choix du dataset

Notre dataset, **Chest-XRAY**, contient des images radiographiques de thorax de tailles variées. Ces images sont classifiées en trois catégories : les patients atteints de pneumonie virale, de pneumonie bactérienne, et les patients sains. L'objectif de notre modèle est de détecter la présence ou l'absence de pneumonie, afin de fournir une première analyse automatique pour aider les radiologues.

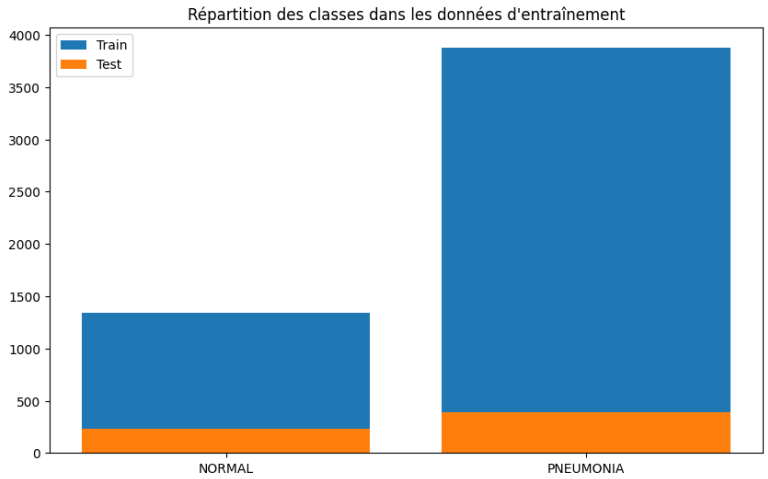

Figure 1: Répartition des classes dans le dataset Chest-XRAY. On remarque une sur-représentation de la classe **PNEUMONIA**, ce qui peut introduire un déséquilibre des données à prendre en compte lors de l'entraînement.

Le dataset pourrait également être utilisé pour distinguer les pneumonie virale ou bactérienne. Cela nécessiterait cependant une séparation supplémentaire des classes, afin de structurer les données de manière à ce que l'IA puisse identifier ces sous-catégories spécifiques. Cette séparation pourrait augmenter la complexité du modèle mais offrirait des informations médicalement plus pertinentes. Vous trouverez un aperçu du dataset en annexe A.

1.2 Clarification des concepts

Transfer learning : Le transfer learning consiste à utiliser un modèle pré-entraîné (généralement sur un large ensemble de données) comme point de départ pour une nouvelle tâche. L'idée est de tirer parti des connaissances générales qu'il a acquises lors de l'entraînement initial, afin de résoudre une tâche spécifique sans avoir besoin de tout réapprendre.

Fine-tuning : Le fine-tuning est une étape complémentaire du transfer learning. Après avoir chargé un modèle pré-entraîné, on ajuste certaines ou toutes ses couches en les réentraînant sur un nouvel ensemble de données, afin d'adapter le modèle de manière plus précise à la nouvelle tâche.

mini-batch :Le mini-batch est un sous-ensemble de l'ensemble de données d'entraînement. Plutôt que d'utiliser l'ensemble complet des données pour mettre à jour les poids du modèle à chaque itération, le mini-batch permet de diviser les données en petits lots. Cela permet d'accélérer l'entraînement et de réduire la mémoire nécessaire. De plus, l'utilisation de mini-batches introduit une certaine stochasticité dans l'entraînement, ce qui peut aider à éviter les minima locaux et à améliorer la généralisation du modèle.

époque (epoch) : Une époque est une passe complète à travers l'ensemble de données d'entraînement. Pendant une époque, chaque exemple de l'ensemble de données est utilisé une fois pour mettre à jour les poids du modèle. Le nombre d'époques est un hyperparamètre important qui détermine combien de fois le modèle verra l'ensemble de données d'entraînement. Un nombre suffisant d'époques est nécessaire pour que le modèle converge, mais trop d'époques peuvent entraîner un surapprentissage.

Je vais maintenant vous présenter le notebook de manière progressive. Nous examinerons chaque grande étape du code, que vous pourrez suivre en vous référant au notebook.

2 Chargement des Bibliothèques

Nous commençons par importer les bibliothèques nécessaires pour la manipulation des données et des modèles.

Listing 1: Importation des bibliothèques

```
import numpy as np  # Bibliotheque pour
    \ les operations sur les tableaux
import pandas as pd  # Bibliotheque pour la manipulation et l'analyse de donnees

import os  # Module pour les interactions avec le systeme d'exploitation
from datetime import datetime  # Module pour manipuler les dates et les heures

from tqdm import tqdm  # Bibliotheque pour afficher des barres de progression

from sklearn.metrics import {
    accuracy_score,
    balanced_accuracy_score,
    f1_score,
    cohen_kappa_score,
    top_k_accuracy_score,
    confusion_matrix,
    classification_report
} # Fonctions pour evaluer les performances des modeles de machine learning

import plotly.express as px  # Bibliotheque pour creer des visualisations
import plotly.subplots as sp  # Module pour creer des sous-graphiques

import torch  # Bibliotheque pour le calcul tensoriel et l'apprentissage profond
import torch.nn as nn  # Module pour definir des reseaux de neurones
import torch.optim as optim  # Module pour les algorithmes d'optimisation
import torchvision  # Bibliotheque pour les modeles de vision par ordinateur
from torchvision import models  # Module pour importer des modeles pre-entraines
from torch import flatten  # Fonction pour aplatir les tenseurs
import torchvision.transforms as transforms  # Module pour les transformations d'images
from torch.utils.data import random_split
# Fonction pour diviser aleatoirement les ensembles de donnees.
```

3 Analyse et évaluation de nos modèles

Dans cette section, nous présentons plusieurs fonctions qui vont nous être particulièrement utiles pour la suite. Au-delà de notre sujet, elles peuvent se trouver utiles pour l'analyse et l'évaluation des modèles de réseaux de neurones en général. Ces fonctions, présentes dans les sections *Other tools* et *Scores et graphe* du notebook, permettent d'extraire des informations importantes sur les paramètres du modèle et d'évaluer ses performances sur les ensembles de données de test et de validation. De plus, elles offrent des possibilités intéressantes.

3.1 Fonctions et leurs rôles

get_model_information(model) : Cette fonction extrait et affiche des informations détaillées sur les paramètres de chaque couche du modèle de machine learning, en séparant les paramètres entraînables (qui requièrent des gradients) et non entraînables (qui ne sont pas modifiés pendant l'entraînement)

Grâce a cette fonction, nous avons extrait le nombre de paramètres entraînables de nos modèles :

- Modèle_tl (ResNet préparé pour le transfert learning) : 13,696
- Modèle (CNN standard) : 2,326,082

Ces deux chiffres montrent clairement que le modèle avec plus de paramètres sera beaucoup plus long à entraîner, car il nécessite de mettre à jour un plus grand nombre de poids et de biais. À l'inverse, le ResNet, utilisé pour le transfert learning, sera plus rapide à entraîner car la majorité de ses paramètres sont déjà appris. Cependant, il sera moins flexible car les couches peuvent être gelées et seules quelques couches finales sont modifiables.

compute_accuracy(labels, outputs) calcule l'accuracy pour un mini-batch de données.

vec_to_int(y_true, y_predicted) transforme les vecteurs *argmax* en entiers. Elle est essentielle pour calculer des métriques comme l'accuracy et le F1-Score.

model_performances(y_true, y_predicted, loss, my_score_df) calcule plusieurs métriques de performance pour les prédictions du modèle.

show_compute_model_performances(y_true, y_predicted, loss, my_score_df, classes) : Cette fonction est similaire à `model_performances`, cette fonction est spécifiquement adaptée pour évaluer les performances sur l'ensemble de test. Elle inclut également l'affichage de la matrice de confusion et du rapport de classification.

create_score_df(training_epoch_scores, validation_epoch_scores, score_type) crée un DataFrame qui trace l'évolution d'une métrique (par exemple : l'accuracy) au fil des époques d'entraînement et de validation.

plot_score_graphs(training_epoch_scores, validation_epoch_scores) : Enfin cette fonction génère des graphiques pour visualiser l'évolution des scores de performance. Elle utilise la majorité des fonctions que l'on vient de voir, afin d'obtenir ses données pour le plot.

4 Processus d'Entraînement et de test du Modèle

L'entraînement d'un modèle de machine learning consiste à ajuster les paramètres du modèle pour minimiser une fonction de perte, en utilisant des données d'entraînement. Cette section présente les différentes étapes du processus d'entraînement, illustrées par des fonctions Python qui effectuent les calculs nécessaires.

4.1 Calcul des sorties du modèle et évaluation des performances

La première étape consiste à calculer les sorties du modèle pour des entrées spécifiques et à évaluer les performances pour chaque mini-Batch. Cela se fait à l'aide de la fonction `compute_model_outputs`. Cette fonction prend en entrée les données d'entraînement (`inputs`) et les labels associés (`labels`), et elle retourne les sorties du modèle ainsi que nos métriques de performance.

Puis on trouve la fonction principale pour entraîner le modèle est `train_model`, qui gère plusieurs époques d'entraînement. À chaque époque, elle calcule la perte et l'exactitude pour tous les mini-Batch d'entraînement, met à jour les poids du modèle, puis évalue les performances sur le jeu de validation.

On retrouve bien les définitions de l'époque (epoch) et du mini-Batch avec les boucles qui parcourent nos données :

Listing 2: Train_model

```
# Entrainement du reseau de neurones
def train_model(...):
    # ...
    # Indiquer au modele qu'il est en phase d'entrainement
    model.train()

    # Pour chaque epoque
    for epoch in range(epoch_number):
    # ...
        # Assigner l'iterateur tqdm a la variable "progress_epoch"
        with tqdm(train_loader, unit=" mini-batch") as progress_epoch:

            # Pour chaque mini-batch defini dans via la variable "progress_epoch"
            for inputs, labels in progress_epoch:
                # ...
```

Enfin, après chaque époque d'entraînement, il est essentiel de valider les performances du modèle sur un ensemble de validation pour éviter le sur-apprentissage. La fonction `validate_model` effectue cette tâche en évaluant la perte et l'exactitude sur le jeu de validation.

4.2 Test du Modèle

Après l'entraînement, il est crucial d'évaluer les performances du modèle sur un ensemble de test pour s'assurer qu'il généralise bien aux nouvelles données. La fonction `test_model` effectue cette tâche en évaluant la loss et l'accuracy sur le jeu de test.

Ces étapes permettent de s'assurer que le modèle est correctement entraîné et évalué.

Désormais, maintenant que les grandes fonctions tirées des tutoriels 3 et 5 du cours ont été expliquée. Nous allons rentrer dans le vif du sujet avec le choix des paramètres, utilisateurs et la création de nos modèles.

5 Paramètres Utilisateurs

L'entraînement de nos modèles repose sur plusieurs paramètres clés. Ceux-ci influencent la vitesse et l'efficacité de l'apprentissage. Voici quelques paramètres essentiels que nous avons définis pour simplifier le code d'entraînement :

Listing 3: Code d'entraînement simplifié

```
# Taille du mini-batch
batch_size = 32

# Taux d'apprentissage
learning_rate = 0.001

### Pour le transfert learning
epoch_number_tl = 3

### Pour notre CNN
epoch_number_cnn = 15
```

Nos modèles sont entraînés sur un GPU, spécifiquement le P100, via la plateforme Kaggle. Cette configuration GPU permet de réduire significativement le temps d'exécution, qui est en moyenne d'environ 60 minutes pour l'ensemble du Notebook.

Vous remarquez que le nombre d'époques utilisé pour le transfer learning est bien plus petit (3) que celui utilisé pour notre propre CNN (15). Cela s'explique par le fait que les modèles sont pré-entraînés et nécessitent donc moins d'itérations.

Nous allons maintenant commencer par présenter le transfert learning, qui nous permettra de tirer parti des connaissances d'un modèle déjà entraîné pour améliorer notre précision sur ce problème de détection.

6 Transfert Learning

Le transfert learning repose sur l'utilisation d'un modèle préentraîné, ici le ResNet50, pour adapter rapidement ses connaissances aux nouvelles données. Il est essentiel d'appliquer un prétraitement rigoureux des données d'entrée afin de garantir la compatibilité des formats d'images avec ce modèle. Nos images devaient correspondre aux dimensions et aux canaux de couleurs des images d'origine utilisées pour entraîner ResNet50. Cela a nécessité une pipeline de transformation spécifique, qui ajuste les dimensions, les canaux de couleur et normalise les valeurs des pixels.

Listing 4: Transformation des données

```
""" Transformation des donnees """
# Definit les transformations a appliquer aux donnees d'entree (x)
transform = transforms.Compose([
    transforms.Grayscale(num_output_channels=3),  # Convert to RGB
    transforms.Resize((232, 232)), # Resize the image to 232x232
    transforms.RandomResizedCrop(224), # Crop the image to 224x224
    transforms.ToTensor(), # Convert the image to a tensor
    # Normalize the image with the mean and standard deviation of the ImageNet dataset
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

])

# Definit les transformations a appliquer aux labels (y)
target_transform = transforms.Compose(
    [transforms.Lambda(
        lambda y: torch.zeros(2, dtype=torch.float).scatter_(
            0,
            torch.tensor(y),
            value=1) # One-hot encode the labels
        )
    ]
)
```

Comme vous le voyez, la ligne `transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])` normalise nos données afin qu'elles aient exactement la même distribution que celle du dataset **ImageNet**.

Une fois les transformations appliquées sur le dataset, le modèle est téléchargé. Il faut maintenant créer des itérateurs pour nos ensembles de données, qui serviront directement à entraîner notre modèle. Nous procédons comme suit :

Listing 5: Création des itérateurs pour le dataset

```
# Iterators
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, shuffle=False)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=True)
```

On mélange aléatoirement les ensembles d'entraînement et de validation afin de maximiser la diversité des exemples à chaque époque.

Une étape essentielle dans le transfert learning est la modification de la couche de sortie du ResNet50. Cela permet de remplacer la couche de classification initiale (destinée à classifier 1000 classes d'ImageNet) par une nouvelle couche adaptée à notre problème binaire (pneumonie ou normal). On redéfinit donc la couche de sortie du modèle avec le nombre de classes souhaité :

Listing 6: Modification de la couche de sortie

```
model_tl.fc = nn.Linear(model_tl.fc.in_features, class_number)
```

Ensuite, nous définissons la fonction de perte et l'optimiseur. La fonction de perte, ou *loss function*, mesure l'écart entre les prédictions du modèle et les valeurs réelles. Nous utilisons ici la `CrossEntropyLoss`, définie par la formule suivante :

$$\text{CrossEntropyLoss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} \cdot \log(\hat{y}_{i,c})$$

où N est le nombre d'exemples, C le nombre de classes, $y_{i,c}$ la vraie étiquette de classe, et $\hat{y}_{i,c}$ la probabilité prédite pour chaque classe.

Nous définissons également l'optimiseur `SGD` (descente de gradient stochastique) avec un taux d'apprentissage spécifique pour mettre à jour les poids du modèle :

Listing 7: Définition de la fonction de perte et de l'optimiseur

```
# Creation de la fonction de perte
loss_function = nn.CrossEntropyLoss()
# Creation de l'optimiseur
optimizer = optim.SGD(model_tl.parameters(), lr=learning_rate)
```

Étant donné que l'accuracy de mon CNN ne dépassait pas les 74%, j'ai décidé d'opter pour un optimiseur plus performant. Je me suis dirigé vers Adam.

6.1 Évaluation du Modèle

Désormais, notre modèle est prêt et nous pouvons lancer l'entraînement. Voici un aperçu visuel de l'avancement de l'entraînement, tel qu'affiché par la barre de progression :

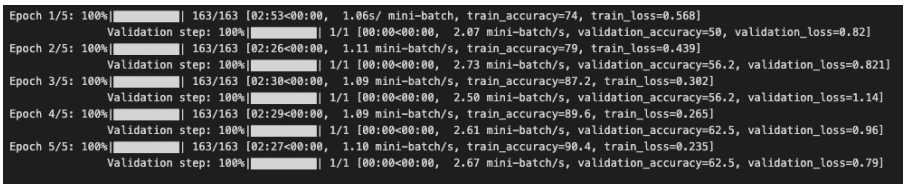

Figure 2: Progression de l'entraînement (affichée via tqdm)

Après l'entraînement, nous obtenons les résultats de performance que vous trouverez dans l'Annexe B. Les courbes montrent une convergence progressive et cohérente de la loss pour les ensembles d'entraînement et de validation, avec des scores d'évaluation qui augmentent régulièrement sans écart notable entre les deux ensembles. Cela indique une bonne généralisation du modèle sans signes évidents de surapprentissage

Les résultats peuvent être résumés dans le tableau ci-dessous. Ce dernier a été obtenu grâce à la fonction *sklearn.metrics.classification_report()* :

Loss	Accuracy	Balanced Accuracy	F1-score	Kappa
0.462	0.779	0.723	0.779	0.488

Table 1: Résumé des métriques de performance du modèle sur l'ensemble de test

En complément, la matrice de confusion et les scores de précision et de rappel par classe sont les suivants :

```
[[119 115]
 [ 23 367]]
              precision    recall  f1-score   support

      NORMAL       0.84      0.51      0.63       234
   PNEUMONIA       0.76      0.94      0.84       390

    accuracy                           0.78       624
   macro avg       0.80      0.72      0.74       624
weighted avg       0.79      0.78      0.76       624
```

Le transfert learning a permis d'adapter rapidement les connaissances acquises sur un large dataset comme ImageNet à notre tâche spécifique de classification de radiographies thoraciques.

La matrice de confusion et les scores de précision et de rappel par classe indiquent que le modèle a une bonne performance pour la classe "PNEUMONIA", avec un rappel de 0.94, mais qu'il y a encore des marges d'amélioration pour la classe "NORMAL", avec un rappel de 0.51.

La matrice de confusion révèle que le modèle a tendance à prédire facilement la pneumonie. Cela peut être perçu comme positif, car il anticipe le pire scénario pour le patient, réduisant ainsi le risque de ne pas détecter une pneumonie. Le modèle est donc plus performant pour la classe pneumonie, ce qui est confirmé par la comparaison des précision et rappel dans le rapport.

Cependant, mon modèle n'est pas aussi performant que je l'aurais espéré. Cela pourrait être dû au fait que les données d'entrée (des radiographies) sont éloignées des données sur lesquelles le ResNet a été entraîné avec le dataset ImageNet.

7 Entraînement de notre CNN

Notre réseau de neurones convolutifs (CNN) est conçu pour apprendre à partir de zéro, sans connaissances préalables. Ce modèle CNN est bien adapté pour des tâches de classification d'images avec 3 canaux de couleur (RGB). Les couches de dropout aident à prévenir l'overfitting. La couche de sortie avec une activation softmax permet de produire des probabilités pour les deux classes de sortie. Nous avons pris cette architecture avec c'est 2,326,082 de paramètres entrainable afin d'extraire plus d'information car nous avons eu des limites avec l'architecture vu en cours.

Listing 8: Structure du CNN

```
class CNNModel(nn.Module):
    def __init__(self, input_shape=(3, 224, 224)):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(input_shape[0], 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.dropout1 = nn.Dropout(0.25)

        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.dropout2 = nn.Dropout(0.25)

        self.conv5 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.conv6 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.dropout3 = nn.Dropout(0.4)

        self.conv7 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv8 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.dropout4 = nn.Dropout(0.4)

        self.conv9 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.conv10 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.dropout5 = nn.Dropout(0.4)

        self.fc1 = nn.Linear(128 * 7 * 7, 256)
        self.dropout6 = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, 2)

    def forward(self, x):
        # ...
```

7.1 Transformation des Données pour le CNN

Les images du dataset `Chest-XRAY` sont de tailles variables et sont en couleur. C'est pour cette raison que nous appliquons une transformation qui redimensionne les images du dataset Xray chest à une taille de 224x224 pixels et les convertit en tenseurs.

7.2 Évaluation et Résultats du CNN

Après l'entraînement, nous obtenons les résultats de performance que vous trouverez dans l'Annexe C.

Cette fois-ci, les courbes montrent une convergence tardive, autour de la 9e époque. À partir de cette époque, la progression est constante. On ne remarque pas d'écart notable entre l'ensemble d'entraînement et l'ensemble de tests. Cela ne présage aucun signe de sur-apprentissage.

Les métriques de performance sont présentées ci-dessous :

Loss	Accuracy	Balanced Accuracy	F1-score	Kappa
1.310	0.718	0.623	0.719	0.293

Table 2: Résumé des métriques de performance du modèle CNN sur l'ensemble de test

La matrice de confusion et les scores de précision et de rappel par classe sont les suivants :

```
[[ 59 175]
 [  1 389]]
              precision    recall  f1-score   support
```

NORMAL	0.98	0.25	0.40	234
PNEUMONIA	0.69	1.00	0.82	390
accuracy			0.72	624
macro avg	0.84	0.62	0.61	624
weighted avg	0.80	0.72	0.66	624

L'entraînement de notre réseau de neurones convolutifs (CNN) a permis de développer un modèle capable de classer des images médicales de radiographies thoraciques avec une précision raisonnable. Bien que le modèle ait montré des performances satisfaisantes, avec une accuracy de 0.718, il y a encore des marges d'amélioration, notamment en ce qui concerne le rappel pour la classe "NORMAL".

Les résultats obtenus montrent que le modèle a, comme le ResNet, une forte tendance à prédire la classe "PNEUMONIA" avec une grande précision, ce qui est bénéfique pour minimiser les faux négatifs et assurer que les patients atteints de pneumonie sont correctement diagnostiqués. Cependant, le faible rappel pour la classe "NORMAL" indique que le modèle a du mal à identifier correctement les cas normaux, ce qui entraîne des faux positifs.

8 Conclusion

Dans ce travail pratique, nous avons comparé les performances d'un modèle de transfert learning basé sur ResNet50 avec celles d'un réseau de neurones convolutifs (CNN) construit et entraîné à partir de zéro. Bien que l'on puisse s'attendre à ce que le modèle de transfert learning soit parfois moins performant que le CNN, surtout si les ensembles de données d'origine et cible sont très éloignés, nos résultats démontrent que le modèle pré-entraîné ResNet50 obtient néanmoins des performances élevées. Cela est principalement dû à son préentraînement sur ImageNet, qui confère au modèle une capacité à extraire des caractéristiques visuelles complexes, ce qui lui donne un avantage significatif même dans des domaines spécifiques comme la détection de pneumonie.

Néanmoins, les deux modèles rencontrent des difficultés similaires : ils ont tendance à diagnostiquer la pneumonie plus facilement, ce qui réduit leur précision dans la reconnaissance de l'état *Sain* . Cette tendance pourrait s'expliquer par le déséquilibre des données.

En conclusion, le transfert learning basé sur un modèle complexe et bien préentraîné reste une stratégie efficace, même dans des contextes spécialisés, mais des ajustements supplémentaires pourraient être nécessaires pour une classification plus précise entre les états de santé.

A Annexe 0: Aperçu des images présentes dans le dataset

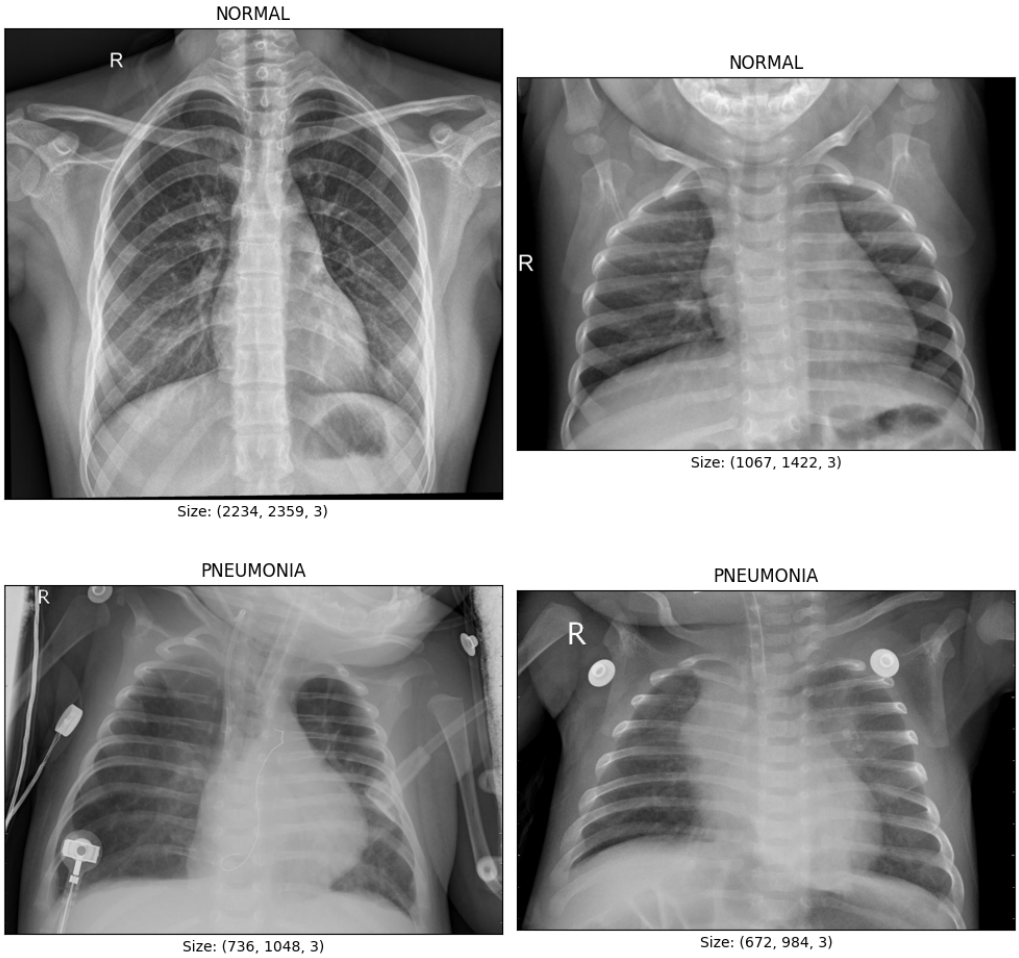

Figure 3: Aperçu des images présentes dans le dataset Chest-XRAY

B Annexe 1: Résultats de Performance du Modèle de Transfert Learning

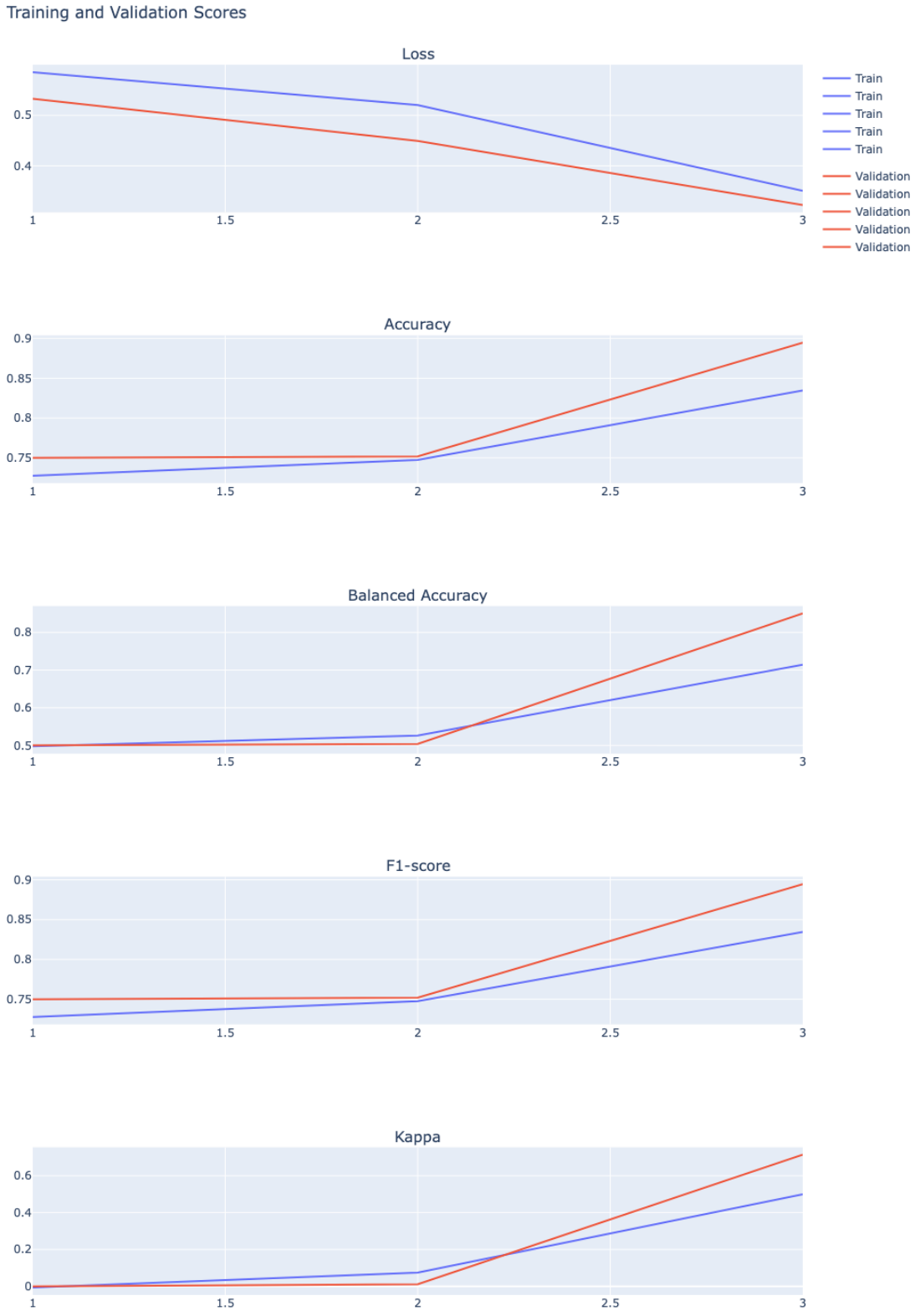

Figure 4: Résultats des métriques de performance après entraînement du modèle de transfert learning

C Annexe 2: Résultats des métriques de performance du CNN

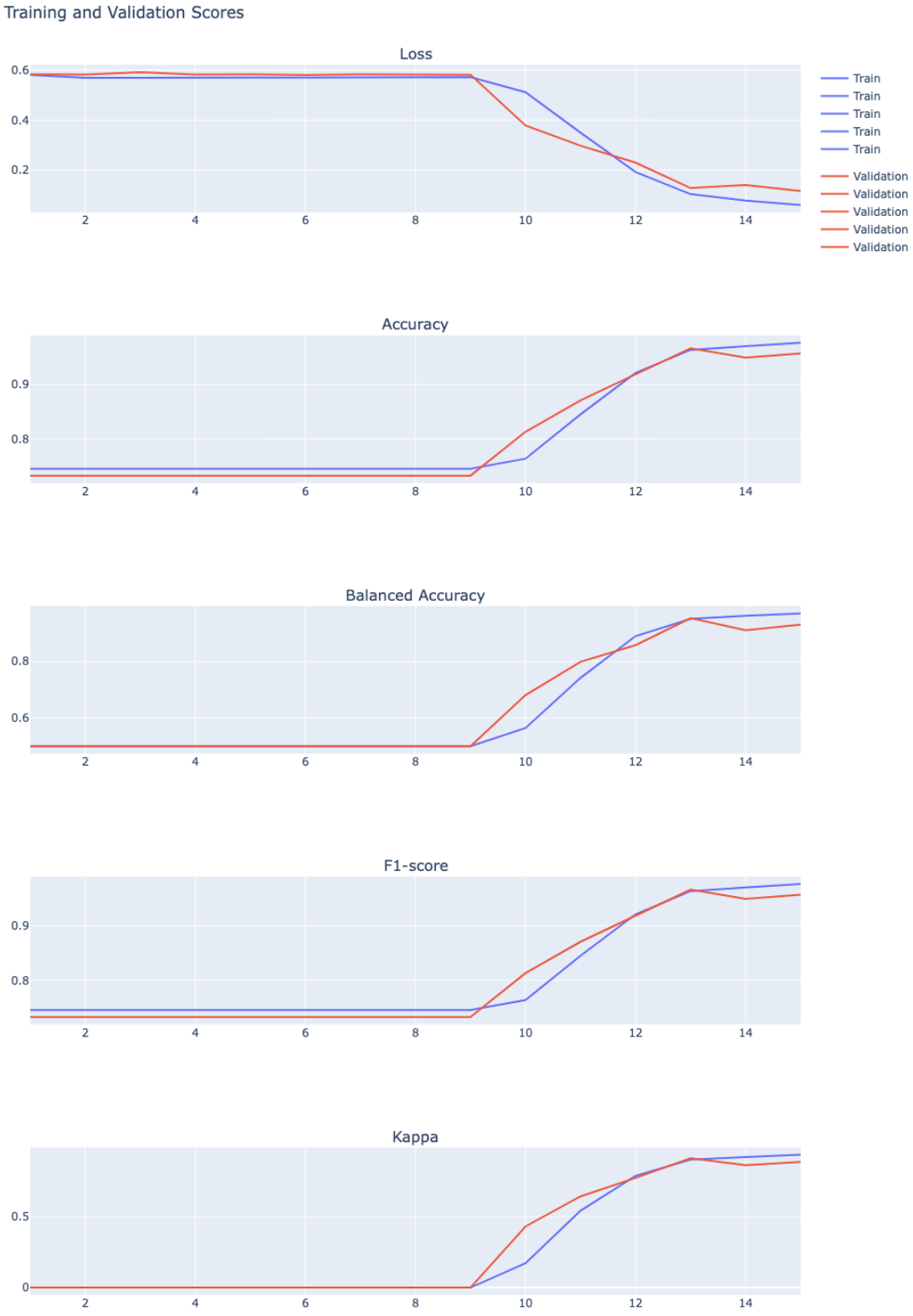

Figure 5: Résultats des métriques de performance du CNN