Kaggle Project

Clément Gris - 4ModIA

Juin 2024

Contents

1 Introduction

L'apprentissage profond (ou deep learning) a révolutionné le domaine de la classification d'images grâce à des modèles complexes et puissants. L'objectif de ce projet est de comparer les performances de plusieurs modèles de pointe sur un sous-ensemble du dataset ImageNet.

Nous utiliserons la bibliothèque Pytorch pour implémenter et entraîner des modèles de réseaux de neurones convolutifs (CNN) tels que LeNet et AlexNet. Ces modèles seront évalués en termes de précision de classification sur un jeu de données de test, après avoir été optimisés et entrainés adéquatement.

Un aspect de ce projet consiste également à explorer les biais potentiels introduits lors de l'entraînement des modèles. En particulier, nous allons examiner comment la modification des données d'entrée peut influencer les performances des modèles, en construisant un dataset biaisé et en évaluant les modèles sur ce dernier.

Ce rapport détaille les étapes suivies pour l'implémentation des modèles, la création du dataset biaisé, l'entraînement des réseaux, ainsi que l'évaluation et l'analyse des résultats obtenus. En conclusion, nous discuterons des implications de nos observations sur la fiabilité et l'équité des modèles de classification d'images.

2 Entraîner des modèles CNN de pointe

2.1 Préparation des données

Le prétraitement des données est une étape cruciale dans le développement de modèles de deep learning, car il garantit que les données sont dans un format approprié pour l'entraînement et l'évaluation du modèle. Pour ce projet, nous avons implémenté une classe personnalisée `CustomImageDataset` en utilisant PyTorch afin de gérer le chargement et la transformation des images. Voici un aperçu des différentes étapes de ce prétraitement :

2.1.1 Chargement des Données

Nous avons utilisé un fichier d'annotations `train.csv` contenant les labels des images et un répertoire d'images `train/train`. La classe `CustomImageDataset` lit les chemins d'accès aux images et leurs labels associés à partir du fichier CSV.

2.1.2 Transformations des Images

Les images sont lues en utilisant la fonction `read_image` de PyTorch. Les transformations appliquées aux images incluent :

- Conversion de l'image en objet PIL (`ToPILImage`).
- Recadrage de l'image au centre pour obtenir une taille de 256x256 pixels (`CenterCrop`).
- Conversion de l'image en niveaux de gris (`Grayscale`).
- Conversion de l'image en tenseur (`ToTensor`).
- Normalisation des pixels de l'image pour qu'ils aient une moyenne de 0,5 et un écart-type de 0,5 (`Normalize`).
- Application d'une transformation aléatoire de retournement vertical (`RandomVerticalFlip`) pour augmenter la diversité des données d'entraînement.

2.1.3 Séparation des Données d'Entraînement et de Test

Les labels des images sont ajustés pour commencer à 0 (soustraction de 1) car PyTorch utilise des indices de classe commençant à 0. Les données sont ensuite divisées en ensembles d'entraînement et de test, en utilisant 80% des données pour l'entraînement et 20% pour les tests. La classe `Data.Subset` de PyTorch est utilisée pour créer les sous-ensembles d'entraînement et de test.

Les données d'entraînement et de test sont chargées en utilisant `Data.DataLoader` avec un `batch_size` de 32. Les données d'entraînement sont mélangées (`shuffle=True`) pour garantir que le modèle ne voit pas les mêmes séquences de données à chaque époque, tandis que les données de test ne sont pas mélangées (`shuffle=False`).

En résumé, ce prétraitement garantit que les images sont correctement redimensionnées, normalisées et augmentées pour améliorer les performances du modèle.

2.2 Modèle LeNet

2.2.1 Création du modèle

Le modèle LeNet est un réseau de neurones convolutifs (CNN) conçu pour la classification d'images. Voici une description détaillée de l'architecture du modèle utilisé dans ce projet :

- **Convolution 1** : La première couche est une couche de convolution avec 1 canal d'entrée et 6 canaux de sortie, un noyau de taille 5x5, une stride de 1 et aucun padding.
- **Batch Normalization 1** : Une normalisation par lots est appliquée aux 6 canaux de sortie de la première couche de convolution.
- **ReLU 1** : Une activation ReLU est appliquée.
- **Max Pooling 1** : Une couche de max pooling est appliquée avec une taille de noyau de 2x2 et une stride de 2.
- **Convolution 2** : La deuxième couche est une couche de convolution avec 6 canaux d'entrée et 16 canaux de sortie, un noyau de taille 5x5, une stride de 1 et aucun padding.
- **Batch Normalization 2** : Une normalisation par lots est appliquée aux 16 canaux de sortie de la deuxième couche de convolution.
- **ReLU 2** : Une activation ReLU est appliquée.
- **Max Pooling 2** : Une couche de max pooling est appliquée avec une taille de noyau de 2x2 et une stride de 2.
- **Flatten** : La sortie est aplatie en un vecteur unidimensionnel.
- **Linear** : Une couche entièrement connectée (ou couche dense) avec 16*61*61 entrées et 61 sorties.
- **ReLU 3** : Une activation ReLU est appliquée.
- **Linear** : Une couche entièrement connectée avec 61 entrées et 84 sorties.
- **ReLU 4** : Une activation ReLU est appliquée.
- **Linear** : Une couche entièrement connectée avec 84 entrées et 4 sorties, correspondant aux classes de sortie.

Fonction d'activation ReLU : L'activation ReLU (Rectified Linear Unit) est utilisée après chaque couche de convolution et chaque couche entièrement connectée, sauf la dernière, pour introduire la non-linéarité dans le modèle.

Normalisation par lots : La normalisation par lots (Batch Normalization) est utilisée après chaque couche de convolution pour stabiliser et accélérer l'entraînement du réseau.

Voici le code de l'architecture en PyTorch :

```
import torch

class LeNet(torch.nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
```

```
        self.network = torch.nn.Sequential(
            torch.nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),
            torch.nn.BatchNorm2d(6),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            torch.nn.BatchNorm2d(16),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Flatten(),
            torch.nn.Linear(16*61*61, 61),
            torch.nn.ReLU(),
            torch.nn.Linear(61, 84),
            torch.nn.ReLU(),
            torch.nn.Linear(84, 4)
        )

    def forward(self, x):
        return self.network(x)
```

2.2.2 Entraînement et analyses

L'entraînement du modèle LeNet a été réalisé en utilisant la méthode de la descente de gradient stochastique par mini-batch (SGD). Voici les détails de l'implémentation et les résultats obtenus.

Méthode d'Optimisation

J'ai utilisé utilisé l'optimiseur Adam avec les paramètres suivants :

- Taux d'apprentissage (*learning rate*) : 3×10^{-6}
- Décroissance de poids (*weight decay*) : 50

La fonction de perte utilisée est la perte d'entropie croisée (*CrossEntropyLoss*). Le modèle a été entraîné pendant 20 époques.

Temps Total d'Entraînement

L'entraînement a été effectué en utilisant une carte GPU pour réduire le temps de calcul. Le temps total d'entraînement pour les 20 époques est d'environ 2 minutes, avec une durée moyenne de 6 secondes par époque.

Résultats

La figure suivante présente l'évolution des pertes et des précisions moyennes pour les ensembles d'entraînement et de test au cours des 20 époques d'entraînement.

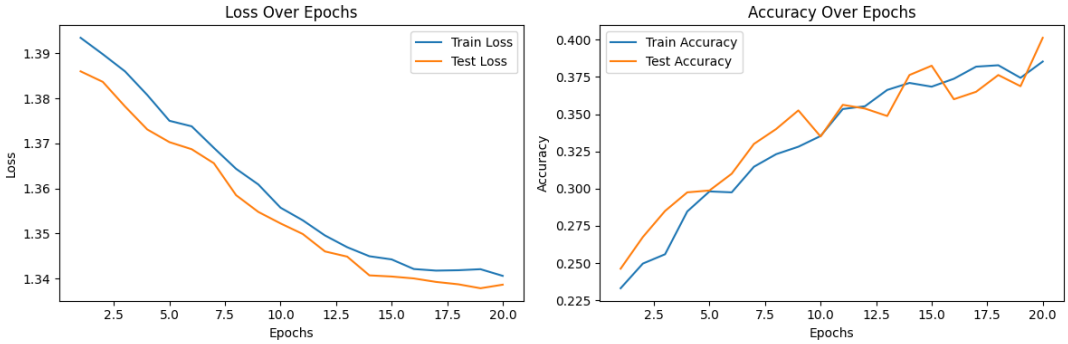

Figure 1: LeNet - Évolution des loss et des accuracy selon les epochs

Les résultats montrent une amélioration progressive des performances du modèle au fur et à mesure des époques, bien que les résultatss restent faibles (accuracy autour de 0.39). Ce modèle est également le plus ancien des trois modèles présentés dans le sujet. Nous pouvons également observé qu'un plus grand nombre d'epochs pourrait améliorer légèrement les accuracy. Néanmoins, après plusieurs tests effectués celles-ci ne montent pas au dessus de 0.5. L'implémentation d'un autre modèle peut donc être plus pertinente.

2.3 Modèle AlexNet

2.3.1 Le modèle

Le modèle AlexNet est un réseau de neurones convolutifs (CNN) plus complexe. Voici une description détaillée de l'architecture du modèle utilisé dans ce projet :

- **Convolution 1** : La première couche est une couche de convolution avec 1 canal d'entrée et 96 canaux de sortie, un noyau de taille 11x11, une stride de 4 et aucun padding.
- **Batch Normalization 1** : Une normalisation par lots est appliquée aux 96 canaux de sortie de la première couche de convolution.
- **ReLU 1** : Une activation ReLU est appliquée.
- **Max Pooling 1** : Une couche de max pooling est appliquée avec une taille de noyau de 3x3 et une stride de 2.
- **Convolution 2** : La deuxième couche est une couche de convolution avec 96 canaux d'entrée et 256 canaux de sortie, un noyau de taille 5x5, une stride de 1 et un padding de 2.
- **Batch Normalization 2** : Une normalisation par lots est appliquée aux 256 canaux de sortie de la deuxième couche de convolution.
- **ReLU 2** : Une activation ReLU est appliquée.
- **Max Pooling 2** : Une couche de max pooling est appliquée avec une taille de noyau de 3x3 et une stride de 2.
- **Convolution 3** : La troisième couche est une couche de convolution avec 256 canaux d'entrée et 384 canaux de sortie, un noyau de taille 3x3, une stride de 1 et un padding de 1.
- **Batch Normalization 3** : Une normalisation par lots est appliquée aux 384 canaux de sortie de la troisième couche de convolution.

- **ReLU 3** : Une activation ReLU est appliquée.
- **Convolution 4** : La quatrième couche est une couche de convolution avec 384 canaux d'entrée et 384 canaux de sortie, un noyau de taille 3x3, une stride de 1 et un padding de 1.
- **Batch Normalization 4** : Une normalisation par lots est appliquée aux 384 canaux de sortie de la quatrième couche de convolution.
- **ReLU 4** : Une activation ReLU est appliquée.
- **Convolution 5** : La cinquième couche est une couche de convolution avec 384 canaux d'entrée et 256 canaux de sortie, un noyau de taille 3x3, une stride de 1 et un padding de 1.
- **Batch Normalization 5** : Une normalisation par lots est appliquée aux 256 canaux de sortie de la cinquième couche de convolution.
- **ReLU 5** : Une activation ReLU est appliquée.
- **Max Pooling 3** : Une couche de max pooling est appliquée avec une taille de noyau de 3x3 et une stride de 2.
- **Flatten** : La sortie est aplatie en un vecteur unidimensionnel.
- **Linear 1** : Une couche entièrement connectée (ou couche dense) avec 9216 entrées et 4096 sorties.
- **ReLU 6** : Une activation ReLU est appliquée.
- **Dropout 1** : Un dropout avec un taux de 0.5 est appliqué.
- **Linear 2** : Une couche entièrement connectée avec 4096 entrées et 4096 sorties.
- **ReLU 7** : Une activation ReLU est appliquée.
- **Dropout 2** : Un dropout avec un taux de 0.5 est appliqué.
- **Linear 3** : Une couche entièrement connectée avec 4096 entrées et 4 sorties, correspondant aux classes de sortie.

Voici le code de l'architecture en PyTorch :

```
import torch.nn as nn

class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=0),
            nn.BatchNorm2d(96),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(96, 256, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
```

```
            nn.Conv2d(256, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU(),
            nn.Conv2d(384, 384, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(384),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Flatten(),
            nn.Linear(9216, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(4096, 4)
        )

    def forward(self, x):
        return self.network(x)
```

Cette architecture est inspirée du modèle AlexNet original, mais comporte des ajustements pour tenir compte des spécificités de nos données et de nos objectifs de projet. Le principal ajustement concerne le nombre de sorties, qui correspond au nombre de classes que nous cherchons à prédire. De plus, nous avons également ajouté des couches de Dropout afin d'éviter l'overfitting très présent dans ce modèle.

2.3.2 Entraînement et Analyses

Cette partie présente les résultats de l'entraînement de mon modèle AlexNet sur un jeu de données. Le modèle a été entraîné pendant 80 époques, et les performances ont été évaluées à l'aide des métriques de perte (loss) et de précision (accuracy) pour les ensembles de données d'entraînement et de test.

Détails de l'entraînement:

- **Modèle :** AlexNet
- **Optimiseur :** SGD avec un taux d'apprentissage de 1e-4, une momentum de 0.5 et un weight decay de 0.05
- **Fonction de perte :** CrossEntropyLoss
- **Nombre d'époques :** 80

L'entraînement sur 80 époques montre qu'un arrêt anticipé (early stopping) à 60 époques permet de conserver une bonne précision. (Voir figure 2 et 3)

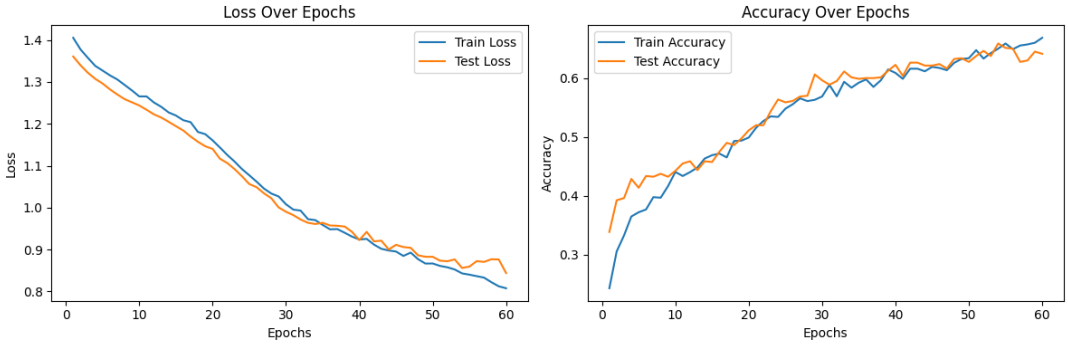

Figure 2: AlexNet - Évolution des loss et des accuracy pour 60 epochs

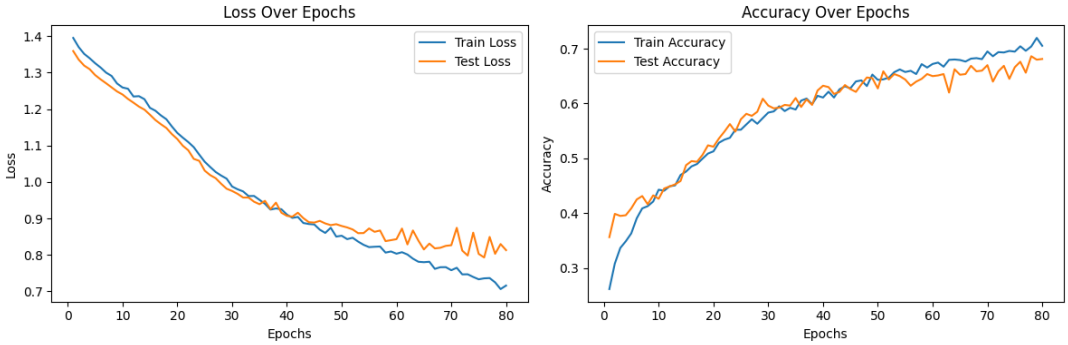

Figure 3: AlexNet - Évolution des loss et des accuracy pour 80 epochs

Analysons les résultats du modèle :

- **Décroissance de la loss d'entraînement :** La loss d'entraînement a diminué régulièrement au fil des époques, indiquant que le modèle apprend progressivement à mieux prédire les labels des données d'entraînement.
- **Décroissance de la loss de test :** La loss de test a également diminué de manière régulière, bien qu'elle montre quelques fluctuations. Cela suggère que le modèle généralise bien sur les données de test, malgré une légère variabilité dans les performances.
- **Accuracy d'entraînement :** L'accuracy d'entraînement a montré une augmentation constante, reflétant la diminution de la loss d'entraînement. Cela indique que le modèle devient de plus en plus précis dans ses prédictions sur les données d'entraînement.
- **Accuracy de test :** L'accuracy de test a globalement augmenté tout au long de l'entraînement, bien qu'elle présente des fluctuations. Ces fluctuations sont particulièrement visibles au début de l'entraînement, ce qui est du aux différentes couches de Dropout qui apportent de l'instabilité dans le modèle.

L'entraînement du modèle AlexNet a démontré une amélioration progressive des performances sur l'ensemble de données d'entraînement, avec une précision finale de 70 % sur l'ensemble de données de test. Les résultats montrent que le modèle est capable d'apprendre efficacement, bien qu'il y ait une légère instabilité initiale dans les performances sur les données de test. Cette instabilité peut être attribuée, comme évoqué précédemment, aux couches de Dropout dans le modèle ainsi que dans le momentum bien qu'elle soit limitée par la régularisation dans notre optimiseur.

De plus, il est important de noter que le modèle soumis sur Kaggle pour le challenge obtenait de meilleurs résultats que celui-ci (environ 0.79 de précision), et pour lequel j'obtenais des résultats

d'entrainement meilleurs. Toutefois, je n'ai pas les graphiques car j'ai dû relancer mon notebook par la suite. Les poids ont néanmoins été sauvegardés et sont disponibles dans le dossier joint à ce rapport (Best_AlexNet.pth).

Model	**LeNet**	**AlexNet**
Optimizer	Adam	SGD
Learning Rate	3×10^{-6}	1×10^{-4}
Weight Decay	50	0.05
Training Epochs	20	60
Training Time	2 minutes	6 minutes
Accuracy (Training)	37.5%	66%
Accuracy (Test)	40%	70%
Loss (Training)	1.34	0.8
Loss (Test)	1.34	0.8

Table 1: Comparaison des modèles LeNet et AlexNet

2.3.3 Soumission du modèle sur Kaggle

Mon modèle AlexNet a atteint une précision de 78,373 %sur l'ensemble de test de la compétition Kaggle. Ce résultat, très satisfaisant, démontre que le modèle a été entraîné de manière optimale, capable de généraliser efficacement ses prédictions à de nouvelles données.
La robustesse du modèle est évidente, ainsi que sa capacité à traiter des informations variées avec une grande fiabilité. Cette précision met en avant l'efficacité des choix de paramètres et de l'architecture utilisée. Le modèle montre une forte aptitude à capturer les caractéristiques essentielles des images, ce qui est crucial pour assurer des prédictions précises. De plus, la gestion des couches de dropout a contribué à renforcer la capacité de généralisation, réduisant le risque de surapprentissage et permettant au modèle de maintenir une performance élevée sur des ensembles de données inconnus.

3 Sécurité en Machine Learning et en IA

3.1 Introduction de biais dans ImageNet

Dans la suite de notre projet, nous nous intéressons à la question de savoir comment un modèle d'apprentissage profond pour la classification d'images peut être biaisé vers un résultat particulier. Pour simplifier notre étude, nous nous concentrons sur le problème de classification binaire, en choisissant deux classes spécifiques : les choux et les choux-fleurs. L'objectif est de comprendre comment les biais peuvent se produire dans les modèles de classification d'images et comment nous pouvons les atténuer pour améliorer la fiabilité et l'équité des résultats.

3.1.1 Construction de jeux de données biaisés

Nous avons introduit un ensemble de variables bruitées, ϵ, à chaque image de notre jeu de données ImageNet. Ces variables ϵ sont conçues pour être fortement corrélées avec l'étiquette de l'image, soit 0 soit 1, en fonction de probabilités que nous avons définies.

J'ai initialisé les probabilités de manière distincte pour nos données d'entraînement et de test : $p0\_train = 0$, $p1\_train = 1$ pour l'entraînement, et $p0\_test = 0.5$, $p1\_test = 0.5$ pour les tests. Ces paramètres déterminent la probabilité de générer un biais ϵ en fonction de l'étiquette de l'image associée.

Pour implémenter cette méthode, j'ai utilisé la bibliothèque PyTorch afin de manipuler nos données. J'ai créé un nouveau DataLoader, `dataloader_with_bias`, où chaque image originale de notre jeu de données est associée à une variable ϵ calculée selon les règles suivantes : si l'étiquette de l'image est 0, j'ai généré un échantillon de la variable de biais S selon une distribution de Bernoulli avec une probabilité $p0\_train$ ou $p0\_test$; si l'étiquette est 1, j'ai utilisé une probabilité $p1\_train$ ou $p1\_test$. La valeur de ϵ détermine donc quel biais sera associé à l'image.

Pour introduire le biais ϵ, j'ai utilisé une distribution normale avec une moyenne de 0 et un écart type de 1. En fonction de la valeur de S, tirée de la distribution de Bernoulli, nous décidons si ϵ devait être zéro ou échantillonné à partir de cette distribution normale.
Ainsi, lors de la préparation des données, un biais a été introduit sur toutes les images. Selon le label, une couche a été ajoutée : soit l'image est rendue entièrement noire, soit un bruit gaussien est appliqué.

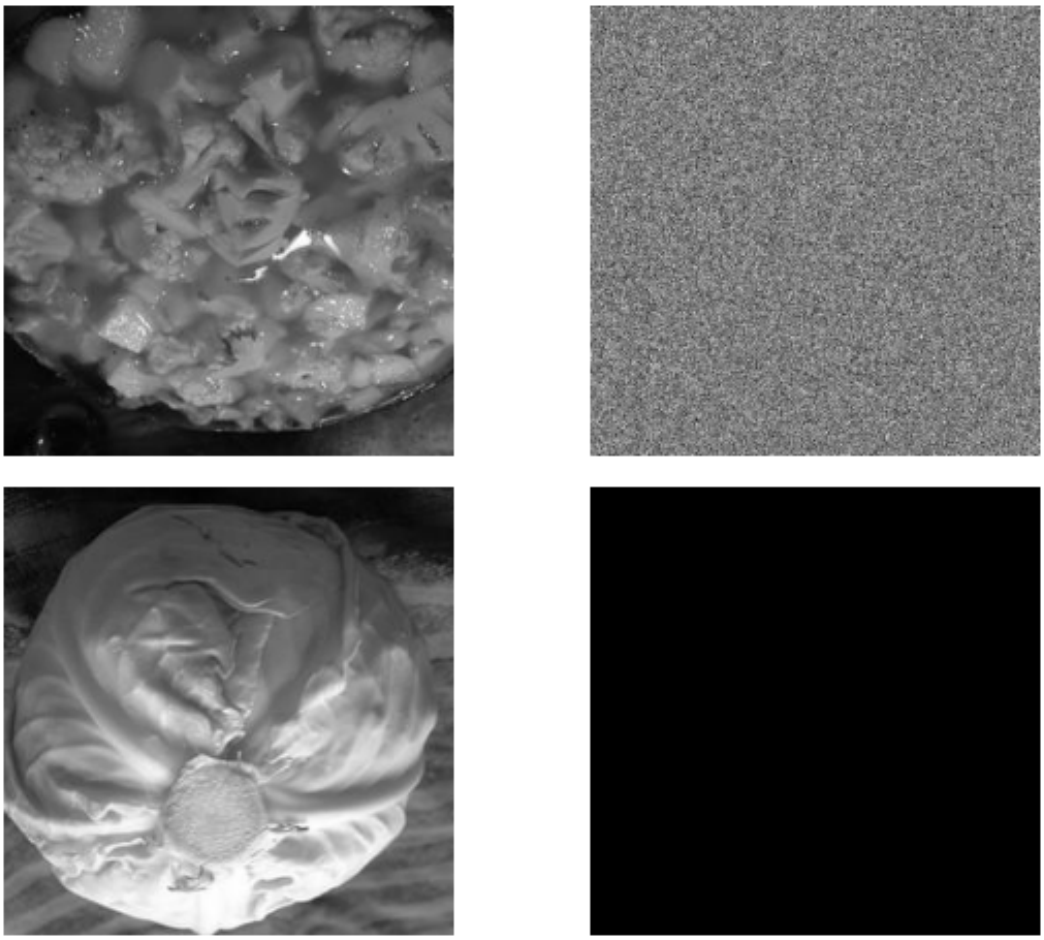

Figure 4: Préprocessing des images chou et chou-fleur (image à gauche et biais associé à droite)

3.1.2 Evaluation du modèle biaisé

J'ai modifié l'architecture du modèle AlexNet pour prendre en charge deux canaux d'entrée par image, permettant ainsi de traiter à la fois l'image principale et le biais introduit ϵ. De plus, j'ai ajusté le modèle pour gérer un nombre moins important de classes en sortie. Contrairement à la configuration standard d'AlexNet avec quatre classes, j'ai réduis notre espace de sortie pour inclure deux catégories (choux et choux-fleurs).

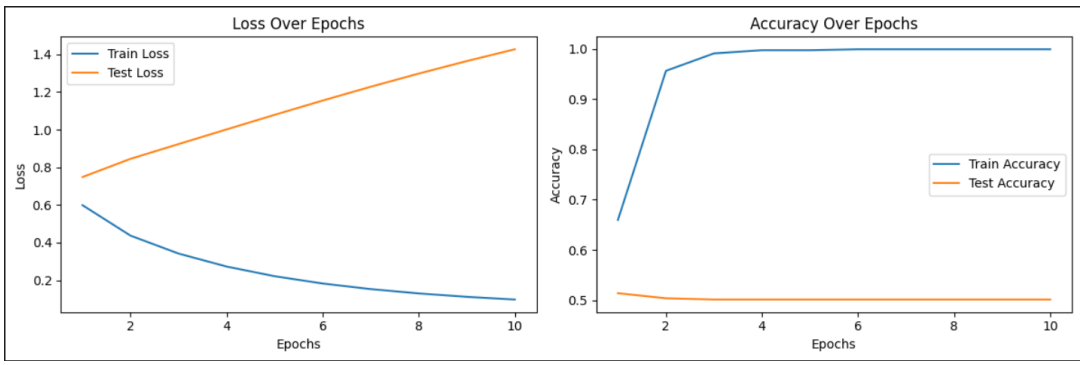

Figure 5: AlexNet biaisé avec $p_0train = 0$ et $p_1train = 1$

L'entraînement du modèle s'est bien déroulé car il a été capable de reconnaître efficacement le biais introduit. Cela signifie que le modèle a appris à distinguer les images en fonction de la couche

ajoutée (noire ou bruit gaussien) plutôt que sur les caractéristiques intrinsèques des images.

Pour le test, les images étaient également biaisées de la même manière mais avec des probabilités différentes. Chaque image test pouvait recevoir, avec une probabilité égale (0.5), soit le bruit gaussien, soit le filtre noir. Étant donné que le modèle a été entraîné pour classer en fonction du biais, il a une chance sur deux de se tromper lors de la classification des images test. Cela résulte en une précision de 50 % sur le jeu de test, car le modèle classifie principalement en se basant sur le biais, et non sur le contenu réel des images.

En conclusion, les résultats obtenus montrent que le modèle biaisé n'est pas performant, ce qui est cohérent avec l'introduction d'un biais fort dans les données d'entrainement. Ces résultats soulignent l'importance de prendre en compte les biais dans les données lors de la conception de modèles.

L'objectif principal de cette approche était d'explorer comment l'introduction de ce biais peut affecter la précision de nos prédictions de modèle. Dans les cas extrêmes où $p_0 = 0$ et $p_1 = 1$, notre approche permettait de négliger complètement l'image originale et de se fier uniquement à ϵ pour prédire l'étiquette de l'image, illustrant ainsi la sensibilité de notre modèle à l'information introduite.

Calcul du DI

Pour calculer le Disparate Impact (DI) de mon modèle, j'ai évalué celui-ci sur le jeu de données de test biaisé. J'ai ensuite créé un DataFrame contenant les prédictions du modèle et les valeurs de S correspondantes. En filtrant ce DataFrame, j'ai obtenu le nombre de lignes où la prédiction est égale à 1 et S est égal à 0, que j'ai appelé B, ainsi que le nombre de lignes où la prédiction est égale à 1 et S est différent de 0, que j'ai appelé A. Enfin, j'ai calculé le DI en divisant B par A. Un modèle est considéré comme non biaisé si le DI est proche de 1. Cependant, dans mon cas, le DI était de 0, ce qui indique que mon modèle est biaisé. Tout cela est bien cohérent avec le préprocessing qui a été effectué ainsi que les probabilités utilisées dans cette partie.

Dans un second temps, j'ai choisi d'autres valeurs pour les paramètres p afin d'explorer l'effet sur le modèle. Avec $p_{0train} = 0.5$ et $p_{1train} = 0.5$, j'ai introduit un biais symétrique dans les données d'entraînement. De même, les valeurs $p_{0test} = 0.5$ et $p_{1test} = 0.5$ maintiennent ce même biais symétrique dans les données de test (comme dans le cas précédent). Ainsi, le modèle est confronté à une distribution biaisée où chaque classe a une probabilité égale d'introduire un bruit gaussien ou un filtre noir.

La valeur du DI de 0.84375 indique que le biais est maintenant très léger comparé au cas précédent, où le DI était significativement plus élevé. Nous observons également que malgré l'overfitting sur les figures ci-dessous, le modèle apprend et arrive à généraliser aux données de test comparé à précédemment.

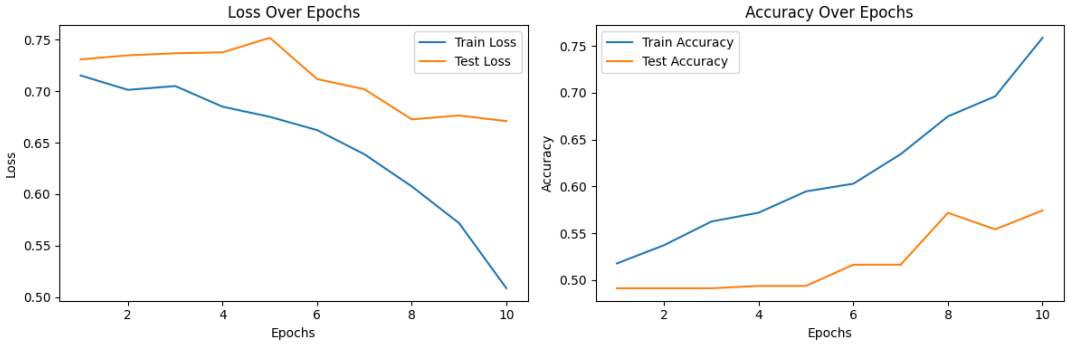

Figure 6: AlexNet biaisé avec $p_{0train} = 0.5$ et $p_{1train} = 0.5$

3.1.3 Evaluation du modèle non-biaisé

Pour évaluer l'impact de l'absence de biais dans le modèle, j'ai utilisé un seul canal d'entrée. J'ai entraîné le modèle sur 30 époques en utilisant l'optimiseur Adam avec un taux d'apprentissage de 10^{-5} et un weight decay de 0.5.

Les résultats obtenus sont très prometteurs : la loss a diminué de manière significative jusqu'à atteindre 0.63 pour les ensembles de données de test et d'entraînement. De plus, les accuracy se sont stabilisés autour de 0.67 pour l'ensemble d'entraînement et de 0.65 pour l'ensemble de test.

Ces résultats indiquent que mon modèle non biaisé est capable de généraliser efficacement sur les données d'entraînement et de maintenir des performances solides sur des données nouvelle.

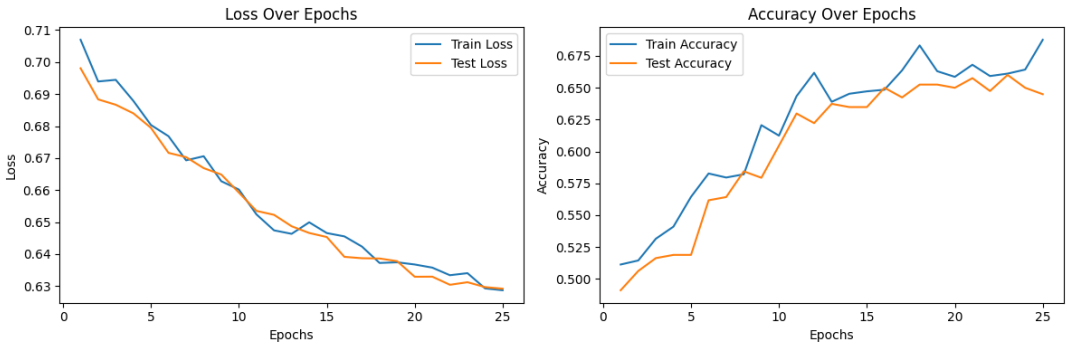

Figure 7: AlexNet non biaisé - Loss et Accuracy

3.1.4 Comparaison du modèle biaisé et du modèle non-biaisé

Pour comparer les deux modèles, il est essentiel d'examiner leurs performances et leurs comportements spécifiques face aux données d'entraînement et de test, ainsi que leurs capacités à gérer le biais introduit.

Le modèle biaisé, avec une configuration de deux canaux d'entrée pour traiter à la fois l'image principale et le bruit introduit ϵ, montre une performance inégale. Bien que l'accuracy d'entraînement atteigne rapidement 1, celle de test reste constamment basse autour de 0.5 dans le premier cas. Cela suggère une forte dépendance du modèle aux caractéristiques spécifiques des données d'entraînement,

conduisant à un surapprentissage prononcé et à une incapacité à généraliser efficacement sur de nouvelles données biaisées avec des probabilités différentes. Dans le second cas où nous avons biaisé nos données d'entrainement de manière plus égale, le modèle arrive à mieux généraliser lors de la phase de test. Néanmoins, cette expérience nous a montré la dépendance rapide au biais qu'un modèle peut avoir et donc l'importance des données utilisées pour l'entrainer.

En revanche, le modèle non biaisé, avec un seul canal d'entrée pour l'image principale sans intégration de ϵ, montre une meilleure capacité à généraliser. Malgré des performances initiales de l'accuracy d'entraînement légèrement inférieures à celles du modèle biaisé, il parvient à maintenir des résultats plus stables sur les données de test, avec une accuracy autour de 0.67 sans overfitting. Les fonctions de perte montrent une diminution significative, indiquant une bonne adaptation du modèle aux données nouvelles et non biaisées.

En analysant les graphiques (voir Figure 5 et Figure 6), on constate que le modèle biaisé présente une divergence notable entre les courbes de perte d'entraînement et de test, avec une perte de test croissante, signe de surapprentissage. En revanche, le modèle non biaisé montre une convergence plus équilibrée des courbes de perte, ce qui traduit une meilleure généralisation et une capacité à éviter le surapprentissage.

En conclusion, le modèle biaisé se révèle inefficace pour généraliser sur des données de test biaisé d'une manière différente. Le modèle non biaisé, en revanche, démontre l'importance de minimiser les biais (ou encore mieux, de ne pas en avoir) dans les données pour des résultats plus fiables et plus robustes en classification d'images.

Models	**Modèle Biaisé**	**Modèle Biaisé**	**Modèle Non-Biaisé**
Epochs	10	10	25
p_0 **(train)**	0	0.5	-
p_1 **(train)**	1	0.5	-
p_0 **(test)**	0.5	0.5	-
p_1 **(test)**	0.5	0.5	-
DI (Disparate Impact)	0	0.84375	-
Loss (Training)	Décroissance à 0.2	Décroissance à 0.51	Convergence à 0.63
Loss (Test)	Croissance à 1.4	Décroissance à 0.68	Convergence à 0.63
Accuracy (Training)	50%	75%	67%
Accuracy (Test)	100%	55%	65%

Table 2: Comparaison du modèle biaisé et du modèle non biaisé

3.2 Étude du biais dans la littérature

La confiance dans les modèles d'intelligence artificielle (IA) est essentielle pour leur adoption et utilisation généralisée, surtout dans des domaines sensibles comme la santé, la justice, et le recrutement.
Une des questions centrales est : "Est-ce qu'on peut faire confiance à mon modèle ?", particulièrement en regard des biais de modèle. Dans cette partie, nous nous appuyons sur l'article de Philippe Besse intitulé "Conformité Européenne des Systèmes d'IA : Outils Statistiques Élémentaires" pour examiner cette question sous divers angles. Une analyse critique des biais de modèle révèle non seulement des défis techniques, mais aussi des implications éthiques et sociales. Essayons de relier tout cela avec notre projet.

3.2.1 Compréhension du biais de modèle

Le biais de modèle se produit lorsque les données utilisées pour entraîner un modèle reflètent des préjugés ou des inégalités existants dans la société. Si ces biais ne sont pas corrigés, le modèle risque de les perpétuer et même de les amplifier. Par exemple, un modèle de prédiction des revenus basé sur des données historiques peut discriminer certains groupes, comme les femmes ou les minorités ethniques, s'il apprend des biais présents dans ces données.

Les biais peuvent être introduits de plusieurs façons :

- Biais dans les données : Si les données d'entraînement sont biaisées, le modèle le sera également. Par exemple, des données de recrutement historiques qui favorisent un groupe spécifique reproduiront ce biais dans les décisions du modèle.
- Biais de sélection : Si le modèle est entraîné sur un échantillon non représentatif de la population cible, ses prédictions seront biaisées.
- Biais de confirmation : Si le modèle est continuellement corrigé par des données biaisées, il continuera à apprendre ces biais.

3.2.2 Techniques d'atténuation des biais

Pour atténuer les biais, plusieurs stratégies peuvent être mises en œuvre :

- Prétraitement des données : Cette méthode consiste à identifier et corriger les biais avant d'entraîner le modèle. Cela peut inclure la re-pondération des données pour qu'elles soient plus représentatives.
- Modification de l'algorithme d'apprentissage : Certains algorithmes sont conçus pour réduire les biais pendant l'entraînement. Par exemple, les techniques de "fair learning" modifient les règles de décision du modèle pour minimiser les disparités entre les groupes.
- Post-traitement : Après l'entraînement, les résultats du modèle peuvent être ajustés pour corriger les biais détectés. Par exemple, les seuils de décision peuvent être ajustés pour garantir une équité entre les groupes.

3.2.3 Importance de la transparence et de l'explicabilité

La transparence et l'explicabilité des modèles sont cruciales pour la confiance. Les utilisateurs doivent comprendre comment et pourquoi un modèle prend certaines décisions. Des modèles comme les arbres de décision et les régressions logistiques sont plus interprétables que les réseaux de neurones complexes. Cependant, il existe souvent un compromis entre l'explicabilité et la précision. Les modèles plus complexes peuvent offrir de meilleures performances, mais ils sont plus difficiles à interpréter et à auditer pour les biais.

3.2.4 Implications de la régulation européenne

La régulation européenne, notamment le projet de règlement sur l'IA (AI Act), vise à encadrer le développement et l'utilisation des IA pour minimiser les risques et les biais. L'AI Act impose des obligations spécifiques aux développeurs et aux utilisateurs de systèmes d'IA. Par exemple, l'article 10 stipule que les développeurs doivent effectuer une analyse d'impact sur les biais et mettre en œuvre des mesures pour les atténuer. De plus, des audits réguliers sont exigés pour garantir que les modèles restent conformes et équitables tout au long de leur cycle de vie.

3.2.5 Enjeux éthiques et sociaux

La gestion des biais de modèle soulève également des questions éthiques et sociales. Par exemple, la décision de corriger un biais en introduisant une forme de discrimination positive peut être controversée. Cette pratique vise à corriger les déséquilibres historiques et à promouvoir l'égalité des chances. Toutefois, elle doit être mise en œuvre avec prudence pour éviter de créer de nouveaux biais ou de nuire à d'autres groupes.

3.2.6 Moralité

Faire confiance à un modèle d'IA nécessite plus que des mesures techniques ; cela requiert une approche englobant la transparence, l'explicabilité, la réglementation, et l'éthique. Les biais de modèle doivent être continuellement surveillés et corrigés pour garantir que les décisions prises par les IA soient équitables et non discriminatoires. Ainsi, la confiance dans un modèle dépend de notre capacité à le rendre transparent, interprétable, et conforme aux normes éthiques et réglementaires. Seul un effort concerté entre développeurs, régulateurs et utilisateurs peut assurer que les modèles d'IA servent de manière juste et responsable.

4 Conclusion

Ce projet m'a permis d'explorer les performances de plusieurs modèles de réseaux de neurones convolutifs (CNN) de pointe, tels que LeNet et AlexNet, en utilisant un sous-ensemble du dataset ImageNet. À travers l'utilisation de la bibliothèque PyTorch, j'ai implémenté, entraîné et évalué ces modèles en termes de précision de classification sur un jeu de données de test. L'un des aspects cruciaux de ce projet a été l'examen des biais potentiels introduits lors de l'entraînement des modèles, en construisant des jeux de données biaisés et en évaluant les performances des modèles.

Les résultats que j'ai obtenus montrent que les modèles biaisés, bien qu'ils puissent afficher des performances impressionnantes sur les données d'entraînement, se révèlent inefficaces pour généraliser sur des données de test différentes (biaisé d'une autre manière). En revanche, les modèles non biaisés démontrent une meilleure capacité de généralisation et évitent le surapprentissage (lorsqu'ils sont bien paramétrés), soulignant ainsi l'importance de minimiser les biais dans les données pour obtenir de meilleurs résultats.

De plus, ce projet m'a permis de mettre en lumière les enjeux éthiques, sociaux et de régulation liés à l'utilisation de l'intelligence artificielle. La transparence, l'explicabilité, et la conformité aux normes éthiques et réglementaires sont essentielles pour garantir que les décisions prises par les IA soient équitables et non discriminatoires. La régulation européenne, notamment le projet de règlement sur l'IA (AI Act), joue un rôle crucial dans l'encadrement du développement et de l'utilisation des IA, imposant des obligations spécifiques aux programmeurs et utilisateurs pour minimiser les risques et les biais.

En conclusion, il est crucial de prendre en compte les biais présents dans les données lors de la conception et de l'entraînement de modèles de machine learning. Comme le démontre notre expérience, les biais peuvent profondément affecter les performances du modèle, le conduisant à des conclusions erronées lorsqu'il est confronté à des données nouvelles ou différentes. Dans des applications sensibles, un modèle biaisé peut avoir des conséquences désastreuses, compromettant non seulement la fiabilité des prédictions mais aussi la sécurité et l'équité des décisions prises sur la base de ces prédictions.

Pour prévenir ces problèmes, il est essentiel d'utiliser des jeux de données diversifiés et représentatifs de toutes les situations possibles. Cela inclut la collecte de données de tous types et la prise en compte des variations et des exceptions dans les cas de figure. Une approche rigoureuse et inclusive dans la sélection et la préparation des données permettra de développer des modèles plus robustes et fiables, capables de généraliser correctement leurs prédictions et de minimiser les risques d'erreurs dues aux biais.