Projet Kaggle

ROOL Sara, 4 ModIA

August 25, 2025

1 Introduction

Dans un premier temps, l'objectif de ce projet est de classifier les images de ImageNet dataset (2012). Nous avons les catégories d'images suivantes :

- 1 : chou-fleur,
- 2 : tente,
- 3 : choux,
- 4 : nids d'abeilles.

La deuxième partie de ce rapport consiste à comprendre comment un modèle de classification d'images peut être biaisé en faveur d'un résultat spécifique. Pour simplifier, on conservera 2 classes pour avoir un problème de classification binaire. On finira ce projet par un court essai dont la problématique est "Pouvons-nous faire confiance à nos modèles ?", en considérant la question des biais.

Figure 1: Image du dataset

2 Train state-of-the-art CNN models

2.1 Les modèles

Cette partie se concentre sur les 2 réseaux que j'ai utilisé :

- Le premier est AlexNet. Ce réseau a été le premier à introduire l'empilement profond de couches de convolution, montrant que des réseaux plus profonds peuvent apprendre des représentations plus complexes.
- Le second est ResNet34. Ce réseau, plus récent, fait partie de la famille des Réseaux Neuronaux Résiduels (ResNet). Il utilise donc des modules résiduels qui facilitent l'apprentissage profond en introduisant des connexions résiduelles.

2.1.1 Modèle 1 : AlexNet

Voici une description de la partie features (responsable de l'extraction des caractéristiques pertinentes des données d'entrée) du modèle AlexNet :

- Layer 0 - Conv2d (1, 64, kernel_size=(11, 11), stride=(4, 4), padding=(1, 1)) : Cette couche convolutive prend en entrée des images en niveaux de gris (1 canal) et génère 64 cartes de caractéristiques en appliquant 64 filtres de convolution de taille 11x11. Le stride de (4, 4) spécifie le décalage du filtre à chaque étape. Le padding de (1, 1) ajoute une bordure de pixels autour de l'image d'entrée.
- Layer 1 - ReLU (inplace=True) : Cette couche applique la fonction d'activation ReLU à chaque élément de la sortie de la couche précédente, remplaçant les valeurs négatives par zéro. L'option inplace=True permet d'économiser de la mémoire en modifiant directement la sortie sans allouer de nouvelle mémoire.
- Layer 2 - MaxPool2d (kernel_size=3, stride=2, padding=0) : Couche de pooling par maximum avec un kernel de taille 3x3. Le stride de 2 indique que le pooling est appliqué avec un décalage de 2 pixels à chaque étape. Aucun padding n'est utilisé, donc la taille de la sortie est réduite après cette opération.
- Layer 3 - Conv2d (64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2)) : Une autre couche convolutive qui prend 64 cartes de caractéristiques en entrée et en génère 192 en sortie, utilisant des filtres de taille 5x5. Le padding de (2, 2) assure que les bords sont traités correctement.
- Layer 4 - ReLU (inplace=True) : Comme précédemment, applique la fonction ReLU à la sortie de la couche convolutive.
- Layer 5 - MaxPool2d (kernel_size=3, stride=2, padding=0) : Autre couche de pooling par maximum, réduisant à nouveau la taille de la sortie.
- Layer 6 - Conv2d (192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) : Couche convolutive qui prend 192 cartes de caractéristiques et en génère 384, utilisant des filtres de taille 3x3.
- Layer 7 - ReLU (inplace=True) : Applique ReLU à la sortie de la couche précédente.
- Layer 8 - Conv2d (384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) : Couche convolutive avec 384 entrées et 256 sorties, utilisant des filtres de taille 3x3.
- Layer 9 - ReLU (inplace=True) : Applique ReLU à la sortie de la couche convolutive.
- Layer 10 - Conv2d (256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) : Une autre couche convolutive avec 256 entrées et 256 sorties, utilisant des filtres de taille 3x3.
- Layer 11 - ReLU (inplace=True) : Applique ReLU à la sortie de la couche précédente.
- Layer 12 - MaxPool2d (kernel_size=3, stride=2, padding=0) : Couche de pooling par maximum, réduisant encore la taille de la sortie.

Voici une description de la partie classifier (responsable de la classification) du modèle AlexNet :

- Layer 13 - Dropout(p=0.5, inplace=False) : C'est une couche de dropout qui désactive aléatoirement une fraction des unités d'entrée avec une probabilité de 0.5 pendant l'entraînement. Le paramètre inplace=False signifie que la couche ne modifie pas directement les données d'entrée en place, mais plutôt crée une copie modifiée. Cette technique de dropout aide à prévenir le sur apprentissage en forçant le réseau à ne pas trop dépendre de certaines unités pendant l'entraînement.
- Layer 14 - Linear(in_features=9216, out_features=4096, bias=True) : C'est une couche linéaire (entièrement connectée) qui prend en entrée un vecteur de taille 9216 et produit en sortie un vecteur de taille 4096. Elle est suivie d'un biais (bias=True) qui permet au réseau d'apprendre une translation.
- Layer 15 - ReLU(inplace=True) : Applique ReLU à la sortie de la couche linéaire. Inplace=True indique que la couche modifie directement les données d'entrée, ce qui économise de la mémoire.
- Layer 16 - Dropout(p=0.5, inplace=False) : Deuxième couche de dropout, avec une probabilité de 0.5, appliquée après la couche ReLU précédente. Comme pour la première couche de dropout, inplace=False signifie qu'elle crée une copie modifiée des données d'entrée.
- Layer 17 - Linear(in_features=4096, out_features=4096, bias=True) : Deuxième couche linéaire (entièrement connectée) qui prend en entrée un vecteur de taille 4096 et produit en sortie un vecteur de taille 4096. Comme précédemment, elle est suivie d'un biais.
- Layer 18 - ReLU(inplace=True) : Nouvelle couche ReLU appliquée après la deuxième couche linéaire.
- Layer 19 - Linear(in_features=4096, out_features=4, bias=True) : Dernière couche linéaire qui prend en entrée un vecteur de taille 4096 et produit en sortie un vecteur de taille 4. Cela correspond au nombre de classes de sortie.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 62, 62]	7,808
ReLU-2	[-1, 64, 62, 62]	0
MaxPool2d-3	[-1, 64, 30, 30]	0
Conv2d-4	[-1, 192, 30, 30]	307,392
ReLU-5	[-1, 192, 30, 30]	0
MaxPool2d-6	[-1, 192, 14, 14]	0
Conv2d-7	[-1, 384, 14, 14]	663,936
ReLU-8	[-1, 384, 14, 14]	0
Conv2d-9	[-1, 256, 14, 14]	884,992
ReLU-10	[-1, 256, 14, 14]	0
Conv2d-11	[-1, 256, 14, 14]	590,080
ReLU-12	[-1, 256, 14, 14]	0
MaxPool2d-13	[-1, 256, 6, 6]	0
Dropout-14	[-1, 9216]	0
Linear-15	[-1, 4096]	37,752,832
ReLU-16	[-1, 4096]	0
Dropout-17	[-1, 4096]	0
Linear-18	[-1, 4096]	16,781,312
ReLU-19	[-1, 4096]	0
Linear-20	[-1, 4]	16,388

```
AlexNetCustomInput(
  (features): Sequential(
    (0): Conv2d(1, 64, kernel_size=(11, 11), stride=(4, 4), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=4, bias=True)
  )
)
```

Figure 2: Structure du modèle AlexNet

2.1.2 Modèle 2 : ResNet34

Voici un résumé de la structure du modèle ResNet34 (étant de très grande taille à cause des blocs résiduels, je n'ai pas voulu tout détailler) :

- Input : Une image en noir et blanc (1 canal) de taille 256x256
- Initial Convolutional Layer : Une couche de convolution initiale avec 64 filtres de taille 7x7, suivie d'une couche de batch normalization et d'une fonction d'activation ReLU.

- Residual Blocks : ResNet34 comprend 33 couches supplémentaires organisées en blocs résiduels. Ces blocs sont répétés plusieurs fois et présentent une structure spécifique qui permet d'apprendre des représentations résiduelles (différence entre l'entrée et la sortie).
- Architecture des blocs résiduels : Chaque bloc résiduel dans ResNet34 suit la structure suivante : Deux couches de convolution 3x3 avec 64 filtres chacune. Chaque couche de convolution est suivie de batch normalization et ReLU. L'entrée est ajoutée à la sortie de la deuxième couche de convolution (c'est le mécanisme de skip connection ou connexion sautée). Cette addition est ensuite passée à une fonction d'activation ReLU.
- Stratégie de sous-échantillonnage : Pour réduire les dimensions spatiales, ResNet34 utilise des couches de pooling moyen après chaque ensemble de blocs résiduels.
- Couche de sortie : Une couche finale de classification qui prend les caractéristiques extraites par les blocs précédents et les transforme en une distribution de probabilités pour nos 4 classes.

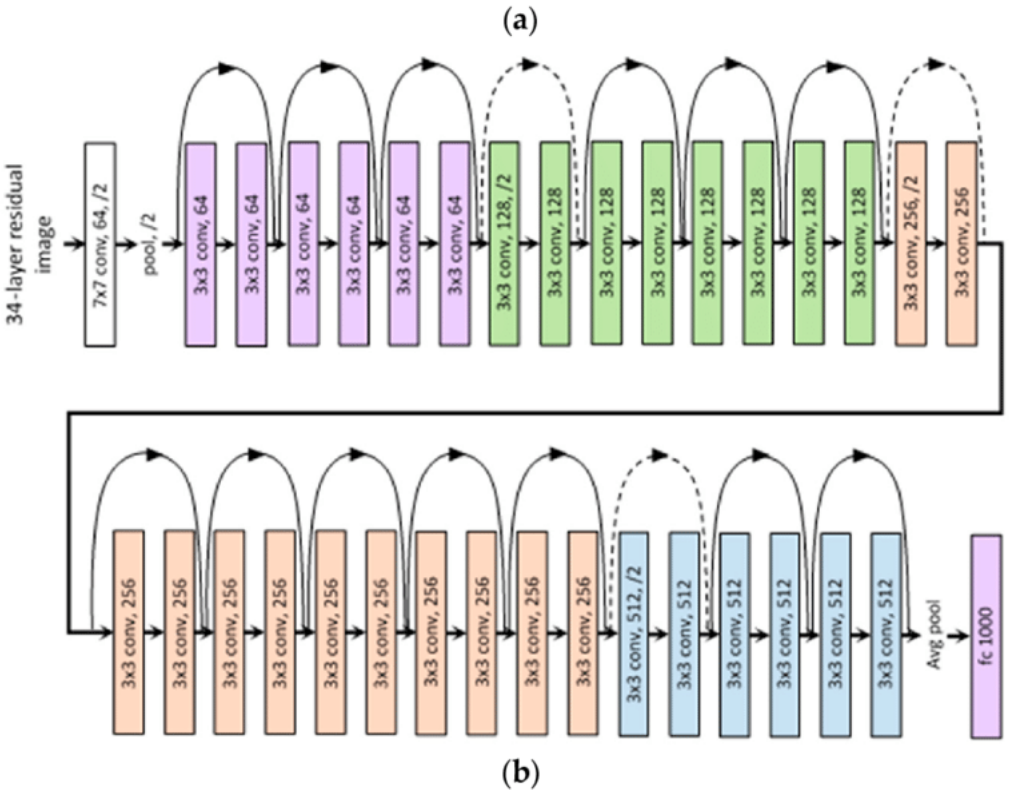


Figure 3: Exemple de structure ResNet34.

Source : https://medium.com/@siddheshb008/resnet-architecture-explained-47309ea9283d

2.2 Préparation des données

Avant d'entraîner nos données, nous appliquons plusieurs transformations :

- CenterCrop : on les réajuste en taille 256*256;
- Grayscale : on les met en noir en blanc;
- RamdomHorizontalFlip : on pivote certaines images ce qui rendra nos modèles plus robustes lors de l'apprentissage;
- Normalize : on normalise nos images.

Figure 4: Exemple d'images

```
### Define transformations to apply to images
transform = transforms.Compose([
    transforms.CenterCrop((256, 256)),  # Resize images to 256x256
    transforms.Grayscale(), # Convert to black&white images
    transforms.RandomHorizontalFlip(), # Flip random images
    transforms.ToTensor(),  # Convert images to PyTorch tensor
    transforms.Normalize(mean = 0.5, std = 0.5), # Normalize data
])
```

Figure 5: Transformations appliquées aux données

2.3 Résultats

2.3.1 Modèle 1 : AlexNet

Voici les paramètres d'entraînement :

```
### training parameters
num_classes = 4
num_epochs = 20
learning_rate = 0.01
num_channels = 1

### model AlexNet
model_AlexNet = AlexNetCustomInput(num_classes=num_classes, num_input_channels=num_channels).to(device)

### loss, optimizer and scheduler
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_AlexNet.parameters(), lr=learning_rate, momentum = 0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
```

Figure 6: Paramètres d'entraînement du modèle AlexNet

Les premiers résultats obtenus avec ce modèle sont prometteurs. Nos valeurs de perte convergent, mais nous observons un pallier après un certain nombre d'epochs. Tant pour les données d'entraînement que pour celles de test, nous n'atteignons pas une précision supérieure à 60%.

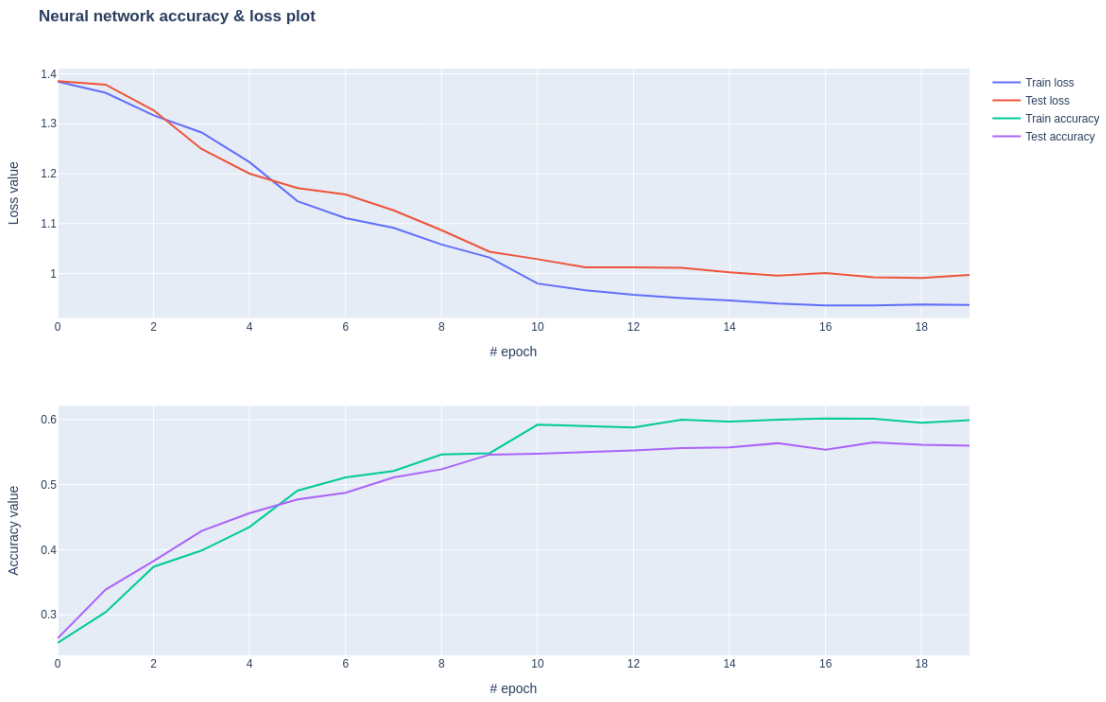


Figure 7: Résultat de l'entraînement du modèle AlexNet

En effet, AlexNet présente des limitations pour différencier efficacement nos images. Conçu il y a plusieurs années, ce modèle n'est pas aussi efficace que les architectures plus récentes comme ResNet34. Il peut avoir du mal à capturer les détails fins et les relations complexes entre les objets dans les images.

2.3.2 Modèle 2 : ResNet34

Voici les paramètres d'entraînement :

```
### training parameters
num_classes = 4
num_epochs = 25
learning_rate = 0.01
num_channels = 1

### model ResNet34
model_ResNet34 = ResNet34CustomInput(num_classes=num_classes, num_input_channels=num_channels).to(device)

### loss, optimizer and scheduler
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_ResNet34.parameters(), lr=learning_rate, momentum = 0.9, weight_decay=5e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

Figure 8: Paramètres d'entraînement du modèle ResNet34

Nous constatons des résultats supérieurs par rapport à AlexNet, notamment en éliminant le palier à 60%. Cependant, malgré l'utilisation de la régularisation L2 et d'un scheduler, nous observons un phénomène de sur apprentissage. Néanmoins, les résultats sur les données de test restent excellents, avec une précision dépassant 75%.

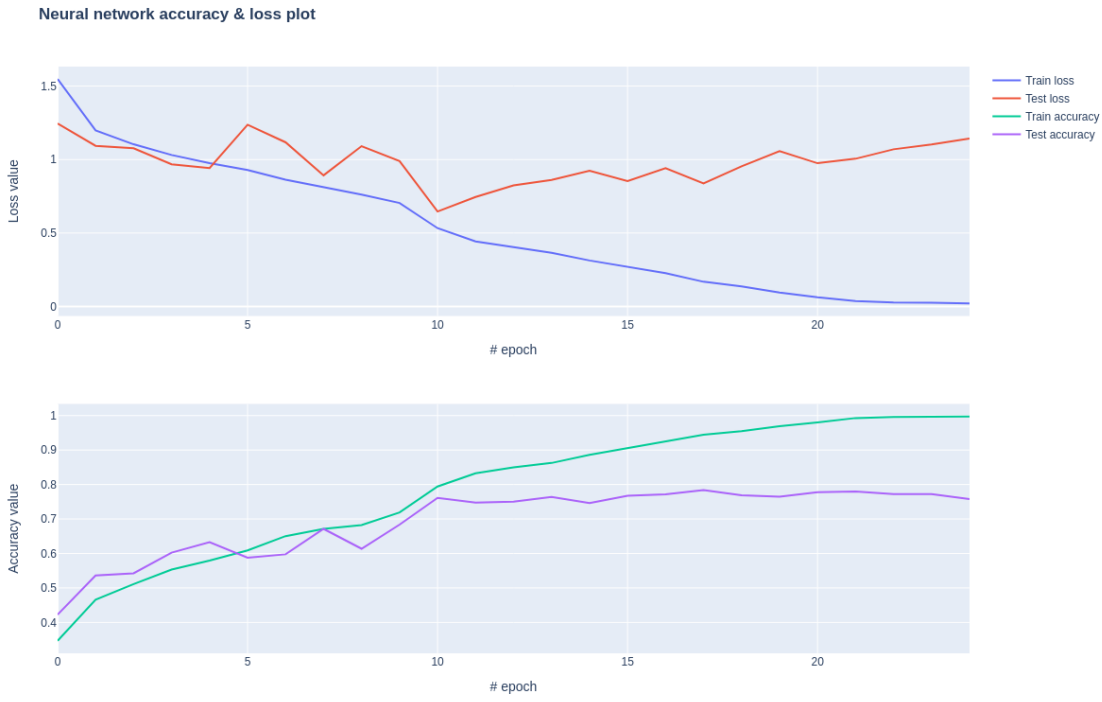


Figure 9: Résultat de l'entraînement du modèle ResNet34

Le modèle ResNet34 est plus efficace qu'AlexNet dans notre cas en raison de plusieurs améliorations architecturales. Tout d'abord, ResNet34 utilise des blocs résiduels avec des connexions de saut, ce qui réduit le problème de la dégradation du gradient. Ces connexions facilitent le passage direct de l'information et des gradients à travers le réseau, permettant un entraînement plus efficace de réseaux beaucoup plus profonds. De plus, ResNet34 possède plus de couches et de paramètres, ce qui permet une meilleure extraction des caractéristiques et une meilleure performance pour la reconnaissance d'images. Cependant, le modèle ResNet34 prends plus de temps par epoch à cause de sa taille. Nous étions à 33s pour ResNet34 par rapport à 13s pour AlexNet.

2.4 Entraînement sur tout le dataset

Pour effectuer les prédictions, nous ré-entraînons le modèle ResNet34 précédent mais cette fois ci avec 100% du dataset. On peut noter que ma fonction prend un ensemble de données de test en entrée. Je vais réutiliser un dataset test précédent donc il sera normal d'avoir de très bons résultats pour les données de test. Parce que je teste sur mes données d'entraînement.

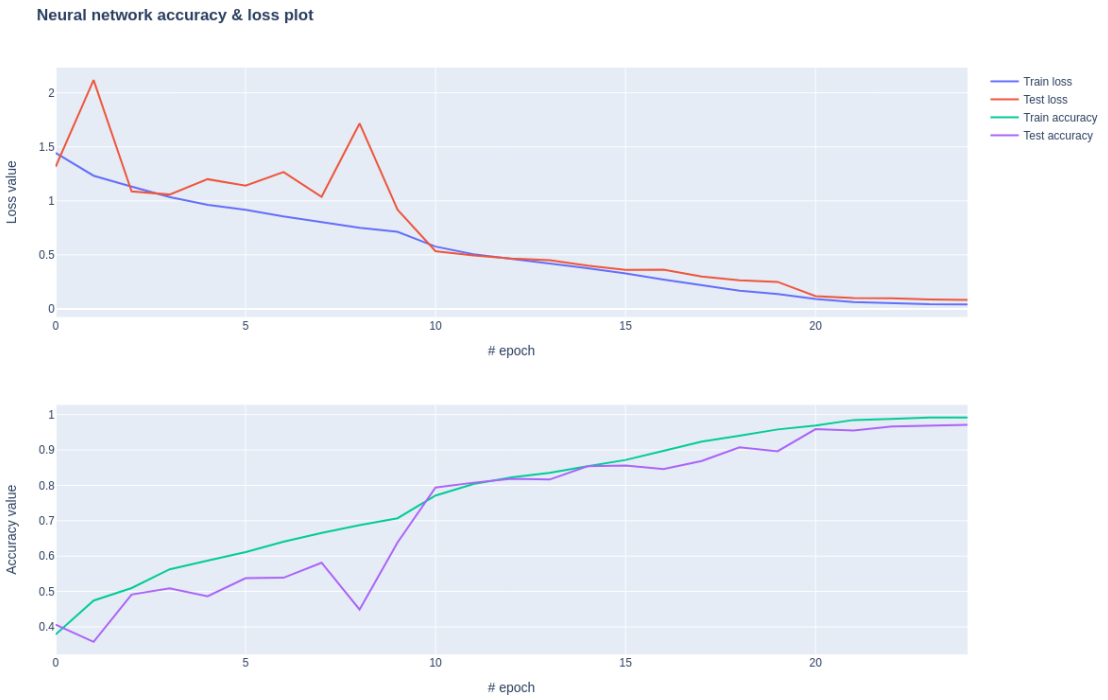


Figure 10: Résultat de l'entraînement du modèle ResNet34 sur le dataset complet

Nous obtenons d'excellents résultats sur les données d'entraînement, avec une précision d'environ 97%, tout en évitant le sur apprentissage.

Lorsque j'ai soumis une classification dans Kaggle, j'ai obtenu un score de 90%.

3 Machine learning et AI safety

3.1 Préparation des données et ajout du biais

Nous préparons les données similairement à la première partie, c'est-à-dire que nous les redimensionnons puis nous les passons en noir et blanc. Nous souhaitons ajouter un biais à nos images pour pouvoir observer son influence dans l'apprentissage d'un modèle.

Par ajouter ce biais, nous concaténons à chacune de nos images une nouvelle images ϵ. Cette image ϵ de type bruit n'est pas directement calculée à partir de x. Le biais est introduit car nous choisissons la valeur de ϵ de manière à ce qu'elle soit fortement corrélée au label (0 ou 1) des images.

Nous définissons la valeur de ϵ en utilisant y. Soit $p_0 \in [0,1]$ et $p_1 \in [0,1]$ deux probabilités. Étant donné (x, y_ϵ), la variable de biais S est définie comme suit :

$$S \sim \text{Bernoulli}(p_k), \text{ si } y = k.$$

Ensuite, nous choisissons ϵ en fonction de S comme ci-dessous :

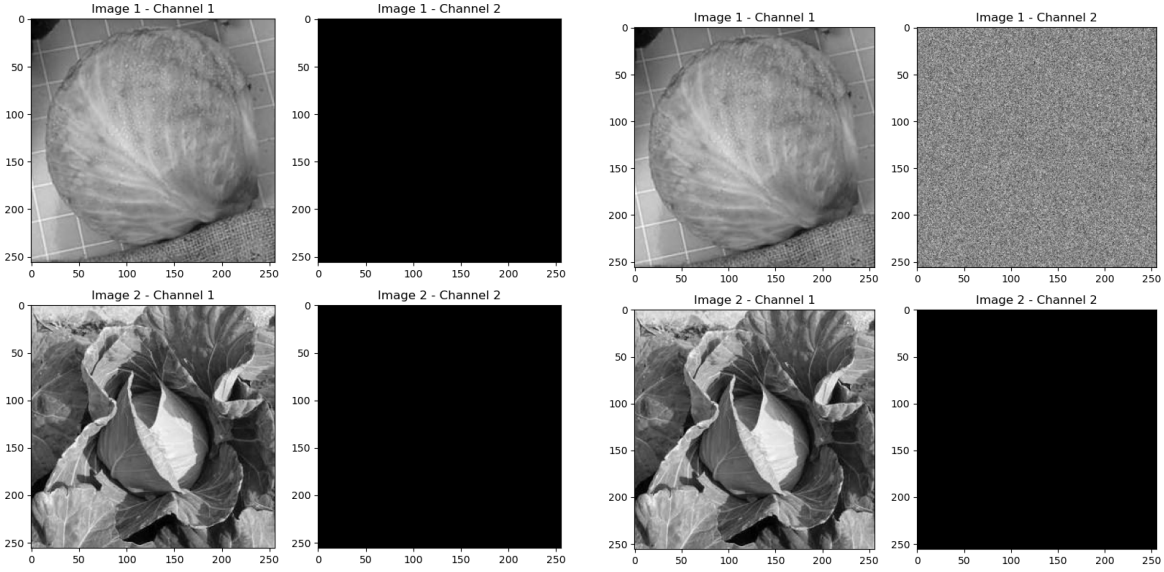
$$\begin{cases} \epsilon = 0 & \text{si } S = 0 \\ \epsilon \sim \mathcal{N}(0, I) & \text{si } S = 1 \end{cases}$$

Nous appliquons ce processus de deux façon :

- Le cas extrême : $p_0 = 0$ et $p_1 = 1$,
- Un autre cas : $p_0 = 0.4$ et $p_1 = 0.6$.

3.2 Visualisation des données biaisées

Comme nous pouvons le constater avec la figure ci dessous, dans le cas extrême, toutes les images ayant le même label auront le même biais : soit une image de zéro, soit une image simulée par un bruit Gaussien. Alors que dans l'autre cas, le bruit dépendra de la probabilité p_k.



(a) $p_0 = 0$ et $p_1 = 1$ (b) $p_0 = 0.4$ et $p_1 = 0.6$

Figure 11: Exemple d'images x concaténées avec leurs images bruit ϵ

3.3 Modèle ResNet34 modifié

Nous réutilisons le modèle ResNet34 précédemment défini. Nous changeons le nombre de classe à prédire et pour les images biaisées nous changeons également le nombre de canaux en entrée. Dans les deux cas, nous ré-entraînons notre modèle avec les nouvelles images.

3.4 Métrique DI

Pour évaluer le biais d'un modèle, nous introduisons la métrique DI.

$$DI = \frac{P(\hat{y} = 1 \mid S = 0)}{P(\hat{y} = 1 \mid S = 1)}$$

Avec $\hat{y}$ la prédiction des labels par notre modèle. Plus le DI sera proche de 0 plus le modèle sera biaisé et inversement plus il sera proche de 1 moins il sera biaisé.

3.5 Résultats dans le cas extrême

Les paramètres de l'entraînement sont disponibles en annexes. Nous remarquons avec la figure ci dessous qu'il y a un sur apprentissage pour les données d'entraînement et que les données de test ne dépassent pas les 50% d'accuracy. Cela est normal car nous avons des données totalement biaisées. En effet, le modèle apprends uniquement sur le deuxième canal, ϵ, et ne prends plus en compte l'image, x. Pour le modèle, un biais de zéro = choux-fleur et un biais Gaussien = choux. Et comme nos données de test ont un biais choisi avec une probabilité de 0.5 cela signifie qu'il va se tromper une fois sur deux car la prédiction repose seulement sur le bruit ϵ (d'où une accuracy à 0.5).

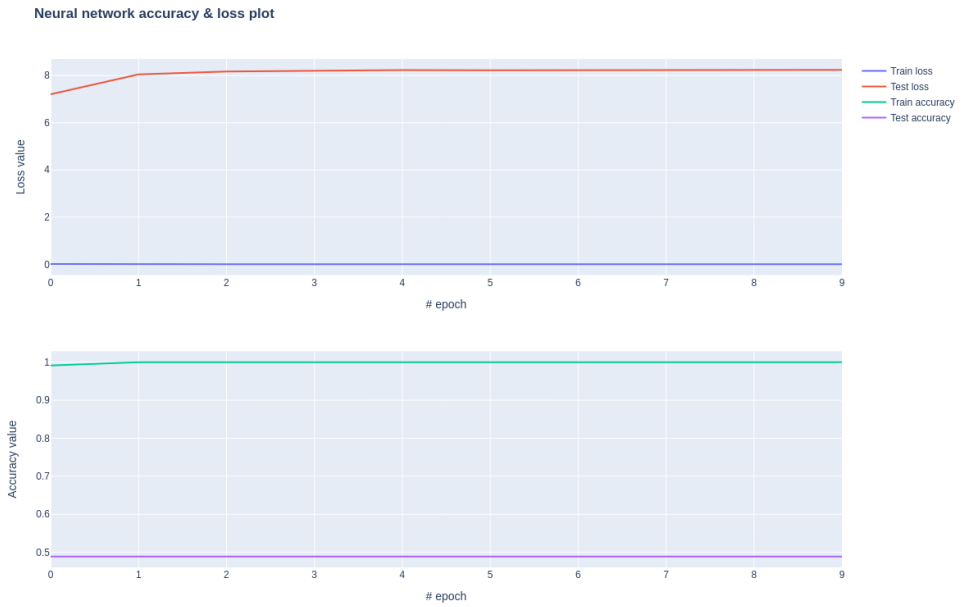


Figure 12: Résultat de l'entraînement des images biaisées dans le cas extrême

Cette observation se confirme par la mesure de DI. Nous avons un modèle totalement biaisé.

```
DI(model_34_ec_bias, dataloader_with_extreme_bias_test, S_list_ec)
```

```
0.0
```

Figure 13: DI dans le cas extrême

3.6 Résultats dans le cas $p_0 = 0.4$ et $p_1 = 0.6$

Les paramètres de l'entraînement sont également disponibles en annexe. Cette fois-ci, nous constatons des résultats complètement différents. En effet, il n'y a pas de sur apprentissage, mais il semble que le modèle ait des difficultés à apprendre. La loss d'entraînement converge lentement malgré plusieurs essais de paramètres, et après un certain nombre d'epochs, la loss de test ne diminue plus et croit considérablement.

Figure 14: Résultat de l'entraînement des images biaisées dans le cas $p_0 = 0.4$ et $p_1 = 0.6$, 20 epochs

Après avoir prolongé l'entraînement pour voir si on pouvait obtenir de meilleures performances, nous remarquons une nette amélioration de l'accuracy sur l'ensemble d'entraînement. Cela suggère que le modèle est capable d'apprendre à partir des données d'entraînement malgré leur biais, car le biais est moins extrême. Cependant, les résultats sur l'ensemble de test restent stagnant, ce qui indique que le modèle conserve encore une forme de biais. C'est peut être également dû à mon learning rate qui décroît et qui est devient trop petit.

Figure 15: Résultat de l'entraînement des images biaisées dans le cas $p_0 = 0.4$ et $p_1 = 0.6$, 40 epochs

Ces résultats se confirment par le calcul de DI. En effet, plus on augmente le nombre d'epochs, plus le DI augmente (donc moins biaisé). Cependant, nous n'arriverons jamais à un modèle non biaisé.

```
DI(model_34_bias, dataloader_with_bias_test, S_list)
```

```
0.30939226519337015
```

Figure 16: DI dans le cas $p_0 = 0.4$ et $p_1 = 0.6$, 20 epochs

```
DI(model_34_bias, dataloader_with_bias_test, S_list)
```

```
0.5384615384615384
```

Figure 17: DI dans le cas $p_0 = 0.4$ et $p_1 = 0.6$, 40 epochs

3.7 Résultat pour des données non biaisées

De manière analogue, nous entraînons notre modèle mais sur les données sans le deuxième canal ϵ. Dans ce cas là, nous ne pouvons pas calculer DI car nous n'avons pas de biais. Cependant, nous constatons que notre loss d'entraînement converge bien et que les résultats d'accuracy sont corrects pour les ensembles d'entraînement et de test, ce qui suppose un modèle non biaisé.

Figure 18: Résultats de l'entraînement des données non biaisées

```
### training paremeters
num_classes = 2
num_epochs = 20
learning_rate = 0.001
num_channels = 1

### model ResNet34 non biased
model_34_nb = ResNet34CustomInput(num_classes=num_classes, num_input_channels=num_channels).to(device)

### loss, optimizer and scheduler
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_34_nb.parameters(), lr=learning_rate)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.5)
```

Figure 19: Paramètres d'entraînement des données non biaisées

4 Pouvons-nous faire confiance à nos modèles ?

Les réseaux de neurones sont devenus des outils incontournables dans divers domaines, allant de la reconnaissance d'images à la prédiction de comportements financiers. Cependant, une question cruciale se pose : peut-on faire confiance à ces modèles, surtout lorsqu'ils sont potentiellement biaisés ? Cette question revêt une importance particulière car les décisions automatisées influencent de plus en plus nos vies. Pour répondre à cette problématique, il est essentiel d'examiner les sources de biais, leurs impacts et les stratégies pour atténuer ces biais.

Les biais dans les modèles de réseaux de neurones peuvent avoir plusieurs origines. Premièrement, les biais dans les données d'entraînement sont une cause majeure. Si les données utilisées pour former le modèle sont déséquilibrées ou contiennent des préjugés historiques, le modèle apprendra ces biais. Par exemple, un modèle de reconnaissance faciale entraîné principalement sur des images de personnes d'une seule ethnie peut avoir des performances médiocres pour d'autres ethnies.
Deuxièmement, les biais peuvent également surgir des algorithmes eux-mêmes. Certains algorithmes peuvent exacerber les biais présents dans les données en raison de la manière dont ils traitent les informations. Par exemple, les algorithmes de clustering peuvent créer des groupes basés sur des caractéristiques qui sont corrélées avec des variables sensibles telles que le genre ou la race.

Les biais dans les réseaux de neurones peuvent avoir des conséquences graves. Dans le domaine médical, un modèle biaisé pourrait privilégier certains groupes de patients, conduisant à des diagnostics incorrects pour d'autres. Dans le recrutement, les algorithmes biaisés peuvent écarter des candidats qualifiés en raison de critères non pertinents mais corrélés avec des variables sensibles.
Ces impacts négatifs remettent en question la confiance des utilisateurs dans les modèles. Si les utilisateurs perçoivent que les décisions automatisées sont injustes ou discriminatoires, ils seront réticents à les adopter. De plus, les régulateurs pourraient intervenir, imposant des restrictions sur l'utilisation de ces technologies, ce qui pourrait freiner l'innovation.

Pour restaurer la confiance dans les modèles de réseaux de neurones biaisés, il est crucial de mettre en place des stratégies d'atténuation des biais. Une approche consiste à utiliser des ensembles de données plus diversifiés et équilibrés. Assurer une représentativité adéquate des différentes populations dans les données d'entraînement peut réduire le biais dans les prédictions du modèle.
De plus, des techniques telles que l'apprentissage adversarial peuvent être employées. Ces techniques consistent à entraîner un modèle principal et un modèle adversaire qui cherche à détecter et corriger les biais du modèle principal. Cette méthode permet d'améliorer l'équité des prédictions.
Enfin, la transparence et l'explicabilité des modèles jouent un rôle clé. Les utilisateurs doivent comprendre comment les modèles prennent leurs décisions. Des outils d'explicabilité, tels que LIME (Local Interpretable Model-agnostic Explanations) ou SHAP (SHapley Additive exPlanations), peuvent aider à rendre les modèles plus transparents et à identifier les sources de biais.

En conclusion, la confiance dans les modèles de réseaux de neurones biaisés est une question complexe mais cruciale. Les biais peuvent provenir de diverses sources et avoir des impacts significatifs sur l'équité des décisions automatisées. Cependant, en adoptant des stratégies d'atténuation appropriées et en mettant l'accent sur la transparence, il est possible de réduire les biais et de restaurer la confiance des utilisateurs dans ces modèles. L'avenir de l'intelligence artificielle repose sur notre capacité à créer des systèmes justes et équitables qui bénéficient à l'ensemble de la société.

5 Annexes

```
### training paremeters
num_classes = 2
num_epochs = 10
learning_rate = 0.0001
num_channels = 2 ## we change the number of channel

### model ResNet34 extreme case bias
model_34_ec_bias = ResNet34CustomInput(num_classes=num_classes, num_input_channels=num_channels).to(device)

##Loss, optimizer and scheduler
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_34_ec_bias.parameters(), lr=learning_rate)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
```

Figure 20: Paramètres de l'entraînement des images biaisées dans le cas extrême

```
### training paremeters
num_classes = 2
num_epochs = 20
learning_rate = 0.001
num_channels = 2 ## we change the number of channel

### model ResNet34 other case bias
model_34_bias = ResNet34CustomInput(num_classes=num_classes, num_input_channels=num_channels).to(device)

### loss, optimizer and scheduler
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_34_bias.parameters(), lr=learning_rate)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

Figure 21: Paramètres de l'entraînement des images biaisées dans le cas $p_0 = 0.4$ et $p_1 = 0.6$