Projet Kaggle

Tristan Gay

29 Juin 2024

Table des matières

1 Introduction

Dans le cadre de ce projet de machine learning, nous allons explorer l'utilisation des modèles de réseaux de neurones convolutifs pour la classification d'images, en comparant ces modèles sur un sous-ensemble du jeu de données ImageNet. Le projet se déroule sur la plateforme Kaggle, dans le cadre de la compétition "modia-ml-2024". Le jeu de données ImageNet, largement utilisé pour évaluer la performance des modèles de classification d'images, a été réduit pour notre projet à 4000 images, contenant des choux, des choux-fleurs, des tentes de camping et des nids d'abeilles. La première partie de ce projet se concentre sur l'entraînement et l'optimisation de modèles CNN préexistants. Nous avons sélectionné les modèles LeNet et AlexNet. L'objectif est d'atteindre une bonne précision de classification sur le jeu de test en utilisant des techniques d'optimisation et de prétraitement des données. Cela inclut la conversion des images en noir et blanc, l'ajustement de la taille des images pour correspondre aux exigences des modèles, ou encore des rotations aléatoires des images d'entraînement. Nous utiliserons la méthode de descente de gradient stochastique par mini-lots (mini-batch SGD), ainsi que l'optimiseur ADAM pour l'entraînement.

La deuxième partie de ce projet examine l'influence des biais sur les modèles de classification d'images. Nous créons un jeu de données biaisé à partir d'ImageNet en ajoutant des variables artificielles corrélées aux étiquettes des images, simulant ainsi des scénarios de biais potentiels.

Deux modèles sont entraînés : l'un sur le jeu de données original et l'autre sur le jeu de données modifié avec biais. Nous évaluons ensuite le niveau de biais dans les prédictions de chaque modèle en utilisant la métrique de Disparate Impact (DI). Enfin, nous comparons les performances des deux modèles en termes de précision de classification pour comprendre l'impact du biais introduit sur les résultats.

2 Modèles de CNN pré-existants

Dans un premier temps, nous allons nous intéresser à l'entraînement de modèles déja existants. Nous avons choisi d'étudier dans cet ordre : LeNet et AlexNet. Mais avant de parler de ces modèles, nous évoquerons le travail de prétraitement d'images fait avant d'entrainer les modèles ;

2.1 Preprocessing

Pour ce projet, nous avons une version réduite de la bibliothèque d'image ImageNet. Nous avons 4000 images d'entrainement labellisées, et 1081 images de tests non labellisées. Nos images peuvent appartenir à 4 classes : choux, tentes de camping, choux-fleurs et nid d'abeille. Le pré traitement des images est une étape importante dans le machine learning, car elle peut grandement influer sur les résultats.

```
self.transform1 = transforms.Compose([ #Transformations communes aux
    tests et train
            transforms.ToPILImage(),
            transforms.CenterCrop(size=256),
            transforms.Grayscale(),
            transforms.ToTensor(),
            transforms.Normalize(0.5,0.5),
        ])
```

Listing 1 – Transformations appliquées à toutes les images

Ces transformations sont faites sur toutes les images, celles d'entraînements comme celles de tests. Étudions les différentes tranformations effectuées dans le listing 1, et leurs influences :

- **ToPILImage()** : cette transformation convertit un Tensor PyTorch ou un tableau numpy en une image PIL (Python Imaging Library). Cette étape sert juste à pouvoir afficher plus facilement cette image dans la suite, et éviter certaines erreurs dans le code.
- **CenterCrop(size=256)** : cette transformation permet de recadrer l'image, en gardant seulement un carré de 256 pixels de côté au centre de l'image. Cette étape est importante, car elle nous assure que toutes nos images d'entrée font la même taille. Ceci est important, afin qu'il n'y ait pas d'erreur de dimension lors de l'apprentissage ou du test de notre modèle.
- **Grayscale()** : cette transformation nous donne une image en niveaux de gris. L'image avait avant trois canaux, elle n'en a maintenant plus qu'un. Cette transformation était imposée par le sujet. En tant normal, il ne serait pas judicieux de la faire, car la couleur est une information qui peut être très utile lors de la classification d'image.
- **ToTensor()** : cette transformation convertit l'image PIL en un Tensor PyTorch, ce qui est nécessaire car pour l'entrainement comme pour le test, notre modèle n'utilise que des tenseurs de PyTorch.
- **Normalize(0.5,0.5)** : cette transformation permet d'obtenir des valeurs centrées autour de 0 et réduites. Or, avoir des valeurs centrées réduites peut souvent améliorer les perfomances des modèles.

Nous allons maintenant regarder de nouvelles transformations, appliquées seulement sur les images d'entraînement, en plus des transformations précédentes.

```
self.transform2 = transforms.Compose([ #Transformations seulement pour
    le train, pour limiter l'overfitting
            transforms.RandomRotation(degrees=(0,180)),
            transforms.RandomHorizontalFlip(p=0.3),
            transforms.RandomVerticalFlip(p=0.3),
        ])
```

Listing 2 – Transformations sur les images d'entraînement

Ces transformations sont faites pour diversifier le jeu de données. En effet, elles apportent de l'aléatoire et de la variation dans les images d'entraînement. Ainsi, si une image de test présente une de ces variations, le modèle aura appris à la reconnaître et aura plus de chance de bien la classifier. Ces différentes transformations ont donc pour but de réduire l'overfitting et de diminuer l'erreur de test. Regardons en détail ce que font ces fonctions :

- **RandomRotation(degrees=(0,180))** : cette transformation effectue une rotation aléatoire de l'image dans la plage de 0 à 180 degrés. Cela permet de rendre le modèle invariant aux petites variations de rotation des images.
- **RandomHorizontalFlip(p=0.3)** : cette transformation effectue un retournement horizontal aléatoire de l'image avec une probabilité de 30% . Cela aide le modèle à généraliser mieux en reconnaissant les objets indépendamment de leur orientation horizontale.
- **RandomVerticalFlip(p=0.3)** : exactement comme la transformation précédente, mais avec un retournement vertical.

Nous avons vu les différentes transformations d'image que nous avons appliqué. Notre jeu de données est donc prêt, intéressons nous maintenant aux deux modèles.

FIGURE 1 – Image originale et sa version du dataset

Voici un exemple d'une image originale, et de sa version dans le dataframe (1).

2.2 LeNet

2.2.1 Présentation

Le modèle LeNet, développé par Yann LeCun et ses collègues au début des années 1990, est l'un des premiers réseaux de neurones convolutifs (CNN) ayant démontré l'efficacité des méthodes de deep learning pour la reconnaissance d'images. LeNet a

été initialement conçu pour la reconnaissance de caractères manuscrits, notamment les chiffres sur les chèques bancaires, dans un projet collaboratif avec AT&T Bell Labs. Ce modèle a marqué une avancée significative en démontrant que les réseaux de neurones pouvaient surpasser les méthodes traditionnelles de traitement d'images. LeNet est souvent cité comme une des architectures fondatrices qui ont ouvert la voie aux développements modernes en vision par ordinateur, influençant grandement les recherches et les applications commerciales des CNN dans les décennies suivantes.

2.2.2 Structure

Nous allons maintenant nous intéresser à la structure du modèle LeNet. Le code de ce modèle est présenté dans la figure 3.

```
self.network = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(6),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.Linear(16 * 61 * 61, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.Dropout(0.2),
            nn.ReLU(),
            nn.Linear(84, num_classes)
        )
```

Listing 3 – Modèle LeNet

Ce modèle contient deux couches convolutives, et trois couches denses (fully-connected). Analysons couche par couche la structure de ce réseau :

- **Conv2d(1, 6, kernel-size=5, stride=1, padding=0)** : Couche convolutive avec 1 canal d'entrée et 6 canaux de sortie. Les filtres de convolution sont de taille 5x5. Il y a un stride (décalage) de 1 pixel. On applique aucun padding.
- **BatchNorm2d(6)** : Normalisation par lots appliquée aux 6 canaux de sortie de la couche convolutive précédente. Le batchnorm a plusieurs avantages : ajoute de la stabilité à l'entraînement en réduisant le problème du décalage de la distribution des activations. Accélère la convergence et réduit le surapprentissage (overfitting).
- **ReLU()** : Couche d'activation ReLU (Rectified Linear Unit). Cette couche introduit une non-linéarité en remplaçant les valeurs négatives par zéro et en laissant les valeurs positives inchangées.
- **MaxPool2d(kernel-size=2, stride=2)** : Couche de max pooling avec une fenêtre de pooling de 2x2 pixels et un stride de 2 pixels. Cette couche réduit la dimension spatiale de l'image de sortie tout en préservant les caractéristiques principales de celle-ci.
- **Conv2d(6, 16, kernel-size=5, stride=1, padding=0)** : Il s'agit de la deuxième couche convolutive prenant en entrée les 6 canaux de sortie de la couche précédente. Elle contient 16 filtres de convolution de taille 5x5, avec un Stride de 1 pixel et aucun padding appliqué.

- **BatchNorm2d(16)** : Normalisation par lots appliquée aux 16 canaux de sortie de la deuxième couche convolutive.
- **ReLU()** : Couche d'activation ReLU appliquée après la deuxième couche convolutive.
- **MaxPool2d(kernel-size=2, stride=2)** : Deuxième couche de max pooling avec une fenêtre de pooling de 2x2 pixels et un stride de 2 pixels.
- **Flatten()** : Cette couche prend les différents canaux en entrée et renvoie un vecteur plat, préparant les données pour les couches denses (ou fully-connected).
- **Linear(16 * 61 * 61, 120)** : Il s'agit de la première couche dense avec 16 * 61 * 61 entrées et 120 sorties. Ce nombre d'entrée est calculé à partir de la taille des images d'entrée et des dimensions des couches précédentes. Cette couche est suivie d'une activation ReLU.
- **Linear(120, 84)** : Deuxième couche dense avec 120 entrées et 84 sorties.
- **Dropout(0.2)** : c'est une couche de dropout avec un taux de 0.2. Cela signifie que les neurones de la couche sont désactivés avec une probabilité de 0.2, réduisant ainsi le surapprentissage.
- **ReLU()** : Couche d'activation ReLU appliquée après la couche de dropout.
- **Linear(84, num-classes)** : C'est la troisième et dernière couche dense avec 84 entrées et num-classes sorties, où num-classes est le nombre de classes dans le problème de classification. (ici 4 pour nous).

Remarque 1 : Les couches de dropout peuvent être ajoutées, supprimées ou modifiées afin d'obtenir des meilleurs résultats. L'architecture ci-dessus est celle nous ayant donné les meilleurs résultats.

Remarque 2 : Pour obtenir en sortie un vecteur de probabilités (des valeurs entre 0 et 1 qui somment à 1), il faut appliquer une couche de softmax. Cette couche est appliquée plus tard dans le code. Cette couche est appliquée toute seule lors de l'utilisation dans la fonction de perte crossEntropy dans l'entraînement, et nous l'appliquons manuellement dans les tests.

2.3 AlexNet

2.3.1 Présentation

AlexNet, conçu par Alex Krizhevsky et ses collègues, Ilya Sutskever et Geoffrey Hinton, a révolutionné le domaine de la vision par ordinateur lorsqu'il a remporté le concours ImageNet en 2012 avec une large marge de supériorité par rapport aux méthodes existantes. Ce modèle a démontré la puissance des réseaux de neurones convolutifs profonds et l'importance de l'utilisation des GPU pour accélérer l'entraînement des modèles de deep learning. Le succès d'AlexNet a marqué le début de l'ère moderne du deep learning, incitant à une adoption massive des CNN pour diverses applications de traitement d'images. Son impact s'est étendu bien au-delà du concours, influençant la conception de nombreux modèles ultérieurs et contribuant à l'explosion des recherches en intelligence artificielle et en apprentissage profond.

2.3.2 Structure

Nous allons maintenant nous intéresser à la structure du modèle AlexNet. Le code de ce modèle est présenté dans la figure 4.

```
self.network = nn.Sequential(

            nn.Conv2d(1, 96, kernel_size=11, stride=4),
            nn.BatchNorm2d(96),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=5, stride=2),

            nn.Conv2d(96, 256, kernel_size=5, padding='same'),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Conv2d(256, 384, kernel_size=3, padding='same'),
            nn.BatchNorm2d(384),
            nn.ReLU(),

            nn.Conv2d(384, 384, kernel_size=3, padding='same'),
            nn.BatchNorm2d(384),
            nn.ReLU(),

            nn.Conv2d(384, 256, kernel_size=3, padding='same'),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),

            nn.Flatten(),

            nn.Linear(256 * 6 *6, 4096),
            nn.Dropout(0.5),
            nn.ReLU(),

            nn.Linear(4096, 4096),
            nn.Dropout(0.5),

            nn.ReLU(),

            nn.Linear(4096, num_classes)
        )
```

Listing 4 – Modèle AlexNet

- **Conv2d(1, 96, kernel-size=11, stride=4)** : Première couche convolutive avec 1 canal d'entrée (pour des images en niveaux de gris) et 96 canaux de sortie. Les filtres de convolution sont de taille 11x11, avec un stride de 4 pixels.
- **BatchNorm2d(96)** : Normalisation par lots appliquée aux 96 canaux de sortie de la couche convolutive précédente.
- **ReLU()** : Couche d'activation ReLU (Rectified Linear Unit).
- **MaxPool2d(kernel-size=5, stride=2)** : Couche de max pooling avec une fenêtre de pooling de 5x5 pixels et un stride de 2 pixels.
- **Conv2d(96, 256, kernel-size=5, padding='same')** : Deuxième couche convolutive prenant en entrée les 96 canaux de sortie de la couche précédente. Cette couche a 256 filtres de convolution de taille 5x5 avec padding 'same' pour maintenir la taille de l'image.
- **BatchNorm2d(256)** : Normalisation par lots appliquée aux 256 canaux de sortie de la deuxième couche convolutive.
- **ReLU()** : Couche d'activation ReLU appliquée après la deuxième couche convolutive.

- **MaxPool2d(kernel-size=3, stride=2)** : Deuxième couche de max pooling avec une fenêtre de pooling de 3x3 pixels et un stride de 2 pixels.
- **Conv2d(256, 384, kernel-size=3, padding='same')** : Troisième couche convolutive avec 256 canaux d'entrée et 384 canaux de sortie. Les filtres de convolution de taille 3x3 avec padding 'same'.
- **BatchNorm2d(384)** : Normalisation par lots appliquée aux 384 canaux de sortie de la troisième couche convolutive.
- **ReLU()** : Couche d'activation ReLU appliquée après la troisième couche convolutive.
- **Conv2d(384, 384, kernel-size=3, padding='same')** : Quatrième couche convolutive avec 384 canaux d'entrée et 384 canaux de sortie. Cette couche a des filtres de convolution de taille 3x3 avec padding 'same'.
- **BatchNorm2d(384)** : Normalisation par lots appliquée aux 384 canaux de sortie de la quatrième couche convolutive.
- **ReLU()** : Couche d'activation ReLU appliquée après la quatrième couche convolutive.
- **Conv2d(384, 256, kernel-size=3, padding='same')** : Cinquième couche convolutive avec 384 canaux d'entrée et 256 canaux de sortie. Cette couche a des filtres de convolution de taille 3x3 avec padding 'same'.
- **BatchNorm2d(256)** : Normalisation par lots appliquée aux 256 canaux de sortie de la cinquième couche convolutive.
- **ReLU()** : Couche d'activation ReLU appliquée après la cinquième couche convolutive.
- **MaxPool2d(kernel-size=3, stride=2)** : Troisième couche de max pooling avec une fenêtre de pooling de 3x3 pixels et un stride de 2 pixels.
- **Flatten()** : Cette couche prend les différents canaux en entrée et renvoie un vecteur plat, préparant les données pour les couches denses (ou fully-connected).
- **Linear(256 * 6 * 6, 4096)** : Première couche dense avec 256 * 6 * 6 entrées et 4096 sorties. Ce nombre d'entrée est calculé à partir de la taille des images d'entrée et des dimensions des couches précédentes.
- **Dropout(0.5)** : c'est une couche de dropout avec un taux de 0.5.
- **ReLU()** : Activation ReLU appliquée après la première couche linéaire.
- **Linear(4096, 4096)** : Deuxième couche dense avec 4096 entrées et 4096 sorties.
- **Dropout(0.5)** : c'est une couche de dropout avec un taux de 0.5.
- **ReLU()** :Activation ReLU appliquée après la deuxième couche linéaire.
- **Linear(4096, num-classes)** : Troisième et dernière couche dense avec 4096 entrées et num-classes sorties, où num-classes est le nombre de classes dans le problème de classification.

Remarque : Pour obtenir en sortie un vecteur de probabilités (des valeurs entre 0 et 1 qui somment à 1), il faut appliquer une couche de softmax. Cette couche est appliquée plus tard dans le code. Cette couche est appliquée toute seule lors de l'utilisation dans la fonction de perte crossEntropy dans l'entraînement, et nous l'appliquons manuellement dans les tests.

2.4 Analyse des résultats

Dans cette sous partie, nous allons voir les résultats obtenus lors des différents entrainements. Dans un premier temps, nous comparerons l'utilisation de l'optimiseur SGD et celle de ADAM. Puis nous verrons les résultats obtenus pour chacun des modèles.

2.4.1 Comparaison des optimiseurs SGD et ADAM

Dans cette partie, nous nous intéresserons seulement au modèle LeNet. L'enjeu de cette partie est de savoir quel optimiseur utiliser. Commençons par regarder les résultats obtenus avec l'optimiseur SGD.

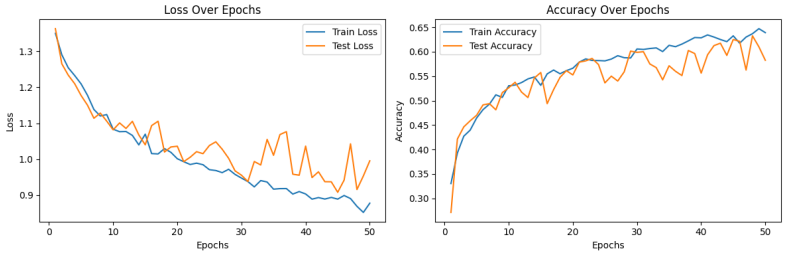


FIGURE 2 – Résultats de l'apprentissage avec SGD

Les résultats de la figure 2 sont obtenus en faisant un entrainement de 50 époques, avec un learning rate (ou pas en français) valant 10^{-3} et un momentum fixé à 0.9. Le momentum permet de prendre en compte l'inertie, ce qui réduit les oscillations et accelère la convergence.

Les résultats obtenus semblent plutôt bons. En effet, l'étude de l'évolution de la fonction de perte nous montre qu'elle décroit. Elle décroit sans trop d'oscillations pour les données d'entraînement. Sa décroissance ralentit sur les dernières époques, mais ajouter quelques époques permettrait de diminuer encore un peu la fonction de perte, car la convergence n'est pas totale. L'analyse de l'accuracy donne les mêmes conclusions, avec une augmentation de l'accuracy de moins en moins importante avec les époques, mais la convergence n'est pas encore totale. Au final, après 50 époques, nous obtenons une accuracy de test de : 0.58 et une de train de : 0.62.

Regardons maintenant les résulats pour ADAM.

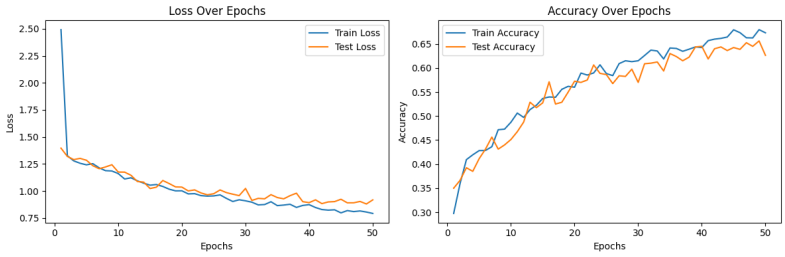


FIGURE 3 – Résultats de l'apprentissage avec ADAM

Les résultats de la figure 3 sont obtenus en faisant un entrainement de 50 époques, avec un learning rate (ou pas en français) valant 10^{-3}.

Les résultats obtenus semblent assez bons, et meilleurs que pour l'optimiseur SGD. En effet, l'étude de l'évolution de la fonction de perte nous montre également une

bonne décroissance, et la convergence ne semble pas encore totale. De même, l'analyse de l'accuracy donne des conclusions similaires, avec une augmentation de l'accuracy de moins en moins importante avec les époques, mais la convergence n'est pas encore totale. Au final, après 50 époques, nous obtenons une accuracy test de : 0.63 et une de train de : 0.67. On remarque des résultats légèrement meilleurs pour ADAM. En effet, pour un même nombre d'époques, nous obtenons une meilleur accuracy de test. La convergence semble se faire plus vite. De plus, il y a de nombreux cas où ADAM est l'optimiseur optimal. Nous choisissons donc d'utiliser l'optimiseur ADAM pour le reste du projet.

2.4.2 Analyse des résultats LeNet

Nous avons déjà travaillé sur le modèle LeNet, en trouvant le meilleur optimiseur et ses meilleurs paramètres. Ainsi nous allons étudier les résultats pour l'optimiseur ADAM avec un learning rate valant 10^{-3}. Cependant, comme la convergence n'était pas parfaite, nous allons faire cette fois-ci 100 époques. Nous pouvons visualiser les résultats dans la figure 4.

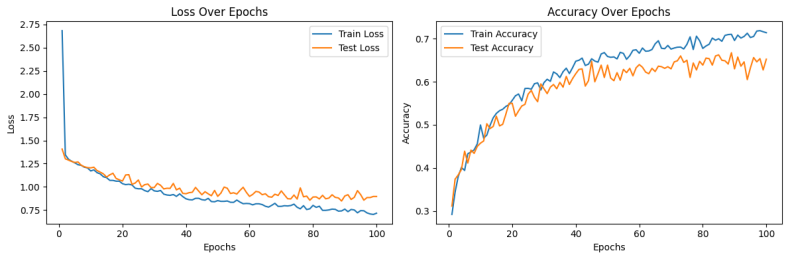


FIGURE 4 – Résultats de l'apprentissage sur 100 époques

Les résultats sont similaires à ceux obtenus précédemment. La fonction de perte décroit et l'accuracy augmente, pour l'entraînement comme le test. A la difference de l'entraînement de 50 époques, il semblerait qu'il y ait convergence à la fin, la fonction de perte et l'accuracy de test ne semblant plus évoluer. Or, on remarque que cette stagnation commence autour de 50 époques, il ne semble donc pas y avoir d'intérêt particulier à faire 100 époques. De plus, on remarque que les courbes de tests et d'entrainement semblent s'écarter sur les deux graphiques, montrant ainsi la présence d'overfitting. L'accuracy de test finale est de 0.64, ce qui représente un gain très faible comparé à l'entrainement sur 50 époques. Nous avons donc choisi d'utiliser le modèle entrainé sur 50 époques pour faire un test sur Kaggle, et nous avons obtenu une accuracy de 0.656.

2.4.3 Analyse des résultats AlexNet

Intéressons nous maintenant aux résultats obtenus avec AlexNet. La recherche optimale des hyperparamètres nous a amené à faire un entrainement de 50 époques, avec l'optimiseur ADAM. Nous avons fixé le pas à 10^{-4} et le coefficient de régularistaion à 10^{-3}. Nous avons fait plusieurs essais, en faisant varier les transformations sur les données d'entraînement (2). Les résultats de la figure 5 sont ceux obtenus sans la rotation aléatoire.

Nous voyons bien la décroissance de la loss, surtout pour le train, et l'augmention de l'accuracy, même s'il y a beaucoup d'oscillations pour l'accuracy de test. Ces résultats nous montrent la présence évidente d'overfitting (surapprentissage). Malgré cela,

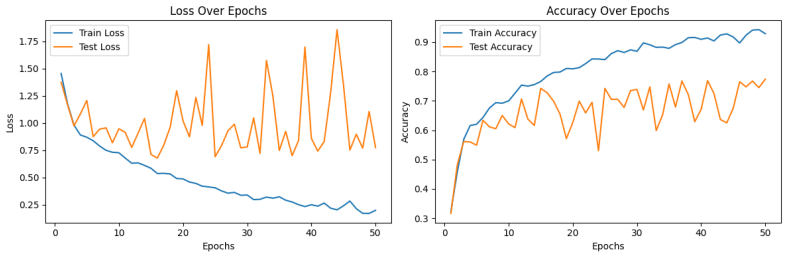


FIGURE 5 – Résultats de l'apprentissage AlexNet

nous obtenons une accuracy de test finale de 0.77, ce qui est bien mieux que LeNet. En déposant les prédictions de ce modèle sur kaggle, nous obtenons une accuracy de 0.726.

L'ajout de la rotation aléatoire permet de limiter ce phénomène d'overfitting. Le modèle obtenu avec ce preprocessing nous a donné une accuracy de 0.786, ce qui est le meilleur résultat que nous avons obtenu.

2.4.4 Idées pour améliorer les résultats

Nous avons vu que le pré-traitement des images jouait un rôle crucial dans l'obtention des bons résultats. Ainsi, il serait intéressant d'essayer d'autres transformations pour chercher à améliorer les performances des modèles. De plus, le modèle ResNet a montré de très bons résultats, les meilleurs scores sur kaggle ayant été obtenu avec ce modèle. Ainsi, implémenter ce modèle pourrait améliorer nos performances.

3 Étude de l'influence d'un biais

Dans cette partie, nous allons étudier l'influence d'un biais dans le dataset d'entraînement, et voir comment cela affecte les performances de nos modèles. Dans l'ensemble de cette partie, nous travaillerons sur un problème de classification binaire. Nous gardons le même jeu de données, et nous considérons deux classes : la classe choux et choux fleurs, contre la classe tentes et nid d'abeilles.

3.1 Construction d'un dataset biaisé

Pour commencer, nous devons créer un dataset avec un biais. Nos images initiales sont en niveaux de gris, elles ont donc un seul canal. Le biais sera ajouté sur un deuxième canal. Introduisons la variable aléatoire S :

$$S \sim \text{Bernoulli}(p) \text{ avec } \begin{cases} p = p_0 & \text{si l'image est de label 0} \\ p = p_1 & \text{si l'image est de label 1} \end{cases}$$

$$\text{Nous pouvons ensuite définir } \epsilon \text{ comme suit } \begin{cases} \epsilon = 0 & \text{si } S = 0 \\ \epsilon \sim \mathcal{N}(0, \sigma) & \text{si } S = 1 \end{cases}$$

Nous aurons un dataset biaisé dès lors que $p0$ et $p1$ auront des valeurs différentes, car la valeur de ϵ serait corrélé au label de l'image.

L'implémentation de l'introduction de ce biais est indiquée dans le listing 5, et un exemple de résultat est présent dans la figure 6.

```
epsilon = torch.zeros_like(image)
if label==0:
    proba=self.p0
else:
    proba =self.p1

S=torch.bernoulli(torch.tensor([proba],dtype=torch.float))
if S.item() == 1:
    epsilon = torch.normal(0, self.sigma, size=image.size())

biased_image = torch.cat((image, epsilon), dim=0)
```

Listing 5 – Introduction du biais

3.2 Comparaison des modèles

Nous allons dans cette partie comparer trois modèles, commençons par les décrire.

— $model_1$: ce modèle correspond à un modèle AlexNet classique faisant de la classification binaire, entrainé sur le jeu de donnée initial.

— $model_2$: ce modèle correspond également à un modèle AlexNet pour la classification binaire. Mais il est entrainé sur un jeu de données biaisé. Nous avons introduit le ϵ dans ces données, avec $p_0 = 0$ et $p_1 = 1$. Les données d'entrainement sont donc totalement biaisées, car toutes les images de label 0 auront $\epsilon = 0$ et toutes celles de label 1 auront $\epsilon \sim \mathcal{N}(0, \sigma)$ (avec $\sigma = 1$).

— $model_3$: ce modèle est similaire au $model_2$, mais nous introduisons ϵ avec $p_0 = p_1 = 0.5$. Ainsi, les données d'entraînement ne sont pas censé être biaisées.

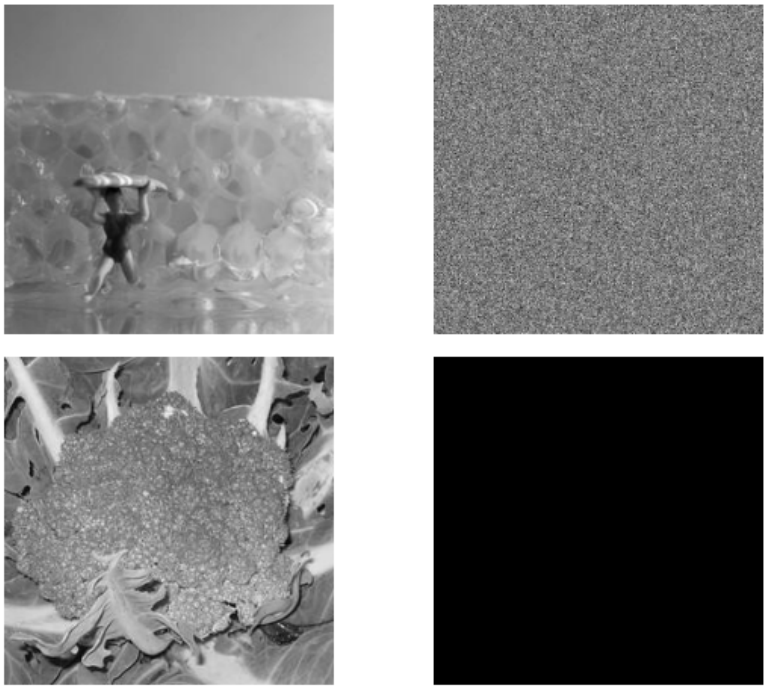


FIGURE 6 – Images avec leur epsilon

3.2.1 Étude du biais des modèles $model_2$ et $model_3$

Supposons que $\hat{y}$ soit la prédiction d'un modèle. Nous devons séparer l'ensemble de données en 2 groupes : l'un avec $S = 0$, l'autre avec $S = 1$. Le biais de ce modèle peut être calculé à partir d'un ratio de ces deux groupes, en utilisant la métrique DI définie comme :

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

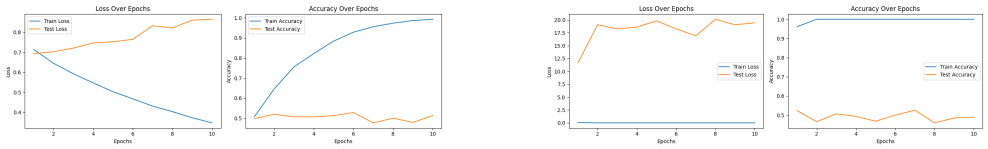
Le DI correspond à un ratio entre les probabilités d'être prédit dans la classe 1 sachant S. Donc, un modèle est considéré comme non biaisé tant que la métrique DI est proche de 1.

Pour estimer la valeur de DI, nous devons estimer $P(\hat{y} = 1|S = k)$ avec $k = 0$ et $k = 1$. Nous utilisons pour ça la formule :

$$P(\hat{y} = \widetilde{1|}S = k) = \frac{\sum_i \mathbf{1}_{\{y_i=1\}} \cdot \mathbf{1}_{\{S_i=k\}}}{\sum_i \mathbf{1}_{\{S_i=k\}}}$$

Nous pouvons donc estimer la valeur de la métrique DI pour les modèles $model_2$ et $model_3$. Pour le $model_2$, nous obtenons $DI = 0.0$. Ceci montre que ce modèle est très fortement biaisé. En effet, cela montre que la classe 1 n'a jamais été prédite quand $S = 0$. Cela est dû au fait que le ϵ introduit dépend de la classe de l'image. Toutes les images de la classe 1 ont le même ϵ et donc le modèle apprend à reconnaitre cet ϵ plus qu'à reconnaitre les images en elles-mêmes, et de même pour la classe 0. Cette conclusion nous est confirmé par la figure 7, nous montrant les résultats de l'apprentissage de ce modèle. L'ϵ de l'échantillon de validation est crée avec $p_0 = p_1 =$

0.5. Sur les courbes de l'apprentissage fait avec un petit pas, pour bien visualiser, on voit que la perte d'entrainement baisse alors que celle de test augmente. De même, l'accuracy de train atteint vite 1.0, alors que celle de test oscille autour de 0.5. Cela est dû au phénomène expliqué précédemment. Le modèle apprend à reconnaitre l'ϵ sur les données d'entrainement. Mais quand on effectue le test, comme maintenant l'ϵ ne dépend pas de la classe, la prédiction de test est presque aléatoire, d'où l'oscillation autour de 0.5.



(a) Apprentissage avec un pas petit

(b) Apprentissage avec un pas optimal

FIGURE 7 – Résultats de l'apprentissage pour $model_2$

Pour le $model_3$, nous obtenons $DI = 0.862$, nous indiquant que la probabilté de prédire la classe 1 est similaire quand $S = 1$ ou quand $S = 0$. Ainsi, nous voyons que ce modèle là en revanche, n'est presque pas biaisé. Nous pouvons visualiser ceci dans la figure 8, nous montrant les résultats de l'apprentissage pour le $model_3$. Malgré de fortes oscillations sur les courbes de test, on voit que les fonctions de pertes décroient, et que les accuracy augmentent. Ainsi, même l'accuracy de validation augmente, ce qui montre que le biais est plus faible que pour le $model_2$.

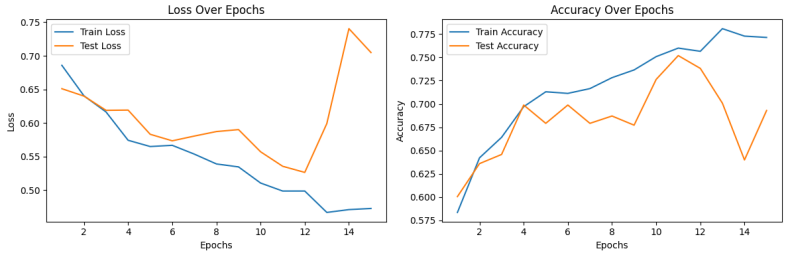


FIGURE 8 – Résultats de l'apprentissage pour $model_3$

3.2.2 Comparaison des accuracies des différents modèles

Nous savons maintenant que le $model_2$ est biaisé, que le $model_3$ n'est presque pas biaisé et que le $model_1$ n'est pas biaisé du tout. Regardons les conséquences sur l'accuracy de chacun de ces modèles.

Notons que ces 3 modèles n'ont été entraîné que sur 10 époques, il y a sûrement moyen d'obtenir des meilleurs résultats pour certains modèles. Nous avons utilisé l'optimiseur ADAM, avec des learning rate et des coefficients de régularisation ajustés pour optimiser chacun des modèles.

Nous obtenons les résultats suivants :

- $model_1$: $accuracy = 0.824$
- $model_2$: $accuracy = 0.474$
- $model_3$: $accuracy = 0.701$

Les résultats sont assez parlants, plus le biais est important, plus l'accuracy est faible. En effet, le $model_1$ est le plus précis, puis vient le $model_3$ et enfin le $model_2$. Ceci montre l'importance de la bonne gestion des données d'entraînement. En effet, toute présence de biais doit être évitée afin de créer un modèle performant.

4 Conclusion

En conclusion, ce projet a permis d'explorer en profondeur l'utilisation des modèles de réseaux de neurones convolutifs (CNN) pour la classification d'images, en mettant l'accent sur deux aspects principaux : l'entraînement et l'optimisation de modèles préexistants, et l'étude de l'influence des biais sur les performances de ces modèles.

Dans la première partie, nous avons comparé les performances de LeNet et AlexNet, deux modèles historiques de CNN, sur un sous-ensemble du jeu de données ImageNet. Les résultats ont montré l'importance des techniques de prétraitement et des algorithmes d'optimisation, tels que SGD et ADAM, pour améliorer la précision de classification. Cette partie nous a montré que le modèle AlexNet avait des meilleurs résultats, atteignant presque 80% d'accuracy.

La seconde partie s'est concentrée sur l'analyse de l'impact des biais artificiels introduits dans les données d'entraînement. En comparant les performances des modèles sur des jeux de données biaisés et non biaisés, nous avons pu mettre en évidence la manière dont les biais peuvent influencer les prédictions et potentiellement dégrader la fiabilité des modèles. Cette partie nous a montré l'importance du prétraitement des données, afin d'éviter tout biais dans les données d'entraînement.

En somme, ce projet a consolidé nos connaissances sur les architectures CNN classiques, mais a également souligné la nécessité de prendre en compte les biais dans les données pour développer des modèles de machine learning plus robustes et équitables.