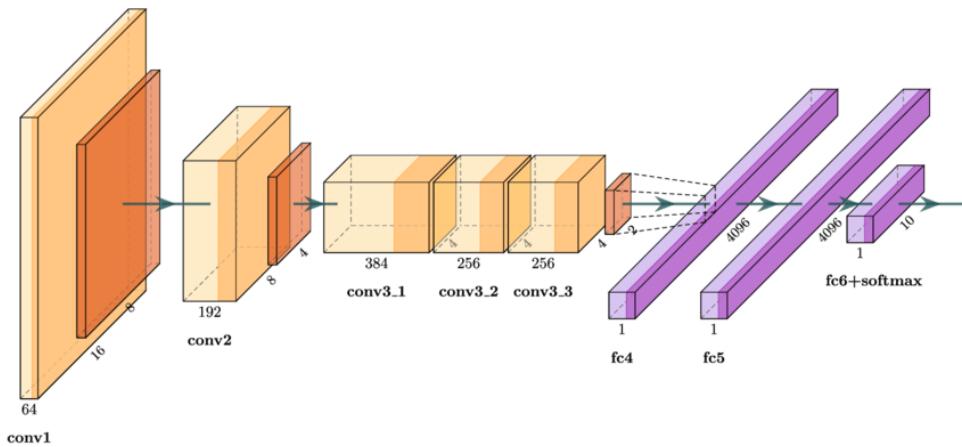


Projet Kaggle

Compte rendu

Ines BESBES



Résumé

Ce projet Kaggle vise à comparer les performances de différents modèles d'apprentissage profond pour la classification d'images sur un sous-ensemble du jeu de données ImageNet. Dans un premier temps, on travaillera sur un problème de classification en 4 classes. Les modèles avec lesquels on va travailler sont LeNet et AlexNet, qu'on essaiera d'optimiser pour éviter le surapprentissage. En effet, on a préféré étudier les modèles LeNet et AlexNet car cela permet de comprendre l'évolution historique des CNN. LeNet a été développé en 1998 et représente les débuts des CNN, tandis qu'AlexNet a été créé en 2012, et a surpassé largement les performances des modèles précédents sur le défi ImageNet. Ainsi, comparer ces deux modèles met en lumière l'impact du passage à des architectures plus complexes. Dans un deuxième temps, une étude sur le biais en Intelligence Artificielle est incluse, où un dataset biaisé est créé. Deux modèles sont comparés : l'un entraîné sur le dataset original et l'autre sur le dataset biaisé. Les performances et le biais des modèles sont évalués à l'aide de la métrique DI (Disparate Impact) et des scores de précision.

Table des matières

1	Entraînement de modèles CNN de pointe	3
1.1	Implémentation des modèles LeNet, AlexNet	3
1.2	Prétraitement des images	3
1.3	Entraînement des modèles par descente de gradient stochastique par mini-lots	3
1.4	Optimisation des hyperparamètres sur un ensemble de validation	5
1.5	Résultats des modèles sur le dataset de validation	5
1.5.1	LeNet régularisé	5
1.5.2	AlexNet régularisé	7
2	Etude du biais en Intelligence Artificielle	8
2.1	Introduction au biais en apprentissage automatique	8
2.2	Problème de classification binaire	8
2.3	Construction du dataset biaisé	8
2.4	Évaluation du biais des modèles	9
3	Comparaison des Modèles	9
3.1	Analyse des résultats et métrique DI	9
3.1.1	Cas extrême où $p_0 = 0$ et $p_1 = 1$	9
3.1.2	Cas où $p_0 = 0.2$ et $p_1 = 0.8$	9
3.2	Discussion sur l'impact du biais	10
3.2.1	Modèles biaisés	10
3.2.2	Modèles non biaisés	10
3.3	Etude du biais en littérature	11

1. Entraînement de modèles CNN de pointe

1.1. Implémentation des modèles LeNet, AlexNet

Pour ce projet, on a implémenté en Pytorch deux modèles de réseaux de neurones convolutifs (CNN) : LeNet et AlexNet. Ce sont des modèles conçus pour la classification d'images.

D'une part, le modèle LeNet est composé de cinq couches : deux couches de convolution et trois couches entièrement connectées. La première couche de convolution prend des images en entrée de taille 256x256, applique un noyau de 5x5 avec padding, suivie d'une activation ReLU et d'un max-pooling de 2x2, réduisant les dimensions spatiales. La deuxième couche de convolution, avec 16 filtres, utilise également un noyau de 5x5, une activation ReLU et un max-pooling de 2x2, réduisant encore les dimensions. La sortie est aplatie en un vecteur unique et passe par trois couches entièrement connectées avec des activations ReLU dans les deux premières couches, aboutissant à une dernière couche qui fournit les probabilités de classe.

D'autre part, le modèle AlexNet est plus récent que LeNet et est composé de huit couches principales : cinq couches de convolution et trois couches entièrement connectées. Les couches de convolution appliquent des filtres de différentes tailles (11x11, 5x5, 3x3) avec des activations ReLU et des max-pooling pour réduire les dimensions spatiales. La sortie des couches de convolution est aplatie et passe par trois couches entièrement connectées avec des activations ReLU dans les deux premières couches, culminant dans une dernière couche qui fournit les probabilités de classe.

On verra plus tard que des méthodes de régularisation seront ajoutées au modèle pour éviter le surapprentissage.

1.2. Prétraitement des images

Afin de préparer les images pour l'entraînement des modèles CNN, on a appliqué plusieurs transformations.

Pour commencer, on a effectué un recadrage centré des images pour les ramener à une taille de 256x256 pixels. Cela permet d'uniformiser la taille des images et de se concentrer sur les éléments les plus pertinents. Ensuite, on a converti les images en niveaux de gris, en ne conservant qu'un seul canal d'information au lieu des trois canaux RGB. Cette étape de conversion en noir et blanc a pour but de simplifier les données d'entrée et de réduire la complexité du modèle. Enfin, on a transformé les images en tenseurs PyTorch, ce qui est le format attendu par nos modèles LeNet et AlexNet.

Ces différentes étapes de prétraitement permettent d'obtenir des données d'entrée homogènes et adaptées à l'entraînement des modèles CNN sur le jeu de données ImageNet.

1.3. Entraînement des modèles par descente de gradient stochastique par mini-lots

L'entraînement des modèles a été effectué en utilisant la méthode de descente de gradient stochastique par mini-lots (SGD). Cette méthode est utilisée dans les deux articles de référence sur LeNet et AlexNet. En effet, elle permet de mettre à jour les poids du modèle après avoir évalué un petit sous-ensemble (batch) du jeu de données d'entraînement, accélérant ainsi le processus d'apprentissage et réduisant les exigences de mémoire.

Si on se contente du prétraitement des données qu'on a évoqué ci-dessus, on remarque que notre modèle sera toujours en surapprentissage, et qu'on obtient des scores d'entraînement et de tests qui ne sont pas satisfaisants. Cela peut être du à une régularisation insuffisante ou à un taux d'apprentissage inapproprié.

Pour LeNet, lorsque l'on fixe le taux d'apprentissage à 0.0005, qu'on prend des mini-lots de taille 8, et qu'on choisit 50 époques, on obtient un score d'entraînement qui vaut 84% environ et un score de test d'environ 45%, ce qui témoigne du surapprentissage. De plus, on remarque que notre fonction de perte pour le test croît, ce qui ne correspond pas à nos attentes. En outre, le temps total d'entraînement est

de 176.98 secondes, ce qui est assez rapide.

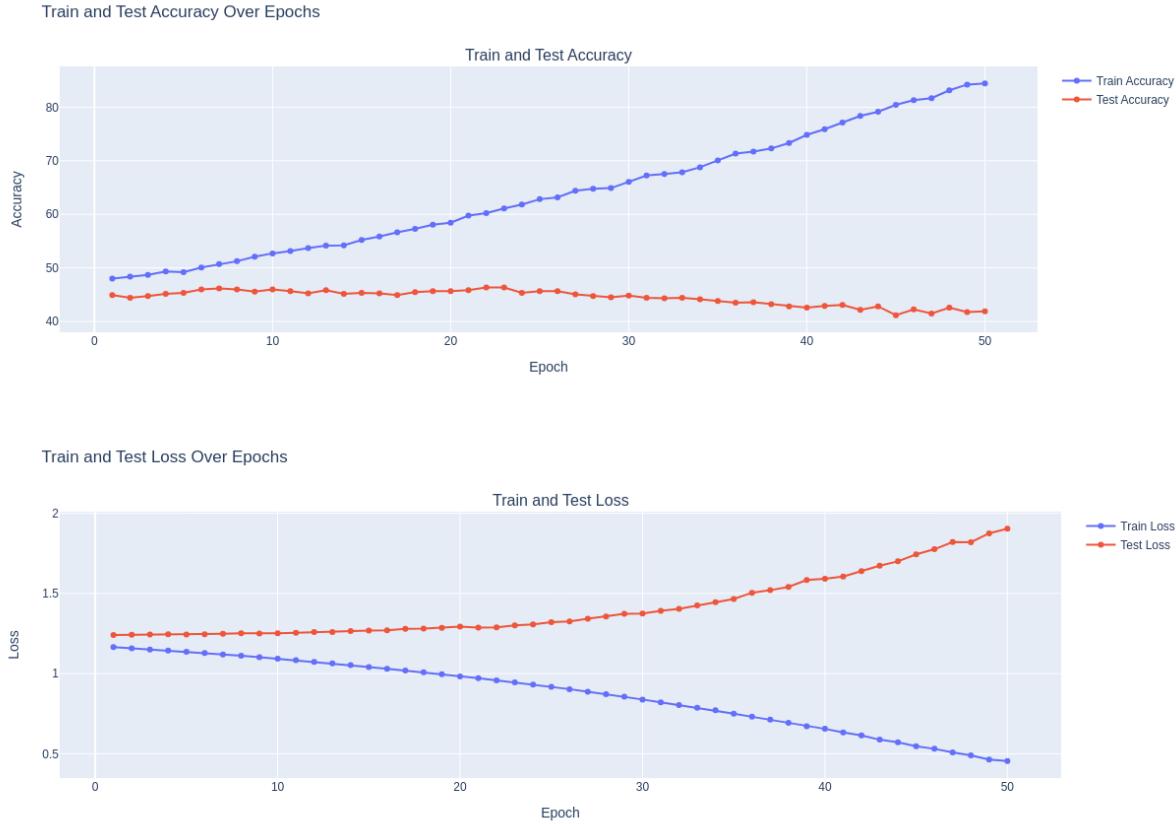


FIGURE 1 – Evolution du score et de la loss en fonction du nombre d'époque avec LeNet

De même, pour AlexNet, on prend plus ou moins les mêmes hyperparamètres que l'article de référence, à savoir un taux d'apprentissage de 0.01, 50 époques et on choisit une taille de mini-lots égale à 64. En conséquence, on observe que le modèle surapprend, et que les scores restent constants, comme le montre la figure ci-dessous.

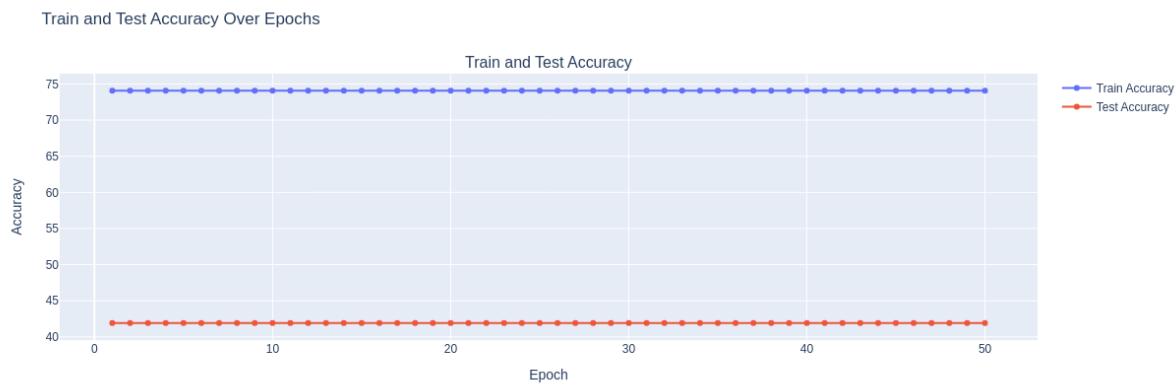


FIGURE 2 – Evolution du score en fonction du nombre d'époque avec AlexNet

En outre, le temps total d'entraînement est de 141.21 secondes. C'est plus rapide que LeNet. En effet, cela peut être du au fait qu'AlexNet possède une structure de couches convolutives et pleinement

connectées qui peut être mieux optimisée pour les calculs parallèles, notamment sur les GPU.

Maintenant, pour pallier au problème du surapprentissage, il faut introduire de nouvelles techniques, comme la régularisation ou la normalisation des données, qu'on verra dans la partie suivante.

1.4. Optimisation des hyperparamètres sur un ensemble de validation

Comme on a vu précédemment, il faut qu'on essaie de modifier notre modèle afin d'obtenir de meilleurs résultats. On va donc créer de nouveaux modèles *ModifiedAlexNet* et *ModifiedLeNet* qui intègrent des techniques de normalisation, l'utilisation de DropOut et du scheduler.

En ce qui concerne le nouveau modèle LeNet, deux couches de normalisation par lot (`self.bn1`, `self.bn2`) ont été ajoutées pour régulariser les activations intermédiaires, contribuant ainsi à la stabilité de l'entraînement et à la prévention du surapprentissage. Une couche de pooling adaptative moyenne (*Adaptive Average Pooling*) a été ajoutée avant les couches entièrement connectées afin de garantir que les activations des convolutions ont toujours une taille de 5×5 avant d'entrer dans la première couche entièrement connectée (`self.fc1`). Ensuite, on a ajouté des couches de dropout (`self.dp1`, `self.dp2`) après les couches entièrement connectées. Cela permet de désactiver de manière aléatoire une fraction de neurones, et empêche donc le modèle de trop s'ajuster aux données d'entraînement. Enfin, on a utilisé un scheduler de taux d'apprentissage (*ReduceLROnPlateau*) afin de réduire le taux d'apprentissage lorsque le modèle stagne en performance, favorisant ainsi une meilleure convergence.

De même, on a ajouté six couches de normalisation par lot pour le nouveau modèle AlexNet. On a également modifié la fonction d'activation en `LeakyRelu` au lieu de `Relu`. Le `Relu` standard souffre du problème des "neurones morts" (*dying ReLU problem*) où certains neurones deviennent complètement inactifs pendant l'entraînement, car leur sortie est toujours nulle pour les entrées négatives. En effet, le `LeakyRelu` tente de résoudre ce problème en attribuant une petite pente non nulle aux valeurs d'entrée négatives. Cela permet d'éviter que des neurones deviennent complètement inactifs. On a aussi appliqué une batch normalization après chaque couche de convolution pour stabiliser les distributions des activations, rendant l'apprentissage plus stable et plus rapide.

Pour chacun des deux modèles, on choisit un taux d'apprentissage de 0.01 car il est suffisamment grand pour apporter des mises à jour significatives aux poids, mais pas trop grand pour faire diverger le modèle pendant l'entraînement. On décide également de définir un momentum de 0.9 pour accélérer l'optimisation dans la direction pertinente et aider le modèle à converger plus rapidement.

En outre, pour LeNet, on décide d'appliquer une régularisation par poids L2 car le modèle surapprend. Cela va permettre de pénaliser les grands poids, encourageant le modèle à trouver des solutions plus simples qui généralisent mieux. On rajoute donc un *weight decay* égal à 0.0001. De plus, on a remarqué que notre modèle ne convergeait plus au bout d'un certain nombre d'époques, et qu'on obtenait toujours le même score. On a donc décidé d'appliquer un scheduler `ReduceLROnPlateau`. Ce dernier adapte le taux d'apprentissage en fonction de la perte de validation. Si la perte atteint un certain niveau, le taux d'apprentissage est réduit, permettant au modèle d'affiner et de trouver un meilleur minimum. Cet ajustement dynamique aide à obtenir une meilleure généralisation sans avoir à intervenir manuellement et faire plusieurs tests. Cela nous fait gagner beaucoup de temps.

D'autre part, pour AlexNet, on a choisi le scheduler `StepLR` pour diminuer systématiquement le taux d'apprentissage à intervalles réguliers (tous les 10 époques). On a remarqué que notre score de test était constant tout au long de l'apprentissage. Or, en réduisant le taux d'apprentissage d'un facteur de 0,1, le modèle peut effectuer des ajustements plus importants initialement, puis des ajustements plus fins au fur et à mesure de l'entraînement. Cela aide à affiner les poids et à améliorer la précision au fil du temps.

1.5. Résultats des modèles sur le dataset de validation

1.5.1 LeNet régularisé

Grâce à l'optimisation des hyperparamètres et à l'utilisation de méthodes de régularisation (qui empêchent le modèle de trop bien apprendre les données d'entraînement) et de normalisation (qui standardisent les données), on arrive à obtenir un modèle avec de bonnes performances, que ce soit sur les

données d'entraînement ou de test.



FIGURE 3 – Evolution du score en fonction du nombre d'époque avec LeNet Modifié

Cela signifie que le modèle n'a plus de problème de surapprentissage, c'est-à-dire qu'il n'apprend pas trop par cœur les données d'entraînement au détriment de sa capacité à généraliser à de nouvelles données. Le modèle arrive donc à bien apprendre sans converger trop rapidement ou trop lentement. De plus, on remarque que cette fois-ci nos deux fonctions de perte décroissent, ce qui indique que le modèle apprend de plus en plus et fait moins d'erreurs de prédiction.

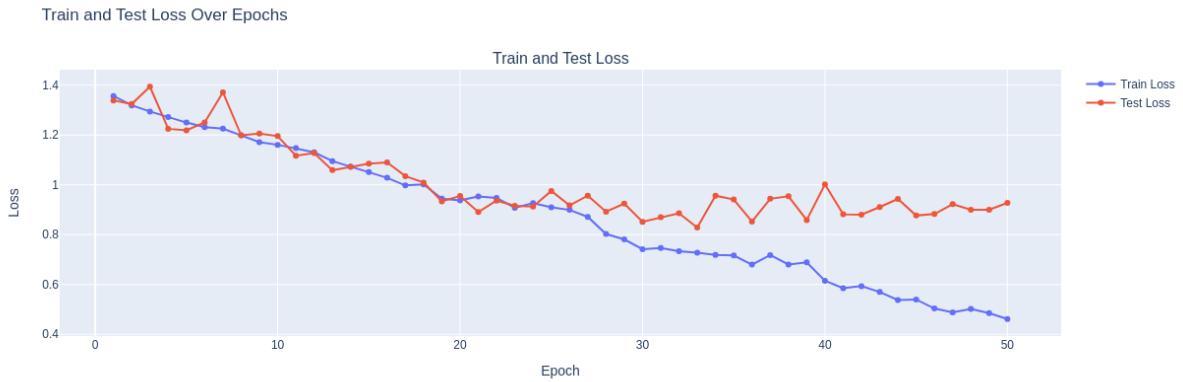


FIGURE 4 – Evolution de la loss en fonction du nombre d'époque avec LeNet Modifié

Enfin, on observe que le temps total d'entraînement est de 179.25 secondes. Le modèle régularisé est un peu plus long que celui de base, ce qui semble logique car on a rajouté des couches comme la normalisation des batchs qui nécessite des calculs additionnels pour normaliser les activations.

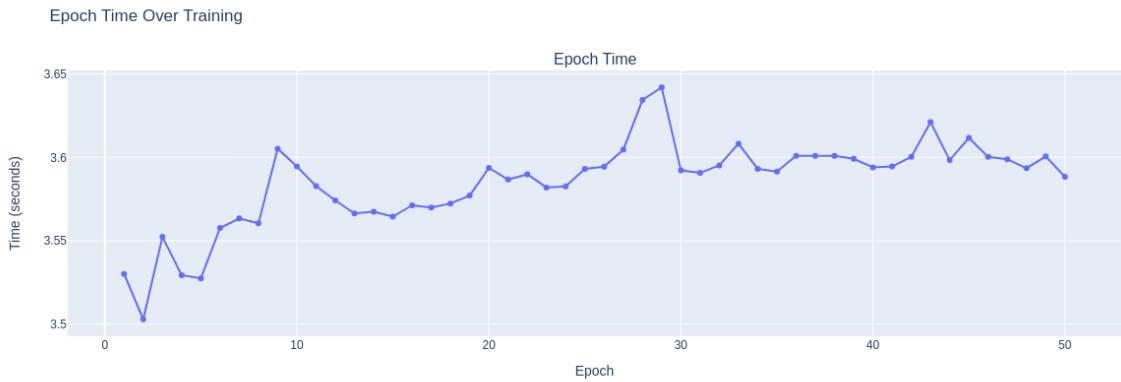


FIGURE 5 – Evolution du temps d’entra nement fonction du nombre d’ poque avec LeNet Modifi 

1.5.2 AlexNet r gularis 

De m me, on obtient de tr s bons r sultats avec le nouveau mod le AlexNet apr s avoir appliqu  les normalisations et les r gularisations.

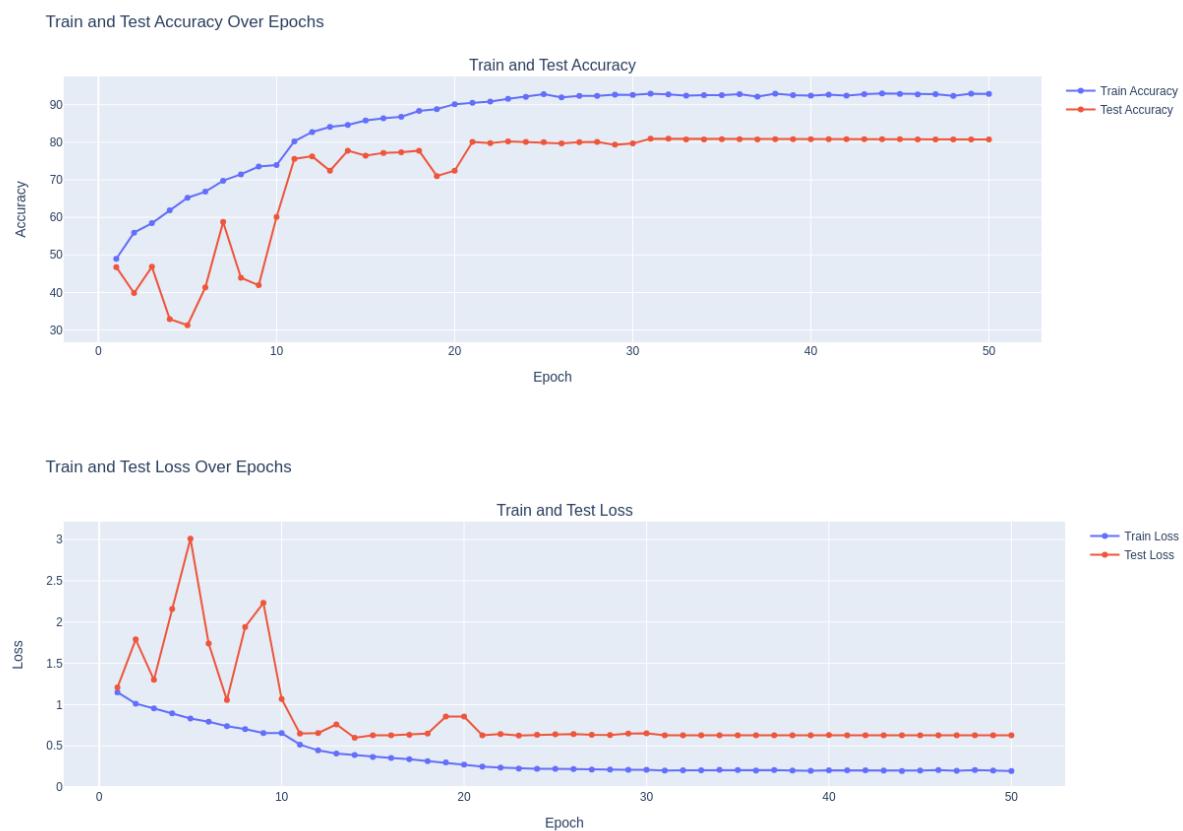


FIGURE 6 – Evolution du score et de la loss en fonction du nombre d’ poque avec AlexNet Modifi 

Le mod le est plus long que le premier et on a un temps total d’entra nement qui vaut 257.63 secondes pour les m mes raisons que LeNet.

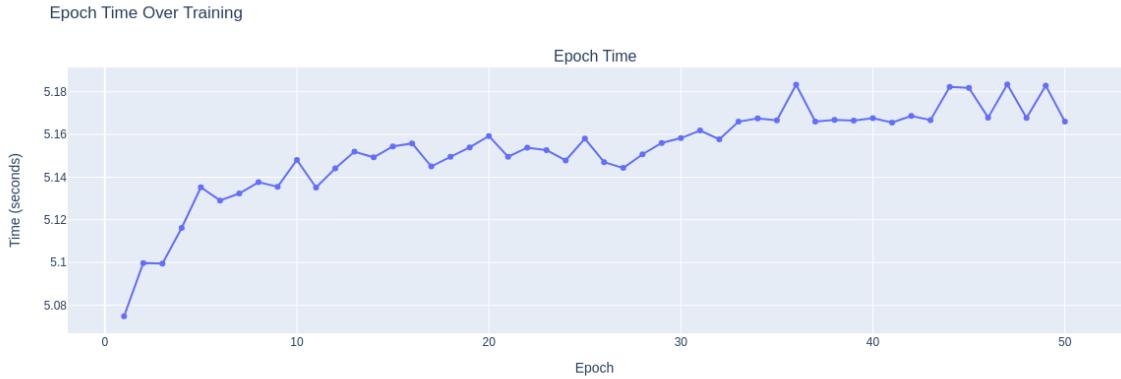


FIGURE 7 – Evolution du temps d’entraînement fonction du nombre d’époque avec AlexNet Modifié

2. Etude du biais en Intelligence Artificielle

2.1. Introduction au biais en apprentissage automatique

Dans cette partie, on va introduire le concept du biais. Le biais en apprentissage automatique se réfère à la tendance d’un modèle à faire des prédictions erronées en raison de données d’entraînement non représentatives ou biaisées. Dans notre cas, on va introduire du biais artificiel pour étudier son effet sur les modèles, et plus précisément sur leurs scores.

2.2. Problème de classification binaire

Dans le code fourni, le jeu de données initial contenant 4 classes d’étiquettes est filtré pour ne conserver que les images appartenant aux classes 1 et 3. Ensuite, les étiquettes sont renumérotées de sorte que la classe 1 devienne la classe 1 et la classe 3 devienne la classe 2. Cela permet de transformer le problème de classification à 4 classes en un problème de classification binaire à 2 classes. Le nouveau jeu de données filtré et recodé est ensuite enregistré dans le fichier "filtered.csv" pour être utilisé dans la suite du projet.

Ensuite, on sépare le nouveau jeu de données en deux sous-ensembles de test et d’entraînement, qui contiennent des données non biaisées. La création du jeu de données d’entraînement biaisé sera expliquée dans les sections suivantes.

D’autre part, avant d’entrainer les modèles, on effectue un prétraitement des données qui est le même que celui dans la première partie, à l’exception que l’on rajoute de la normalisation qui est une forme de régularisation et qui permet d’éviter le surapprentissage. Elle permet également d’éviter les problèmes de stabilité numérique.

2.3. Construction du dataset biaisé

Pour comprendre l’impact de l’introduction du biais, on a construit un dataset biaisé qui correspond au dataset initial mais en y ajoutant une variable ε , qui est une image ressemblant à du bruit, et qui est fortement corrélée aux étiquettes des images (0 ou 1).

Concrètement, on a modifié notre classe `ButterflyDataset` pour pouvoir ajouter ce biais. La variable ε est générée en fonction de la variable de biais S , qui est définie par une distribution de Bernoulli avec une probabilité dépendante de l’étiquette de l’image. Lorsque S vaut 0, ε est un tenseur de zéros, et lorsque S vaut 1, ε est une image de bruit générée selon une distribution normale $\mathcal{N}(0, \mathbf{I})$. Plus formellement, on a :

$$S \sim \text{Bernoulli}(p_y)$$

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

où p_y est la probabilité de Bernoulli qui dépend du label y de l'image.

En utilisant ce dataset modifié, on peut étudier dans quelle mesure le modèle de classification repose sur le contenu de l'image originale par rapport au biais introduit par ε . Dans le cas extrême où $p_0 = 0$ et $p_1 = 1$, le modèle pourrait prédire y en se basant uniquement sur ε sans utiliser l'information de l'image originale x .

Au niveau du code, cela implique de modifier la classe `ButterflyDataset`, et plus précisément la méthode `__getitem__` de cette classe. Dans cette dernière, on utilise la fonction `torch.cat((img, epsilons), dim=0)` pour superposer l'image correspondant au biais (ε) avec l'image originale (d'où $dim=0$). On a également développé d'autres fonctions au sein de la classe pour :

- Générer les valeurs de S à l'aide de la fonction `generate_S`.
- Générer les valeurs de ε à l'aide de la fonction `generate_epsilon`.

Dans la partie qui suit, on va étudier l'impact de ce biais sur les performances de nos modèles.

2.4. Évaluation du biais des modèles

Pour chaque CNN, deux modèles ont été entraînés : l'un sur le dataset original et l'autre sur le dataset biaisé. On a utilisé la métrique de Disparate Impact (DI) pour évaluer le biais des modèles. Cette métrique compare les taux de prédictions positives entre les groupes biaisés et non biaisés, elle est définie comme suit :

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

où : \hat{y} est la prédiction du modèle et S est la variable de biais.

3. Comparaison des Modèles

3.1. Analyse des résultats et métrique DI

Pour le reste du projet, on décide de continuer avec les modèles modifiés, avec leurs paramètres optimaux et on fixe la taille du mini-lots à 32.

3.1.1 Cas extrême où $p_0 = 0$ et $p_1 = 1$

Quand on fixe $p_0 = 0$ et $p_1 = 1$, le DI vaut 0 car on prédit toujours 0 et jamais 1. En effet, $P(\hat{y} = 1|S = 0)$ vaut 0 dans ce cas d'où le résultat (si on inverse le numérateur et le dénominateur, on obtient l'infini).

3.1.2 Cas où $p_0 = 0.2$ et $p_1 = 0.8$

A présent, on fixe $p_0 = 0.2$ et $p_1 = 0.8$. En d'autres termes, lorsque l'étiquette est 0, il y a une probabilité de 0.2 que ϵ soit un tenseur non nul (bruit). Lorsque l'étiquette est 1, cette probabilité est de 0.8.

Cette configuration introduit un biais dans le jeu de données, car ϵ est fortement corrélé avec l'étiquette y . Le modèle pourrait alors apprendre à prédire y en se basant principalement sur ϵ plutôt que sur le contenu de l'image.

En comparaison, dans le jeu de données de test non biaisé (`test_dataset_non_biased`), p_0 et p_1 sont tous deux fixés à 0.5, ce qui signifie que ϵ suit une loi de Bernoulli équilibrée, indépendamment de l'étiquette y .

D'abord, on entraîne LeNet sur un dataset biaisé et on obtient un DI qui est égal à 1.5, ce qui signifie que le modèle est biaisé. De même, pour AlexNet, on obtient un DI égal à 2.28, indiquant que le modèle est biaisé. Ainsi, les modèles entraînés sur le jeu de données biaisé (LeNet et AlexNet) présentent également

un biais important, ce qui signifie qu'ils ne généralisent pas bien et s'appuient trop sur le bruit ϵ pour faire leurs prédictions, plutôt que sur les véritables caractéristiques des images.

3.2. Discussion sur l'impact du biais

3.2.1 Modèles biaisés

Lorsqu'on entraîne nos modèles sur le dataset biaisé, et qu'on teste sur le jeu de données non biaisé, on obtient un score de test égal à 63% pour LeNet et 50% pour AlexNet. De fait, le modèle LeNet a mieux réussi à s'adapter au biais du dataset d'entraînement que le modèle AlexNet.



FIGURE 8 – Evolution du score en fonction du nombre d'époque (LeNet en haut, AlexNet en bas)

Cela indique que les modèles ont du mal à bien généraliser sur le jeu de données de test, malgré un entraînement sur le dataset biaisé. En effet, cela s'explique par le fait que le biais introduit dans le dataset d'entraînement a fortement influencé l'apprentissage du modèle. Au lieu d'apprendre à se baser sur les caractéristiques visuelles pertinentes des images, le modèle a probablement appris à se fier principalement à la variable de biais ϵ pour faire ses prédictions.

3.2.2 Modèles non biaisés

Dans le cas contraire, lorsqu'on entraîne nos modèles sur un jeu de données d'entraînement non biaisé, et qu'on les teste sur un jeu de données de test non biaisé, on obtient de bons résultats :

- Pour le modèle LeNet, on obtient un score de **70%** sur le jeu de test non biaisé.
- Pour le modèle AlexNet, on obtient un score de **80%** sur le jeu de test non biaisé.

Cela se traduit par de bons scores de performance pour ces deux modèles, comme on peut le voir dans les figures correspondantes.

Cela montre que lorsque les données d'entraînement et de test sont non biaisées, les modèles arrivent à bien apprendre les caractéristiques pertinentes des images, sans se fier uniquement au bruit corrélé avec

les étiquettes. Ils peuvent alors généraliser efficacement à de nouvelles données non biaisées. En résumé, l'absence de biais dans les jeux de données d'entraînement et de test permet d'obtenir des modèles performants et généralisables, contrairement au cas précédent où le biais dans les données d'entraînement entraînait un biais dans les prédictions des modèles.



FIGURE 9 – Evolution du score en fonction du nombre d'époque (LeNet en haut, AlexNet en bas)

3.3. Etude du biais en littérature

On tentera de répondre à la problématique suivante : "Est-ce qu'on peut faire confiance à mon modèle ?" du point de vue du biais des modèles, en s'appuyant sur l'article "Conformité européenne des systèmes d'IA : outils statistiques élémentaires".

L'article soulève des questions essentielles sur la conformité des systèmes d'IA aux futures réglementations européennes, notamment en termes de biais et de discrimination. En tant que concepteur de modèles d'IA, peut-on vraiment faire confiance à ces modèles et les déployer en toute confiance ? Aujourd'hui, des mesures sont prises pour protéger les gens, comme le RGPD (Règlement général sur la protection des données). On vit également dans un monde où les biais et discriminations potentiels des modèles d'IA sont un enjeu majeur. Comme le montre l'exemple d'un score de crédit biaisé, un modèle peut apprendre à prédire la variable cible en se basant principalement sur des variables corrélées mais non pertinentes, comme un bruit aléatoire. Cela conduit à des prédictions biaisées et discriminatoires. Pour faire confiance à un modèle, il faut donc s'assurer rigoureusement qu'il n'est pas biaisé, en analysant en détail les données d'entraînement et les performances, comme ce qu'on a fait précédemment. Il faut faire des tests sur des données non biaisées pour évaluer le biais réel du modèle.

En conclusion, on pense que pour faire confiance à un modèle, il faut d'abord faire en sorte que nos données soient bien et surtout représentatives du problème traité.