CentraleSupélec

Projet ComputerVision & DeepLearning Noise2Noise

Superviseuse: Hudelot Céline

Mots clés et thèmes

ComputerVision, DeepLearning, Noise2Noise, ResNet, U-Net, Filtres, Data augmentation.

Répertoire du code

Le code de mon projet se trouve sur GitLab.

Répertoire du dataset

Le dataset de mon projet se trouve sur Google Drive.

Michot Albane, Nonclercq Rodolphe

16 Avril, 2023

Table des matières

Computer Vision & DeepLearning - Rapport de projet

Abstract

Le but de ce projet du cours de Computer Vision est de poursuivre le projet de DeepLearning par l'application de méthodes de ComputerVision (filtres, implémentation de deux modèles en série, poursuite de la data augmentation par l'implémentation d'un générateur et optimisation de l'entraînement), pour améliorer nos résultats obtenus.
Le projet de DeepLearning consiste à implémenter un modèle similaire au Noise2Noise, un réseau de débruitage d'images entraîné sans image de référence propre. Kaggle mettant à disposition un dataset basé sur le papier original, l'objectif est d'étudier et d'implémenter différentes architectures de réseaux de neurones afin de vérifier les affirmations et résultats du papier original. Le code des auteurs se basant sur TensorFlow, le framework PyTorch est ici utilisé comme alternative.

1. Introduction

1.1. Description

Avec Noise2Noise, les auteurs souhaitent tirer parti des DNN (*deep neural networks*) pour éviter la modélisation de vraisemblance statistique explicite préalable des images bruitées (*noisy images*) et à la place apprendre à mettre en correspondance les images bruitées avec les images propres (*clean images*) non observées. En particulier, le débruitage peut être réalisé en utilisant uniquement des "mauvaises images" ou images bruitées, si le bruit est additif et non biaisé. En d'autres termes, il est possible, à partir des données d'apprentissage sur deux images bruitées de manière indépendante, d'apprendre indirectement le modèle de vraisemblance statistique explicite de la corruption sans se baser sur un modèle de débruitage qui utilise des images propres (*ground truth*).
Formellement, si l'on considère ϵ et δ deux bruits non biaisés et indépendants, les auteurs du papier démontrent que minimiser :

$$\mathbb{E}\left[\|\phi(X+\epsilon;\theta)-(X+\delta)\|^2\right] \tag{1}$$

est équivalent à minimiser :

$$\mathbb{E}\left[\|\phi(X+\epsilon;\theta)-X)\|^2\right] \tag{2}$$

où X est le signal propre, ϵ et δ sont des bruits aléatoires non biaisés et indépendants tels que $\mathbb{E}[\epsilon] = \mathbb{E}[\delta] = 0$ respectivement, $\phi(X+\epsilon;\theta)$ est la réponse du modèle sur le signal bruité avec ϵ.

Par conséquent, il est possible d'entraîner un modèle avec des paires d'échantillons bruitées, s'ils décrivent tous deux les mêmes échantillons propres indisponibles avec différents bruits non biaisés, additifs et indépendants.

L'objectif de ce projet est de montrer que **les architectures développées permettent de débruiter une image corrompue avec un training (entraînement) uniquement basé sur des images bruitées**.

1.2. Présentation du dataset

L'idée de ce projet provient d'une compétition Kaggle dont l'objectif est de construire un réseau de neurones capable de débruiter des images de la même manière que dans le papier original. Ce dataset Kaggle diffère du dataset du papier original : la résolution des images étant plus basse, il permet de réduire le temps d'exécution lors du training.
Le dataset Kaggle est composé de deux fichiers : un fichier *train_data.pkl* constitué de deux tenseurs de taille $50000 \times 3 \times 32 \times 32$, donc 50000 paires d'images bruitées couleur (3 canaux, RGB) indépendantes de taille 32 pixels $\times$ 32 pixels et un fichier *val_data.pkl* contenant deux tenseurs de taille $1000 \times 3 \times 32 \times 32$, donc 1000 paires d'images contenant chacune une image bruitée (bruit différent de ceux des images du *train_data.pkl*) et une image propre (Figure 1).

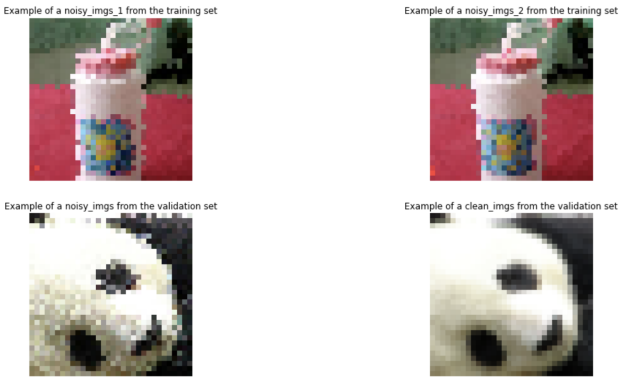

Figure 1: Illustration de la composition du dataset

1.3. Plan mis en place

Afin de répondre au mieux au problème, la démarche suivante est proposée :

1. Présentation et résultats des modèles implémentés
2. Comparaison et discussion avec le modèle discriminant
3. Application de méthodes de Computer Vision
4. Conclusion

2. Présentation et résultats des modèles implémentés

Dans cette section, deux modèles ont été implémentés : un Resnet et un U-Net. Ces deux derniers découlent de l'architecture Noise2Noise du papier original.

2.1. ResNet

2.1.1. Architecture du réseau

Pour l'architecture du premier réseau, nous implémentons une structure similaire au ResNet vu en cours. Sa structure doit prendre en entrée des tenseurs de taille Nx3xHxW, avec N le nombre de paires d'images, et sortir un tenseur de même dimension.

Au début, une structure simple de réseau est mis en œuvre avec uniquement quelques convolutions et des fonctions d'activation simples. L'objectif était de comprendre l'impact des dimensions des couches sur la performance. L'utilisation d'un réseau contenant uniquement des convolutions et des fonctions d'activation Relu conduisait à de mauvais résultats. Changer ReLU par la fonction d'activation Sigmoïde n'a pas amélioré les résulats. ReLU a été ainsi conservée pour l'ensemble des couches ainsi que pour le reste du projet.

Pour améliorer les résultats, des couches MaxPool et Upsample ont été ajoutées mais la couche de Maxpooling avait tendance à augmenter le flou de l'image (image étant déjà en basse résolution) du fait de la réduction de la taille de l'image. En effet, comme l'opération de Maxpooling perd des informations sur les pixels, elle n'était pas adaptée pour cette architecture. Alors, pour réduire cet effet de flou, des *skip connections* (résidus) ont été ajoutés au réseau. Cela permet de ne travailler que sur une partie de l'image, sans trop modifier l'image initiale. Enfin, en ajoutant du *batch normalization*, le temps d'apprentissage diminue, puisqu'il décale et remet à l'échelle les données en fonction de l'entrée de la couche.

Avec le réseau présenté sur la Figure 3, la taille de l'image est toujours la même mais le nombre de canaux est augmenté et diminué à travers des convolutions et convolutions inverses. En commençant par 3 canaux (RGB), une convolution, avec une taille de kernel de 3x3 suivie d'une activation ReLU et d'une batch normalization, augmente le nombre de canaux à 32. Puis, à nouveau, le nombre de canaux est augmenté à 64 et transite par une fonction ReLU. À ce stade, une skip connection est réalisée et le nombre de canaux est augmenté à 96. C'est la "profondeur" maximale atteinte par le réseau. De nouveau, des convolutions sont effectuées, suivies d'une activation ReLU, puis la skip connection est terminée. Le processus de convolution, ReLU et parfois batch normalization est répété plusieurs fois jusqu'à nouveau obtenir 3 canaux. À la fin, un clamping entre 0 et 1 a été ajouté pour que les pixels de la sortie soient dans la bonne gamme de valeurs. Cette architecture est finalement composée de **10 convolutions, 10 activations et 4 batch normalizations**.

2.1.2. Training

Une fois que les différentes couches du réseau ont été choisies, d'autres paramètres peuvent être choisis pour améliorer la performance globale du modèle, basée sur la **métrique PSNR (Peak Signal Noise Ratio)**. La mesure du PSNR, exprimée en décibel (dB) permet de quantifier la performance des modèles en mesurant la qualité de reconstruction de l'image compressée par rapport à l'image propre. Pour pouvoir étudier si le modèle est performant, il suffit de comparer le PSNR du validation set qui calculera le PSNR entre l'image bruitée et la clean image, avec le PSNR après entraînement du modèle qui calculera le PSNR entre la predicted image et la clean image. En conséquence, si un PSNR plus élevé est obtenu après le training du modèle, cela signifie que le signal et donc le traitement de restauration ou d'amélioration sont meilleurs.

Les deux paramètres ayant le plus d'influence sur la qualité du résultat et la vitesse de calcul sont le **nombre d'epochs** et la **taille du batch**. Pour le premier, une augmentation de sa valeur conduit à un temps de calcul trop long et les performances s'améliorent peu. Un nombre de **20 epochs** constituait un bon compromis performance/temps de calcul. Sur la Figure 2, la courbe de la fonction loss semble converger vers un plateau. Cependant, il faudrait augmenter le nombre d'epochs pour observer une véritable convergence.

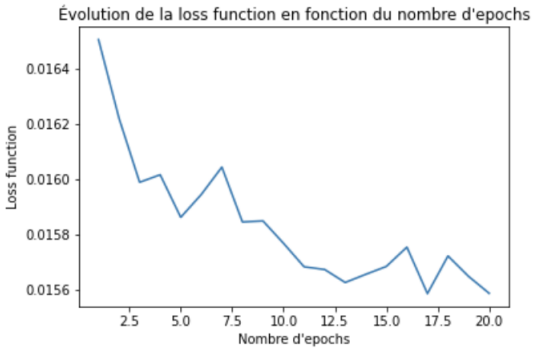

Figure 2: La fonction Loss en fonction du nombre d'epochs

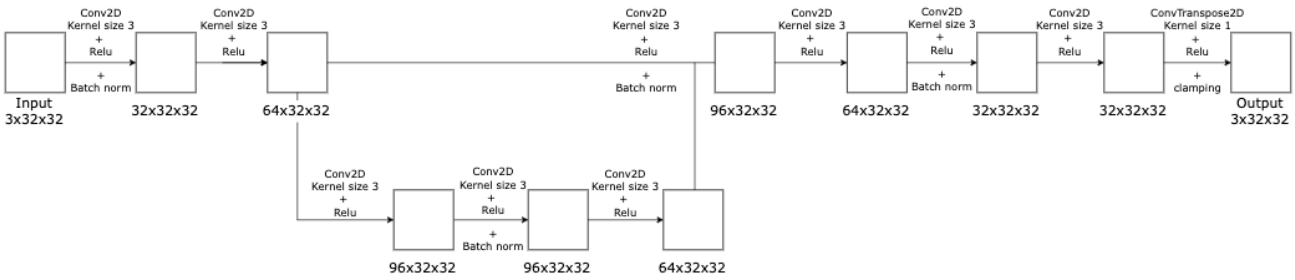

Figure 3: Architecture du réseau ResNet implémenté

Ensuite, la taille du batch est un paramètre important pour le temps de calcul. En effet, cette valeur affecte le nombre d'images entraîné en même temps. Avec une valeur trop faible (par exemple 50), le temps d'apprentissage est trop long car le processus est répété plus de fois. Avec une valeur trop élevée, la taille de la mémoire GPU sature car il est difficile de stocker un grand nombre d'images sur le GPU si la configuration matérielle ne le permet pas. Toutefois, du côté du PSNR, les résultats varient peu. En effet, une taille de batch plus importante ne permet généralement pas d'obtenir une précision élevée, l'optimiseur et le taux d'apprentissage ayant également un impact important sur le résultat. "La réduction du taux d'apprentissage et la diminution de la taille des lots permettront au réseau de mieux s'entraîner, en particulier dans le cas d'un réglage fin." [5]. **Une taille de batch de 100** a été conservée par la suite.

Pour évaluer notre réseau, **l'erreur quadratique moyenne (Mean Squared Error, MSE)** a été sélectionnée. En effet, nous pouvons faire l'hypothèse que le bruit suit une distribution normale alors que l'entropie croisée (Cross-Entropy) suppose généralement une distribution binomiale (mieux adaptée à la classification, pas à la régression comme dans notre cas).

Les images bruitées mises en entrée sont utilisées pour entraîner le réseau avec **l'optimiseur ADAM (Adaptative Moment Estimation)** car il est approprié pour les problèmes avec des gradients très bruités et/ou épars. De plus, il est plus efficace en termes de calcul que le SGD dans ce cas, ce qui permet d'avoir des besoins faibles en mémoire. Pour trouver un taux d'apprentissage (learning rate) correct, nous commençons par un taux d'apprentissage plus élevé ($\alpha = 0.1$) pour sortir des minima locaux et nous le diminuons ensuite à une valeur très faible pour être plus précis. Finalement, le réseau est entraîné avec un **taux d'apprentissage de 0,001**, offrant ainsi un bon compromis entre la performance et la précision (à $\sim 1e-4$, l'entraînement progresse très lentement et à $\sim 1e-2$, il donne de mauvais PSNR).

Dans les DNN comportant de nombreuses couches de convolution et différents chemins à travers le réseau, une **initialisation correcte des poids est extrêmement importante** (les poids évoluent au même rythme dans toutes les couches pendant le training). Sinon, des sections du réseau pourraient fournir des activations excessives, tandis que d'autres sections ne contriburaient jamais. Utilisant le framework de Pytorch, nous prenons la décision de ne pas modifier l'initialisation par défaut. En effet, puisque des couches de convolution et des fonctions d'activation ReLU sont principalement utilisées, la méthode uniforme de Kaiming est utilisée par défaut où les poids suivent la règle :

$$f : \mathbb{R}^N \to \mathbb{R}^M$$

$$w_{i,j} \sim \mathbb{U}\left[-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}\right] \tag{3}$$

2.1.3. Résultats

Après ces implémentations, un **PSNR moyen de 25.21 dB** est obtenu entre la *predicted_image* et la *clean_image* **contre 20.72 dB** entre l'image bruitée du validation set et la *clean_image* (avant training).

Une fois que le réseau a été mis en place, une **augmentation des données a été effectuée** pour améliorer le PSNR. En effet, mettre en place des méthodes d'augmentation des données permet d'améliorer la généralisation.
L'idée est de créer de fausses données annotées et les ajouter à l'ensemble d'apprentissage. Pour cela, les méthodes d'augmentation *flip horizontal* et *flip vertical 180°* ont été générées à partir du notebook *Data_augmentation.ipynb* (Figure 4).

Figure 4: Illustration de l'augmentation des données sur le training set

Ces deux méthodes ont été choisies car elles ne modifiaient pas l'image représentée. Le réseau a donc été **réentraîné avec le nouveau fichier de données d'entraînement généré** *augmented_train_data.pkl* qui comporte 3×50000, soit 150000 paires d'images.

L'augmentation des données a permis d'augmenter notre **PSNR moyen atteignant 25.43 dB contre 25.21 dB** sans l'augmentation des données.

L'entraînement avec l'augmentation des données dure environ 6 minutes sur le GPU de GoogleColab. Le résultat d'une image prédite de l'ensemble de données comparée à l'image d'entrée et à l'image propre peut être observée sur la Figure 5.

Figure 5: Comparaison de l'image prédite (c) avec l'image d'entrée (a) et l'image propre (b)

2.2. U-Net

2.2.1. Architecture du réseau

Pour l'architecture du second réseau, le but était d'étudier une autre architecture en proposant un **réseau similaire au U-network** [3], proposé dans le papier original, où le nombre de canaux d'entrée et de sortie est $n = m = 3$. Nous entraînons le réseau en utilisant 32×32-pixel crops tirées des 50000 paires d'images du training set. Dans l'article original, le U-network est entraîné sur 256 × 256-pixel crops et est conçu en conséquence avec des couches multiples. Dans notre cas, nos images sont 8 fois plus petites et nous décidons de proposer une version plus compacte du U-network. Le réseau prend en entrée un tenseur de 50000 × 3 × 32 × 32 et sort un tenseur avec les mêmes dimensions.

Le U-Net est construit sur un **chemin de contraction** et un **chemin d'expansion**. Le chemin de contraction suit l'architecture typique d'un CNN. Deux convolutions 3x3 (convolutions sans padding) sont appliquées de manière répétitive, chacune étant suivie d'une fonction d'activitation ReLU et d'une average pooling 2x2 pour le downsampling. À chaque étape de downsampling, le nombre de canaux des features est doublé. Dans la résolution la plus basse, les dimensions des images sont réduites à 6 pixels dans les couches d'average pooling. Le chemin d'expansion est composé d'une convolution transposée 3x3 qui divise par deux le nombre de canaux des features, suivie d'un upsampling de la feature map et d'une concaténation avec la feature map recadrée du chemin de contraction. Ensuite, une autre couche de upsampling est appliquée, suivie de deux convolutions 3x3, chacune suivie d'une ReLU. Dans la dernière couche, une convolution 3x3 et une convolution transposée sont utilisées pour récupérer finalement les dimensions des images 32x32. Des skip connections sont réutilisées ici pour relier les données d'entrée et la couche de reconstruction finale. Ces skip connections permettent de récupérer la perte d'information pendant le downsampling comme le max pooling et également au réseau d'apprendre une combinaison de features haut niveau et bas niveau.
Au total, le réseau comporte **8 couches de convolution** (Table 1).

Name	N_{out}	Function
INPUT CONV3_32	32	Convolution 3x3
CONV32_64	64	Convolution 3x3
POOL2_2_1	64	Avgpool 2x2
CONV64_64	64	Convolution 3x3
POOL2_2_2	64	Avgpool 2x2
UPSAMPLE2_1	64	Upsample 2x2
T_CONV64_64	64	Convolution 3x3
CONCAT1	64	Concatenate output
UPSAMPLE2_2	128	Upsample 2x2
T_CONV128_64	64	Convolution 3x3
T_CONV64_32	32	Convolution 3x3
CONV35_32	32	Convolution 3x3
T_CONV32_3	3	Convolution 3x3

Table 1: Architecture du réseau U-Net implémenté

Initialement, une batch normalization a été mise après chaque couche de convolution (pour forcer les statistiques d'activation pendant forward pass en les re-normalisant) en pensant augmenter le PSNR car cela peut aider à prévenir le phénomène de la dégradation de la précision, qui peut se produire lorsque le réseau devient trop profond. Cela peut également aider à réduire les effets de la saturation des fonctions d'activation et de l'explosion du gradient. Toutefois, dans notre cas, le batch normalization avait un impact uniquement au milieu du réseau. Nous avons finalement

conclu pour ce réseau que la procédure PyTorch de normalisation par défaut du framework compensait dans une certaine mesure **l'absence de batch normalization**.

2.2.2. Training

Pour choisir une valeur raisonnable pour le nombre d'epochs, nous décidons de tracer la fonction loss en fonction du nombre d'epochs (Figure 6). Comme attendu, nous observons une décroissance de la fonction loss lorsque le nombre d'epochs augmente. Par conséquent, nous utilisons un **nombre d'epochs de 20** qui offre un bon compromis entre le temps de calcul et la précision.

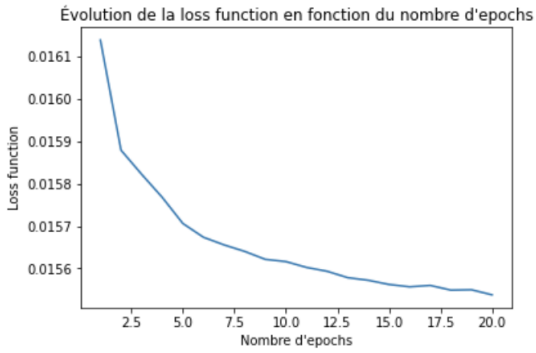

Figure 6: La fonction Loss en fonction du nombre d'epochs

Pour la taille du batch, nous commençons avec une batchsize de 1000 et trouvons progressivement un très bon compromis avec les autres paramètres pour une **batchsize de 100**.

Comme pour le réseau ResNet implémenté, le training est réalisé avec le **loss criterion MSE**, **l'optimiseur ADAM** et un **taux d'apprentissage de 0,001**.
De plus, nous ne modifions pas également **l'initialisation des poids** par défaut.

2.2.3. Résultats

Avec 50000 échantillons d'entraînement, 20 epochs sur le dataset complet, des batchs de 100, l'utilisation de l'optimiseur ADAM et le choix du loss criterion MSE, les performances du modèle sur 1000 échantillons du validation set donnent un **PSNR moyen de 25,34 dB**.

Une fois que le réseau a été mis en place, une augmentation des données a été effectuée pour améliorer le PSNR.
Le réseau a donc été **réentraîné avec le nouveau fichier de données d'entraînement généré** *augmented_train_data.pkl* qui comporte 150000 paires d'images.
L'augmentation des données a permis d'augmenter notre **PSNR moyen atteignant 25.50 dB contre 25.34 dB** sans l'augmentation des données.

L'entraînement avec l'augmentation des données dure environ 13 minutes sur le GPU de GoogleColab.

Le résultat d'une image prédite de l'ensemble de données comparée à l'image d'entrée et à l'image propre peut être observé sur la Figure 7.

Figure 7: Comparaison de l'image prédite (c) avec l'image d'entrée (a) et l'image propre (b)

2.2.4. Discussion

Avec notre architecture U-Net, nous parvenons à obtenir une bonne performance globale, qui dépasse celle du ResNet. En effet, nous avons commencé par créer le U-Net complet proposé dans l'article original. Nous avons rapidement été limités par des dimensions trop faibles du fait que les images de l'article original étaient des images de 256x256. Trop de couches ont en effet prouvé qu'un réseau plus profond n'est pas nécessairement un "meilleur" réseau.

Nous avons obtenu de mauvais résultats avec les couches Maxpool, car cette opération tend à perdre des informations sur les pixels, puisqu'elle renvoie la valeur maximale de la partie de l'image couverte par le noyau. De plus, les opérations de pooling ont un effet de flou sur les images. Nous avons résolu ce problème avec des techniques d'augmentation des données et par l'ajout des skip connections, pour ne pas trop altérer l'image initiale.

3. Comparaison et discussion avec le modèle discriminant

3.1. Expérimentation

Le papier original montre qu'en restaurant des images en ne regardant que des **exemples corrompus produit des performances égales, voire supérieures, à celles obtenues en utilisant des données propres**. Ainsi, il serait possible d'avoir une performance plus élevée en utilisant un réseau Noise2Noise plutôt qu'en utilisant un débruiteur avec un training discriminant (avec une image propre).

Il est ainsi **intéressant de pouvoir comparer un réseau discrimant** utilisant en entrée une image propre et une image bruitée, **avec notre réseau Noise2Noise U-Net** ayant eu la meilleure performance.
Pour cela, l'architecture de notre réseau U-Net est réutilisée et est, dans cette partie, entraînée sur le validation set

(possédant entre autres l'image propre).

Un des problèmes rencontrés est le **nombre faible de paires d'images** pour entraîner notre modèle (1000 paires d'images pour le validation set). Nous avons pour cela appliquer notre fichier *augmented_val_data.pkl* que l'on a pu générer grâce au notebook *Data_augmentation.ipynb* sur les 1000 paires d'images du validation set afin de générer **3000 paires d'images sur lequel le réseau est entraîné**.

3.2. Résultats

La fonction loss obtenue stagne rapidement à partir de 10 epochs (Figure 8), mais par souci de comparaison, nous gardons les mêmes hyper-paramètres que le réseau Noise2Noise U-Net, à savoir **20 epochs**.

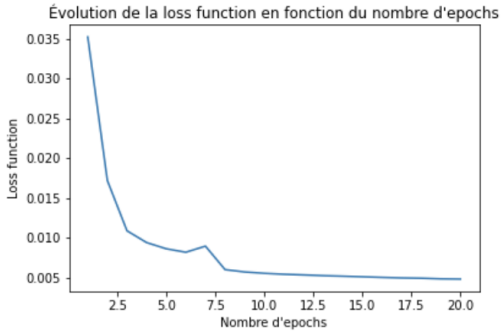

Figure 8: La fonction Loss en fonction du nombre d'epochs

Après ces implémentations, un **PSNR moyen de 21.96 dB** est obtenu entre la predicted image et une des images bruitées **contre 17.88 dB** entre les deux images bruitées du train set (avant training).
Le modèle améliore le débruitage. Néanmoins, il est **difficile de comparer avec le modèle Noise2Noise U-Net** car ce ne sont pas les mêmes images entraînées, le validation set et le training set ont été interchangés. De plus, notre Noise2Noise U-Net est entraîné sur 50000 paires d'images contre 3000 pour le modèle discriminant. Il serait intéressant d'avoir les 50000 images clean du training set pour pouvoir réaliser une véritable comparaison.

L'entraînement dure environ 1 minute sur le GPU de GoogleColab, en raison du plus petit nombre d'images entraînées.

Le résultat d'une image prédite de l'ensemble de données comparée aux images d'entrée bruitées peut être observée sur la Figure 9.

Figure 9: Comparaison de l'image prédite (c) avec la première image d'entrée (a) et la deuxième image d'entrée (b)

4. Application de méthodes de Computer Vision

Dans le cadre de notre projet de débruitage d'images, d'autres méthodes de Computer Vision ont été appliquées pour **améliorer notre PSNR moyen final**.
Dans cette section, trois méthodes ont été implémentées: **méthodes de filtrage, implémentation de deux modèles en série, poursuite de l'augmentation des données par l'implémentation d'un générateur et optimisation de l'entraînement**.
Pour cette partie, uniquement le réseau U-Net avec data augmentation, ayant le meilleur PSNR moyen obtenu, est conservé pour appliquer les deux premières méthodes et pour les dernières méthodes le réseau U-Net est également conservé mais le générateur est appliqué sur le fichier train initial *train_data.pkl*.

4.1. Méthodes de filtrage

Dans les domaines du Deep Learning et de la Computer Vision, les filtres sont des outils importants pour extraire les caractéristiques des images.

Dans le processus de training d'un modèle de Deep Learning, il est courant d'appliquer plusieurs filtres différents sur les images d'entrée afin de capturer les différentes caractéristiques de l'image.

Pour pouvoir évaluer l'efficacité de ces filtres, il est nécessaire de les **appliquer uniquement sur les données d'entrée** du train set (*noisy_imgs_1*) et du validation set (*noisy_imgs*). Le résultat de cette opération est la création de deux nouveaux fichiers pickle pour chaque filtre appliqué sur l'image d'entrée : un pour le train set, à partir du fichier *augmented_train_data.pkl* et un pour validation set, à partir du fichier *val_data.pkl*. Vous pouvez retrouver l'ensemble des fichiers pickle créés sur le Google Drive sous les appellations *augmented_train_data_filtered_[nom_du_filtre].pkl* et *val_data_filtered_[nom_du_filtre].pkl*, ainsi que l'ensemble des résultats et implémentation des filtres sur le notebook *filters.ipynb* dans le dossier *Preprocessing*.

Pour notre projet de débruitage d'images à basse résolution, nous avons appliqué **six filtres** différents sur nos données d'entrée pour tenter d'**améliorer notre PSNR moyen**

final après l'entraînement de notre réseau U-Net. Nous avons divisé ces filtres en trois parties : les filtres de débruitage, les filtres d'amélioration de la netteté et les filtres spécifiques pour les images à faible résolution.

4.1.1. Filtres de débruitage

Tout d'abord, nous avons appliqué des filtres de débruitage classiques tels que les filtres médian, gaussien et bilatéral.

Le filtre médian

Le filtre médian est une technique de filtrage non linéaire, utilisé pour réduire le bruit impulsif. Il remplace chaque pixel par la médiane des pixels voisins. (Figure 10)

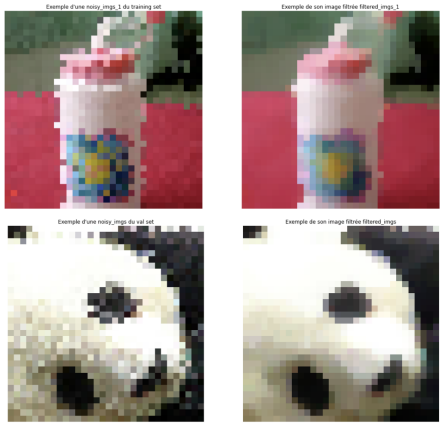

Figure 10: Exemple de l'application du filtre médian sur une image d'entrée du train set (en haut) et du validation set (en bas)

Le filtre gaussien

Le filtre gaussien est une technique de filtrage linéaire, utilisé pour réduire le bruit dans les images. Il remplace chaque pixel par la moyenne pondérée des pixels voisins, où les poids sont déterminés par une fonction Gaussienne (Figure 11).

Le filtre bilatéral

Nous appliquons dans un troisième temps le filtre bilatéral qui est une technique de filtrage non linéaire, utilisée pour réduire le bruit tout en préservant les bords et les détails de l'image. Il remplace chaque pixel par la moyenne pondérée des pixels voisins, où les poids sont déterminés à la fois par une fonction Gaussienne spatiale et une fonction Gaussienne de la différence des couleurs (Figure 12).

Conclusion

Nous avons constaté que l'application de ces filtres n'améliorait pas notre PSNR moyen final après l'entraînement du réseau. Ces trois filtres **débruitent les images et nous perdons ainsi en détail**. Le filtre ayant déjà effectué un débruitage sur nos images, il est probable que certaines informations sur les *features* des images, utiles pour le réseau

Figure 11: Exemple de l'application du filtre gaussien sur une image d'entrée du train set (en haut) et du validation set (en bas)

Figure 12: Exemple de l'application du filtre bilatéral sur une image d'entrée du train set (en haut) et du validation set (en bas)

de neurones, soient perdues.

Nous allons étudier d'autres **méthodes de filtrage d'amélioration de la netteté** pour améliorer la qualité de nos images tout en conservant la faible résolution de celles-ci. En effet, nos images sont à très basse résolution, ce qui permet d'accélérer nos entraînements mais cela empêche l'application de nombreux filtres.

4.1.2. Filtres d'amélioration de netteté

Nous avons exploré les filtres d'amélioration de la netteté, tels que les filtres Unsharp masking et Unsharp masking & Médian.

Filtre unsharp masking

Le filtre unsharp masking est une technique de filtrage, utilisé pour améliorer la netteté des images en renforcant les bords et les détails dans l'image tout en supprimant les bruits de faible amplitude (Figure 13).

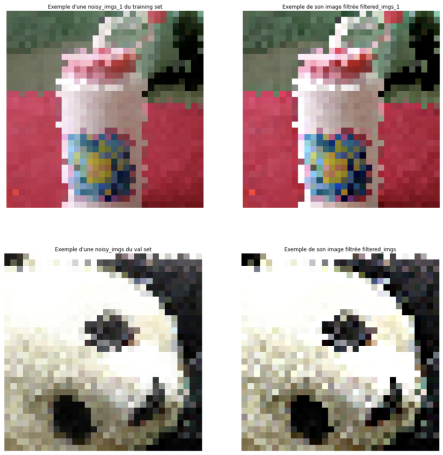

Figure 13: Exemple de l'application du filtre Unsharp masking sur une image d'entrée du train set (en haut) et du validation set (en bas)

Filtre unsharp masking & médian

Le filtre unsharp masking & médian combine le filtre unsharp masking et le filtre médian, vus auparavant. En combinant ces deux filtres, on peut obtenir une image qui est à la fois plus nette et moins bruyante, avec des bords et des contours mieux définis. Le filtre unsharp masking est appliqué en premier pour augmenter la netteté de l'image, puis le filtre médian est appliqué en second pour éliminer les impulsions de bruit.(Figure 14).

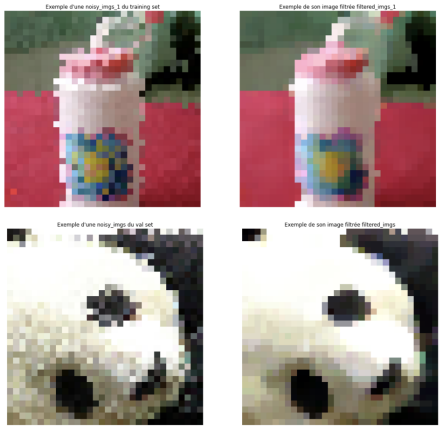

Figure 14: Exemple de l'application du filtre Unsharp masking & Médian sur une image d'entrée du train set (en haut) et du validation set (en bas)

Conclusion

Bien que ces filtres aient amélioré la qualité de l'image, nous avons constaté qu'ils n'amélioraient toujours pas notre PSNR moyen final car ils **perdaient également trop de détails**.

4.1.3. Filtre spécifique pour les images à faible résolution

Filtre de Nagao

Enfin, nous avons appliqué l'un des filtres spécifiques pour les images à faible résolution, le filtre de Nagao, qui est souvent utilisé pour améliorer la netteté des images à faible résolution telles que les images médicales, satellites et de microscope. Il s'agit d'une méthode de filtrage non linéaire, utilisé pour améliorer la qualité des images à faible résolution en réduisant le bruit. Il utilise des patchs de pixels pour calculer des coefficients de corrélation qui sont utilisés pour filtrer les pixels voisins. (Figure 15).

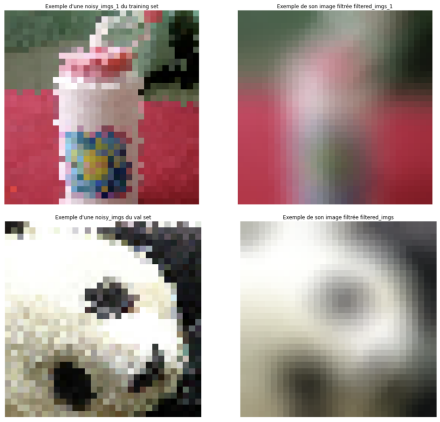

Figure 15: Exemple de l'application du filtre de Nagao sur une image d'entrée du train set (en haut) et du validation set (en bas)

Conclusion

Bien que ce filtre se rapproche plus de notre dataset, nous avons également constaté qu'il ne parvenait pas à améliorer notre PSNR moyen final car il **perdait également trop de détails**.

4.1.4. Résultats de l'application des filtres sur notre réseau

Voici les différents résultats du PSNR obtenus avec l'application des filtres :

Filtre appliqué	PSNR moyen (en dB)
Médian	24.08
Gaussien	25.24
Bilatéral	12.42
Unsharp masking	**25.43**
Unsharp masking & Médian	23.02
Nagao	23.04

Table 2: Résultats de l'application des filtres sur notre réseau U-Net

Nous remarquons que l'application des filtres sur nos données d'entraînement **n'améliorent pas notre PSNR moyen final**, le diminuent au contraire. En effet, nous obtenons initialement sans filtre un PSNR de 25.50 dB. Ceci peut s'expliquer par le fait que notre **réseau U-Net se comportant comme un filtre de débruitage**, l'application d'un filtre avant l'entraînement de notre réseau n'augmente en rien notre PSNR moyen final. Le filtre ayant déjà effectué un débruitage sur nos images, notre réseau considère qu'il n'a plus rien à traiter.

Le filtre qui diminue le moins notre PSNR moyen final est le **filtre unsharp masking**. En effet, il ne s'agit pas d'un filtre de débruitage et permet d'améliorer drastiquement la netteté de notre image (Figure 13).

En somme, notre étude sur ces trois parties de filtres nous a permis de mieux comprendre les limites et les avantages de chaque filtre en fonction de notre dataset et de notre objectif de débruitage d'images à basse résolution.

4.1.5. Benchmark

En vue des résultats non concluants de l'application des filtres sur notre réseau, il serait intéressant, en contrepartie, de **comparer l'efficacité de notre réseau U-Net à l'application des filtres sur nos images sans entraînement du modèle**, à travers la métrique du PSNR (Table 3).

Filtre appliqué sur le val set	PSNR moyen (en dB)
Médian	**22.74**
Gaussien	**24.30**
Bilatéral	**21.78**
Unsharp masking	17.87
Unsharp masking & Médian	**21.42**
Nagao	19.54

Table 3: Résultats du PSNR pour chacun des filtres sans entraînement du réseau

Nous remarquons que l'application des filtres sans entraînement de notre modèle ont des **PSNR moyens finaux moins élevés que celui après entraînement de notre U-Net**. En effet, nous obtenons après entraînement de notre réseau un PSNR de 25.50 dB.

Néanmoins, certains filtres permettent d'avoir des **résultats supérieurs au PSNR initial de 20.72 dB** entre l'image bruitée mise en entrée et notre clean image (sans application de filtre). Il s'agit des **filtres de débruitage** et non des filtres d'amélioration de la qualité d'image (*filtre_unsharp_maskin* et *filtre_nagao*). Ceci montre que l'application de ces filtres permettent de débruiter correctement notre image. Nous observons également que le **filtre Gaussien** est celui qui a le PSNR le plus élevé. Cela permet de soulever l'hypothèse qu'un bruit Gaussien a été distribué sur nos images d'entrées et cela expliquerait pourquoi le filtre de débruitage Gaussien répondrait au mieux au débruitage de nos images.

4.2. Deux modèles en série

La méthode effectuée consiste à **implémenter deux modèles U-Net en série à partir d'une première *predicted image* pour obtenir une deuxième *predicted image***.

L'objectif de cette méthode est d'**améliorer la qualité des prédictions du modèle** en utilisant ses propres prédictions précédentes comme données d'entraînement.

Cependant, cette méthode n'a pas conduit à une amélioration significative du PSNR moyen obtenu sur la deuxième *predicted image*, **24.97 dB contre 25.50 dB** sur la première *predicted image*.
En effet, cette méthode peut entraîner l'introduction de **bruit supplémentaire**, un **surapprentissage du modèle** aux données d'entraînement et une **détérioration de la qualité des prédictions** comme observé sur la Figure 16.

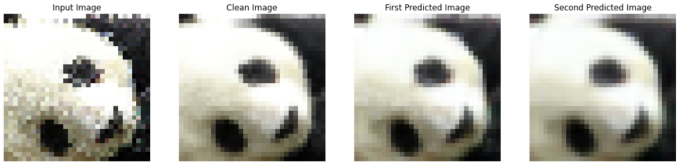

Figure 16: Comparaison de la première image prédite (c), la deuxième image prédite (d), avec l'image d'entrée (a) et l'image propre (b)

4.3. Générateur pour l'augmentation des données

Le but de la data augmentation est de **modifier les données sur des axes libres** pour augmenter les données **sans altérer l'essence du problème traité**.
Ici, on cherche à débruiter une image. Les questions que l'on doit se poser avant d'appliquer des transformations à l'image sont: quel **type de bruit** est utilisé ? le bruitage est t-il **dépendant de la position ou de ces voisins**? le bruitage **dépend t-il des couleurs**?
Nous avons supposé auparavant lors de l'application des filtres que le bruit est **gaussien** et qu'il ne dépend pas de la position et des couleurs. N'ayant pas de réponse précise sur le reste, nous avons appliqué des transformations assez général: flip horizontal, flip vertical et **voulons rajouter la permutation des couleurs** pour le projet de Computer Vision. La permutation des couleurs permet de rendre identique le traitement des trois couleurs, appliquant ainsi une contrainte par les données sur le modèle.

Néanmoins, ayant déjà appliqué pour le projet de deep learning les transformations flip horizontal et flip vertical au fichier initial *train_data.pkl*, le rajout sur le fichier *augmented_train_data.pkl* de la permutation des couleurs pose problème quant à **l'espace mémoire**.
Pour éviter la limitation de RAM, nous avons finalement choisi **d'augmenter les données dans le pipeline d'apprentissage** à partir du fichier initial *train_data.pkl*. Nous **modifions avec des transformations aléatoires**, comme des flips ou une permutation des couleurs, **les batchs dans la boucle de training**. Ainsi la fonction *generator_data*, dans *Preprocessing/Generator.py*, modifie de

la même manière le batch d'entrée et le batch de sortie. L'avantage est la réduction de la taille des données chargées mais la contre partie est un temps de training plus long.
On notera qu'avec cette méthode les instances par époque sont moins nombreuses, donc **pour traiter le même nombre d'instances** que précedemment il **faut augmenter le nombre d'époques** (précedemment nous traitions un training set déjà augmenté de 150000 instances, ayant à présent 50000 instances dans le dataset, il faut multiplier par 3 le nombre d'époques pour retrouver notre PSNR avec data augmentation de 25.50 dB). Cette méthode de data augmentation est beaucoup plus généralisable que nous pouvons ajouter autant de transformations que nous souhaitons.

Avec l'**ajout de la permutation des couleurs** comme méthode d'augmentation des données, nous voulons donc **augmenter notre dataset initial par 4** (flip horizontal, flip vertical et permutation des couleurs), soit 200000 instances. Pour pouvoir traiter le même nombre d'instances nous multiplions par 4 le nombre d'époques initial, soit 20×4 et obtenons un PSNR moyen final de 25.52 dB.

Une autre solution était d'enregistrer les données dans différents fichiers comme nous avions initialement procédé et de mettre en place une politique de loading mais dans ce cas-ci nous devions nous limiter en nombre de transformations ou ne pas tout explorer (et donc créer un potentiel biais).

4.4. Optimisation de l'entraînement

Pour **choisir correctement les hyperparamètres**, en particulier le nombre d'époques, nous avons **rajouté le calcul de la loss sur le set de validation**. Étant dans le cadre d'un dataset Kaggle, nous ne connaissons pas le dataset du test.

Grâce à la loss du validation set, nous pouvons **trouver le modèle optimal** ayant le meilleur **rapport performance/robustesse**.
Nous enregistrons le modèle avec la **loss du validation set la plus faible** (Figure 17).

Le graphique de la loss du train set sur la Figure 17 oscille, probablement dû au bruit sur la sortie. Nous observons sur le graphique de la loss du validation set que nous arrivons à un **plateau à partir de 40 époques** mais nous avons **continué à entraîner jusqu'à 120 époques** (la loss continuant à baisser).
La majorité de l'apprentissage est donc fait en 40 époques mais il n'y a pas d'overfitting apparant.

Nous obtenons ainsi un **PSNR moyen final** sur **120 époques de 25.55 dB**.

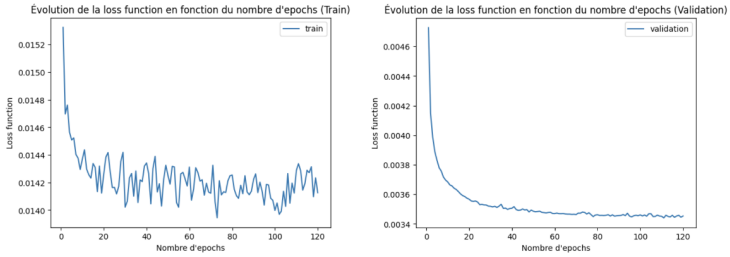

Figure 17: Évolution de la loss du training et validation sets en fonction du nombre d'epochs

5. Conclusion

Pour conclure, à partir du dataset Kaggle, deux architectures ont été implémentées et ont **prouvé leur efficacité dans la suppression de bruit dans les images**.

Le training avec Noise2Noise permet ainsi de pallier le problème du training discriminant qui nécessite l'image propre sous-jacente pour l'image bruitée donnée.

Dans un futur travail, il serait **intéressant de valider nos hypothèses de performance par rapport au modèle discriminant** et ainsi vérifier les résultats du papier original.

L'application de filtres sur le réseau U-Net n'a pas permis l'amélioration de notre PNSR moyen final. En vue des résultats de l'application des filtres sur notre réseau, il était intéressant de **comparer l'application des filtres sur les images par rapport à l'implémentation du réseau**. Le PSNR plus élevé de l'application du réseau témoigne que la mise en oeuvre du Deep Learning permet d'appliquer un débruitage plus performant qui ne serait pas possible avec des méthodes classiques de filtrage.
Néanmoins, un **filtre de super-résolution** pourrait répondre aux problèmes de basse résolution de nos images qui empêchent l'implémentation de nombreux filtres. Il serait pertinent d'observer les résultats obtenus, mais cela implique un temps de calcul plus long et une capacité de mémoire plus importante.

En mettant **deux modèles en série**, cela ne contribue pas à l'augmentation de notre PSNR, un seul modèle suffit. Il serait intéressant d'entraîner deux U-Net à la suite.
L'application d'un générateur pour **l'augmentation des données** est l'unique méthode qui a jusqu'alors permis d'accroître notre PSNR.
Enfin, **l'ajout du calcul de la loss sur le validation set** est indispensable pour trouver le modèle optimal ayant le meilleur rapport performance/robustesse.

Comme nous l'avons vu, l'apprentissage Noise2Noise nécessite néanmoins **deux paires d'images bruitées réalisées indépendamment pour une image propre**. Pour pal-

lier ce problème, la méthode **GAN2GAN (Generated-Artificial-Noise to Generated-Artificial-Noise)** propose un training génératif du bruit pour le débruitage d'images aveugles avec des images bruitées uniques.
D'après l'article original, il est montré que le débruiteur formé avec GAN2GAN, uniquement basé sur des images bruitées uniques atteint une bonne performance de débruitage, se rapprochant presque de la performance des modèles standards formés de manière discriminative ou Noise2Noise qui ont plus d'informations que le GAN2GAN. La méthode consiste à apprendre à générer des paires d'images bruitées synthétiques qui simulent des réalisations indépendantes du bruit dans les images données, puis à effectuer l'apprentissage Noise2Noise d'un débruiteur avec ces paires d'images bruitées générées synthétiquement.

References

[1] Céline Hudelot, *Cours de ComputerVision enseignés à CentraleSupélec.*

[2] Hervé Le Borgne, *Cours de DeepLearning enseignés à CentraleSupélec.*

[3] Ronneberger, Olaf, Fischer, Philipp, and Brox, Thomas. *U-net: Convolutional networks for biomedical image segmentation.* MICCAI, 9351:234–241, Mai 2015.

[4] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, Timo Aila, Octobre 2018, *Noise2Noise: Learning Image Restoration without Clean Data*

[5] Ibrahem Kandel, Mauro Castelli, Mai 2020, *The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset.*

[6] Sungmin Cha, Taeeon Park, Byeongjoon Kim, Jongduk Baek and Taesup Moon, Juillet 2021, *GAN2GAN: Generative noise learning for blind denoising with single noisy images.*

[7] Ph. Bolon, J-M. Chassery, J-P. Cocquerez, D. Demigny, C. Graffigne, A. Montanvert, S. Philipp, R. Zéboudj, J. Zerubia, 1995. *Analyse d'images : Filtrage et segmentation.*

[8] Bergounioux Maïtine, 2000, *Quelques méthodes de filtrage en traitement d'image.*

[9] B. Alsahwa, B. Solaiman, S. Almouahed, É. Bossé, Member, IEEE and D. Guériot, Juin 2016. *Iterative Refinement of Possibility Distributions by Learning for Pixel-Based Classification.*

[10] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, Timo Aila, Octobre 2018, *Noise2Noise: Learning Image Restoration without Clean Data - Official TensorFlow implementation of the ICML 2018 paper.* https://github.com/NVlabs/noise2noise.

[11] *Built-in Functions Python.* https://docs.python.org/2/library/functions.html#super.

[12] *PyTorch library.* https://pytorch.org/docs/stable/torch.html.

[13] *Télécom Paris - Filtrage des images.* https://perso.telecom-paristech.fr/bloch/ANIM/filtrage.pdf.

[14] *Geeks for Geeks.* https://www.geeksforgeeks.org/.

[15] *Unsharp masking with Python and OpenCV.* https://www.idtools.com.au/unsharp-masking-with-python-and-opencv/.

[16] *Stanford - Tutorial 1: Image Filtering.* https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html.