

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
UNIVERSITE MOULOUD MAMMERI DE TIZI-OUZOU



FACULTE DU GENIE ELECTRIQUE ET D'INFORMATIQUE
DEPARTEMENT D'ELECTRONIQUE

Mémoire de Fin d'Etudes De MASTER ACADEMIQUE

Domaine : Sciences et Technologies

Filière : Electronique

Spécialité : Electronique des Systèmes Embarqués

Présenté par

**Aumar Abderezak
Belhacel Chemseddine**

Thème

Suppression des artefacts JPEG par Deep Learning embarqué sur Raspberry Pi

Mémoire soutenu publiquement le 17 /09/2025 devant le jury composé de :

Mr Alouache Djamel
President, MCA, UMMTO

Mr Lahdir Mourad
Encadreur, Professeur, UMMTO

Mme Lahdir Leila
Examineur, MCB, UMMTO

Année universitaire 2024-2025

Remerciements

Avant tout, nous adressons nos louanges et notre profonde gratitude à Allah Tout-Puissant, qui nous a accordé la santé, la persévérance et la patience nécessaires pour mener à bien ce travail et franchir avec succès cette étape de notre parcours académique.

Nous tenons à exprimer nos sincères remerciements à notre encadreur, pour son suivi rigoureux, son accompagnement constant et la richesse de ses conseils, qui ont été d'un apport considérable dans l'élaboration de ce mémoire.

Nos remerciements s'adressent également aux membres du jury, pour l'honneur qu'ils nous font en acceptant d'évaluer notre travail, ainsi que pour l'intérêt, le temps et les remarques constructives qu'ils y consacreront.

Enfin, nous témoignons notre profonde reconnaissance à l'ensemble des enseignants du département d'électronique, dont le dévouement, la compétence et la transmission du savoir ont largement contribué à notre formation et à la réussite de ce modeste travail.

Dédicaces

Après de nombreuses années du travail et d'engagement, je dédie ce mémoire à :

Tout d'abord, Je rends grâce à Allah Tout-Puissant pour Son soutien, Sa guidance, ainsi que pour la force et la patience qu'Il m'a accordées tout au long de mon parcours.

À mon cher père,

Bien que tu ne sois plus présent à mes côtés, ton amour, tes conseils et tes sacrifices demeurent gravés à jamais dans mon cœur. Ta sagesse et tes valeurs continuent de me guider à chaque pas que j'entreprends. Ce travail est aussi le fruit de tes prières passées, de tes encouragements et de ton rêve de me voir réussir. Puisse ce modeste hommage être à la hauteur de tout ce que tu as représenté pour moi. Qu'Allah t'accorde Sa miséricorde et t'accueille en Son vaste paradis.

À ma chère mère,

Toi qui as toujours été mon pilier, mon refuge et ma source d'inspiration. Pour ton amour inconditionnel, tes innombrables sacrifices et tes prières sincères qui m'ont accompagné à chaque étape de ma vie. C'est grâce à ta patience, ton courage et ton soutien indéfectible que j'ai trouvé la force d'avancer et de persévérer malgré les difficultés. À toi, je dois ce que je suis aujourd'hui, et c'est avec une immense gratitude et un profond respect que je t'offre ce modeste travail.

À mon frère et ma sœur,

Pour leur affection, leur encouragement et leur présence réconfortante.

À mon binôme AUMAR Abderezak,

Pour l'entraide, la collaboration et la patience partagée tout au long de ce travail.

À mes enseignants,

Pour leurs efforts, leur savoir transmis et leurs précieux conseils.

À toute ma famille,

Pour leur confiance, leur soutien et leur amour constants

À Raiah Rabah,

Pour son soutien constant et ses encouragements précieux.

À mes amis AMIAR Mahmoud et LOUNNACI Yanis,

Pour leur sincère amitié, leur aide et leur présence à mes côtés.

BELHACEL CHEMSEDDINE

C'est avec une immense fierté que je dédie ce modeste travail :

*À Allah Tout-Puissant,
source de force, de sagesse et de miséricorde, en signe de gratitude pour toutes les
bénédictiones et l'aide qu'Il m'a accordées tout au long de mon parcours.*

*À ma chère mère,
lumière de ma vie, pour son amour inconditionnel, ses prières et ses sacrifices silencieux qui
m'ont toujours donné le courage d'avancer.*

*À mon père ,
pour son soutien constant, ses encouragements et ses précieux conseils qui ont forgé ma
persévérance et ma détermination.*

*À mon frère,
compagnon de route et de cœur, pour sa présence réconfortante et son appui indéfectible dans
les moments difficiles comme dans les réussites.*

*À ma tante maternelle,
pour son sacrifice et son accompagnement précieux dès les premiers pas de ma carrière
scolaire, je lui dédie ce travail avec une profonde reconnaissance.*

*À mes enseignants ,
pour la qualité de leur enseignement, leur dévouement et leur accompagnement constant, qui
ont nourri mon savoir et guidé mes pas vers la réussite.*

*À mes amis, Amiar Mahmoud et Lounnaci Yanis,
pour leur amitié sincère, leur soutien et les beaux moments partagés tout au long de notre
parcours universitaire.*

*À mon binôme BELHACEL chemseddine,
pour la collaboration, l'entraide et la confiance mutuelle qui ont rendu la réalisation de ce
mémoire possible et enrichissante.*

AUMAR ABDEREZAK

Résumé

Ce mémoire traite de la problématique des artefacts visuels générés par la compression JPEG et de leur impact sur la qualité des images numériques. Après avoir étudié les principes de la compression et ses limites, nous avons exploré l'apport des modèles de deep learning, notamment les réseaux convolutifs, pour la restauration d'images dégradées. L'ARCNN (Artifacts Reduction Convolutional Neural Network) a été sélectionné et entraîné sur une base de données représentative afin de réduire efficacement le flou, les effets de blocs et les distorsions de couleurs. Les performances ont été évaluées à l'aide des métriques PSNR et SSIM, confirmant une nette amélioration par rapport aux méthodes classiques. Enfin, une implémentation sur Raspberry Pi 4 a été réalisée, démontrant la faisabilité d'une solution embarquée à faible coût et ouvrant la voie à des applications réelles en vision par ordinateur, sécurité et télémédecine.

Abstract

This thesis addresses the issue of visual artifacts caused by JPEG compression and their impact on digital image quality. After analyzing the principles and limitations of image compression, we explored the contribution of deep learning models, particularly convolutional neural networks, to restore degraded images. The ARCNN (Artifacts Reduction Convolutional Neural Network) was selected and trained on a representative dataset to effectively reduce blurring, blocking effects, and color distortions. Performance was evaluated using PSNR and SSIM metrics, showing significant improvements compared to traditional methods. Finally, an implementation on Raspberry Pi 4 was carried out, demonstrating the feasibility of a low-cost embedded solution, paving the way for real-world applications in computer vision, security, and telemedicine.

Table des matières

Introduction :	1
-----------------------	----------

Chapitre 01 : Compression d'image

1. Préambule :	4
2. La compression d'images :	4
3. Schéma fonctionnel de la compression des images :	5
4. Les objectifs de la compression d'image :	6
5. Les types de la compression :	7
5.1 La compression d'images avec perte :	7
5.1.1 Les méthodes de la compression avec perte :	7
5.1.2 Les avantages et les Inconvénients :	7
5.1.3 Exemples de formats d'image utilisant la compression avec perte :	8
5.2 La compression d'image sans perte :	8
5.2.1 Les méthodes de la compression sans perte :	8
5.2.2 Les avantages et les Inconvénients :	8
5.2.3 Exemples de formats d'image utilisant la compression sans perte :	8
5.3 Choisir entre compression avec ou sans perte :	8
5.3.1 Facteurs à prendre en compte :	8
6. Le Format JPEG :	9
6.1 Définition :	9
6.2 La compression JPEG :	9
6.2.1 Transformation des couleurs :	9
6.2.2 Sous-échantillonnage de la chrominance :	10
6.2.3 Découpage en blocs :	11
6.2.4 Transformée DCT :	11
6.2.5 La quantification :	11
6.2.6 Codage, compression RLE et Huffman :	12
6.3 Étapes de la décompression JPEG	13
6.4 Les artefacts JPEG :	144
6.4.1 Définition :	144
6.4.2 Principaux points pour éliminer les artefacts JPEG :	155

6.4.3	Importance de l'artefact JPEG :	16
6.4.4	Exemples d'artefacts JPEG :	17
7.	Techniques classiques de post-traitement :	17
7.1	Filtrage :	18
7.1.1	Types de filtres :	18
7.2	Interpolation :	20
7.2.1	Types d'interpolation :	20
7.3	Application des techniques de post-traitement aux artefacts JPEG :	21
8.	Discussion :	21

Chapitre 02 : Modèles de Deep Learning pour la restauration d'images

1.	Préambule :	23
2.	Définition des modèles de deep learning :	233
3.	Structure générale d'un modèle de deep learning :	244
4.	Impact des modèles de deep learning sur la restauration d'images :	244
5.	Les différents algorithmes de Deep Learning :	255
5.1	Réseaux neuronaux convolutifs (CNN) :	255
5.2	Réseaux neuronaux récurrents (RNN) :	266
5.3	Réseaux de fonction de base radiale (RBFN) :	266
5.4	Réseaux de mémoire à long et court terme (LSTM) :	266
5.5	Réseaux adversariaux génératifs (GAN) :	266
5.6	Machines de Boltzmann restreintes (RBM) :	267
6.	Principaux modèles de deep learning pour la restauration d'images :	277
6.1	DnCNN : (Denoising Convolutional Neural Network)	277
6.1.1	Définition :	277
6.1.2	Objectif du DnCNN	27
6.1.3	Architecture du DnCNN :	278
6.1.4	Caractéristiques du modèle DnCNN :	30
6.2	ARCNN (Artifacts Reduction CNN) :	30
6.2.1	Définition :	30
6.2.2	Objectifs du ARCNN :	30
6.2.3	Architecture du ARCNN :	32
6.2.4	Caractéristiques du ARCNN :	33
6.3	FFDNet (Fast and Flexible Denoising Network) :	33
6.3.1	Définition :	33

6.3.2	Architecture du FFDNet :	35
6.3.3	Caractéristiques du FFDNet :	36
6.4	DRUNet (A Dilated-Residual U-Net) :	36
6.4.1	Définition :	37
6.4.2	Architecture du DRUNet :	38
6.4.3	Caractéristiques principales du DRUNet :	39
7.	Contraintes liées à l'implémentation de modèles de débruitage d'image (DnCNN, ARCNN, FFDNet, DRUNet) sur Raspberry Pi :	39
7.1	Limitations en termes de mémoire et de stockage :	39
7.2	Faible puissance de calcul :	39
7.3	Absence d'unité de traitement graphique (GPU) :	39
7.4	Complexité logicielle et compatibilité ARM :	40
7.5	Problèmes thermiques et énergétiques :	40
7.6	Dépendance à l'entraînement externe :	40
7.7	Résolution des images et contraintes d'E/S :	40
8.	Discussion :	40

Chapitre 03 : Application et Résultats

1.	Préambule :	42
2.	Sélection du modèle de Deep Learning :	422
2.1	Comparaison des architectures adaptées :	42
2.2	Choix du modèle : ARCNN	43
3.	Collecte et préparation des données :	43
3.1	MS COCO (Microsoft Common Objects in Context):.....	43
3.2	Prétraitement des images :	45
3.2.1	Redimensionnement :	45
3.2.2	Normalisation :	45
3.2.3	Génération des paires d'entraînement :	46
3.2.4	Répartition des jeux de données :	47
4.	Entraînement du modèle :	47
4.1	Environnement de développement (local) :	47
4.1.1	Langage de programmation (PYTHON) :	47
4.1.2	Bibliothèques :	48
4.1.3	Editeur de code :	50
4.2	Environnement de développement (en ligne) :	51

4.3	Stratégie d'entraînement.....	51
4.3.1	Télécharger le fichier arcnn.py	51
4.3.2	Fonctions de perte :	51
4.3.3	Optimisation :	52
4.3.4	Plan d'entraînement :	52
4.3.5	Sauvegarde du modèle :	53
4.4	Évaluation qualitative et quantitative (PSNR, SSIM) :	53
4.4.1	PSNR (Peak Signal-to-Noise Ratio):	56
4.4.2	SSIM (Structural Similarity Index Measure) :	56
5.	Implémentation sur Raspberry Pi :	58
5.1	Matériel :	61
5.1.1	Raspberry Pi 4 Model B :	61
5.1.2	Carte microSD :	61
5.1.3	Lecteur de carte SD :	63
5.2	Préparation et installation du système d'exploitation pour la Raspberry Pi 4 :	63
5.3	Installation des bibliothèques nécessaires :	67
6.	Tests et Résultats :	71
6.1	Résultats quantitatifs :	72
6.1.1	Évaluation avec PSNR	72
6.1.2	Évaluation avec SSIM.....	73
6.2	Résultats qualitatifs :	73
6.3	Interprétation des résultats :	77
7.	Fonctionnement des deux applications développées.....	77
7.1	Application temps différé :	77
7.2	Application temps réel :	80
8.	Discussion :	82
	Conclusion :	84
	Références bibliographiques	

Liste des figures

Chapitre 01 : Compression d'image

Figure 1: Compression d'image	4
Figure 2 : Schéma fonctionnel de la compression d'image avec perte	5
Figure 3 : Organigramme de compression pour JPEG.....	9
Figure 4: Sous-échantillonnage de la chrominance.....	10
Figure 5: Ordre de codage défini par la norme JPEG.	12
Figure 6: Schéma de codage simplifié.	13
Figure 7: Filtre Moyenneur	18
Figure 8: Filtre Median	19
Figure 9: Filtre Gaussien	19
Figure 10: Filtre bilateral.....	20

Chapitre 02 : Modèles de Deep Learning pour la restauration d'images

Figure 11: Structure générale d'un modèle de deep learning.....	24
Figure 12: Architecture de DnCNN	27
Figure 13: Objectifs de ARCNN	30
Figure 14: Architecture de ARCNN	31
Figure 15: Architecture de FFDNet.....	34
Figure 16: Architecture de DRUNet.....	37

Chapitre 03 : Application et Résultats

Figure 17: Exemples d'images de la base de données MS COCO (train2014)	44
Figure 18: Scripte de Redimensionnement	45
Figure 19: Scripte de la Normalisation	46
Figure 20: Organisation de l'ensemble de données dans le dossier ARCNN	47
Figure 21: Langage de programmation PYTHON	48
Figure 22: La bibliothèque PyTorch.....	48
Figure 23: La bibliothèque torchvision	49
Figure 24: La bibliothèque pillow	50
Figure 25: L'éditeur visual studio code	50
Figure 26: Le service google colab	51
Figure 27: Scripte de l'entraînement.....	55
Figure 28: Scripte pour calculer le PSNR	57
Figure 29: Scripte pour calculer le SSIM.....	59
Figure 30: Structure finale de projet développée	60

Figure 31: Raspberry Pi 4 Model B	63
Figure 32: Carte microSD	63
Figure 33: Lecture de la carte SD.....	63
Figure 34: Raspberry Pi Imager	64
Figure 35: Ouverture de Raspberry Pi Imager	64
Figure 36: Le choix de model de la carte Raspberry.....	65
Figure 37: Le choix de système.....	65
Figure 38: La sélection de la carte SD détectée	65
Figure 39: Cliquer “suivant ”	66
Figure 40: Cliquer “ Non ”	66
Figure 41: Cliquer “ oui ”	66
Figure 42: Installation de la carte microSD	67
Figure 43: Exécution de la commande “ sudo apt install python3 ”	67
Figure 44: Vérification de l’installation de PYTHON	68
Figure 45: PYTHON est installé	68
Figure 46: Exécutez de la commande “ sudo apt install python3-pip ”	68
Figure 47: Vérification de l’installation de Pip	68
Figure 48: Pip est installé	69
Figure 49: Exécutez de la commande “ python3 -m venv ~/nom_environnement ”	69
Figure 50: Dossier de l’environnement virtuel	69
Figure 51: Exécutez de la commande “ source ~/nom_environnement/bin/activate ”	70
Figure 52: Environnement virtuel est activé	70
Figure 53: Installation de la bibliothèque Pillow	70
Figure 54: Installation de la bibliothèque Pytorch	71
Figure 55: Installation de la bibliothèque Torchvision.....	71
Figure 56: Images du test.....	72
Figure 57: Image (A) compressée, restaurée par ARCNN.....	74
Figure 58: Image (B) compressée, restaurée par ARCNN	75
Figure 59: Image (C) compressée, restaurée par ARCNN.....	76
Figure 60 : Ouverture de l’application 1	77
Figure 61 : Cliquer sur “Choisi une image ”	78
Figure 62 : Sélectionner l’image.....	78
Figure 63 : Choix du Facteur de compression.....	79
Figure 64 : Compression et Restauration de l’image.....	79
Figure 65 : Affichage de résultats	80
Figure 66 : Ouverture de l’application 2	80
Figure 67 : Connexion au réseau local	81
Figure 68 : Lancement du serveur	81
Figure 69 : Acquisition et transmission	81
Figure 70 : Restauration et affichage du résultat	82

Liste des tableaux

Tableau 1 : Analyse comparative entre les modèles de Deep Learning	42
Tableau 2 : Résultats de PSNR en dB et leurs qualités	56
Tableau 3 : Résultats de SSIM et leurs qualités	58
Tableau 4 : Résultats de PSNR.....	72
Tableau 5 : Résultats de SSIM	73

Introduction

Au cours des dernières décennies, la révolution numérique a profondément transformé notre rapport à l'image. L'essor des smartphones, des appareils photo numériques, des caméras de surveillance et des dispositifs médicaux a conduit à une production massive de données visuelles. Ces images, souvent stockées, partagées ou transmises via des réseaux à bande passante limitée, nécessitent des techniques efficaces de compression afin de réduire leur taille tout en maintenant une qualité visuelle acceptable. Le format JPEG, standard internationalement adopté depuis les années 1990, constitue aujourd'hui l'une des méthodes de compression les plus répandues grâce à sa simplicité, son efficacité et sa compatibilité avec la majorité des systèmes [1]. Toutefois, cette technique repose sur une compression avec perte, ce qui entraîne l'apparition d'artefacts visuels indésirables tels que le flou, les blocs 8×8 ou les distorsions de couleurs. Ces dégradations peuvent limiter l'exploitation de l'image dans des domaines sensibles, notamment en télémédecine, en vision par ordinateur, en reconnaissance faciale ou en sécurité.

Face à cette problématique, la restauration d'images s'impose comme une étape essentielle visant à améliorer la qualité des images compressées et à réduire l'impact des artefacts. Historiquement, cette tâche reposait sur des approches classiques de traitement d'images, basées sur des filtres linéaires ou non linéaires (médian, gaussien, bilatéral, etc.) ou sur des modèles statistiques. Bien que ces méthodes aient permis d'obtenir des résultats intéressants, elles restent limitées en termes de robustesse et de capacité à généraliser sur différents types de dégradations.

C'est dans ce contexte que l'apprentissage profond (deep learning) a marqué un tournant majeur. Grâce aux réseaux de neurones convolutifs (CNN) et à des architectures avancées telles que DnCNN, ARCNN, FFDNet ou DRUNet, il est désormais possible d'atteindre des performances remarquables en restauration d'images [2]. Ces modèles sont capables d'apprendre directement, à partir de grandes bases de données, les relations complexes entre les images dégradées et leurs versions de référence. Ils surpassent ainsi largement les approches traditionnelles, aussi bien en termes de qualité perceptuelle qu'en termes de métriques objectives (PSNR, SSIM). De plus, l'intégration de ces modèles dans des systèmes embarqués, tels que le Raspberry Pi, ouvre de nouvelles perspectives pour des applications temps réel, portables et à faible coût, même si cela soulève d'importants défis liés aux ressources matérielles limitées.

Le présent mémoire s'inscrit dans cette dynamique et vise à étudier, implémenter et évaluer des modèles de deep learning pour la restauration d'images compressées, tout en explorant leur faisabilité sur une plateforme embarquée. Notre démarche consiste à analyser d'abord les fondements théoriques de la compression d'images et ses limitations, puis à présenter les modèles de deep learning les plus représentatifs dans le domaine de la restauration, avant de décrire la méthodologie expérimentale adoptée et de discuter les résultats obtenus.

À travers cette étude, notre objectif est double : d'une part, démontrer l'apport considérable des techniques de deep learning dans l'amélioration de la qualité des images compressées, et d'autre part, évaluer la possibilité d'intégrer ces modèles dans des environnements contraints comme le Raspberry Pi. Ce travail se veut donc une contribution à la fois scientifique et pratique, ouvrant des perspectives pour des applications réelles où l'équilibre entre qualité, performance et efficacité matérielle est essentiel.

Introduction

Ainsi, notre mémoire s'articule autour de quatre chapitres :

Le premier chapitre présente les principes de la compression, avec une attention particulière portée au format JPEG. Nous y détaillons ses étapes (transformation, quantification, encodage), ses avantages, ses limites et les artefacts qui en résultent.

Nous introduisons dans le deuxième chapitre les architectures les plus performantes dans la suppression de bruit et d'artefacts, en particulier DnCNN, ARCNN, FFDNet et DRUNet. Chaque modèle est étudié selon sa structure, ses mécanismes d'apprentissage et ses performances.

Le chapitre trois expose la démarche expérimentale adoptée ainsi que les résultats obtenus. Nous y détaillons la préparation des données (redimensionnement, compression, organisation du dataset), l'entraînement du modèle ARCNN, le choix des hyperparamètres et son déploiement sur Raspberry Pi 4. Les performances sont ensuite présentées et analysées à travers des comparaisons visuelles et quantitatives (PSNR, SSIM, rapidité d'exécution). Enfin, une discussion générale met en évidence les limites rencontrées et propose des perspectives d'amélioration, notamment dans le cadre des systèmes embarqués.

Nous terminons notre mémoire par une conclusion ainsi que par des perspectives pour améliorer notre travail.

Chapitre 01 : Compression d'image

1. Préambule :

Ce chapitre se concentre sur le format JPEG et la méthode de compression qu'il utilise. Nous examinerons les principes fondamentaux de la compression d'image JPEG, en détaillant ses étapes clés telles que la transformation en cosinus discrète (DCT), la quantification, et l'encodage. Nous étudierons également les compromis entre la taille du fichier et la qualité de l'image, ainsi que les paramètres influençant la compression, tels que le facteur de compression et le choix des paramètres de quantification.

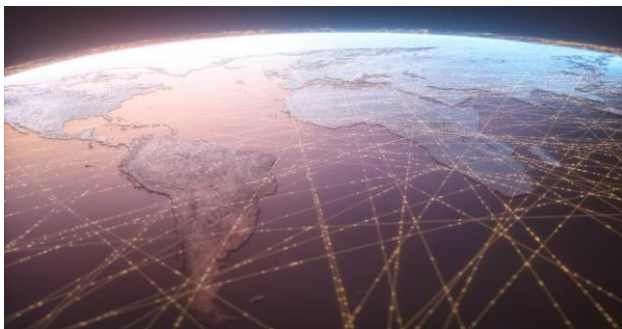
En plus de ses applications dans la photographie numérique, JPEG est également utilisé dans une large gamme de domaines, des sites web à la diffusion de médias en passant par les systèmes de sécurité. Ce chapitre nous permettra d'approfondir notre compréhension de cette norme largement utilisée et d'explorer ses avantages, ses limites et ses applications dans le monde numérique d'aujourd'hui.

2. La compression d'images :

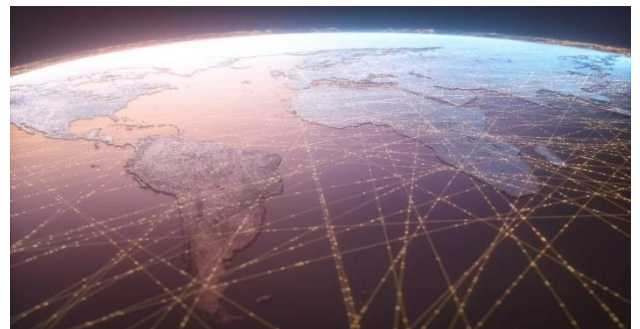
La compression d'images consiste à encoder les données d'une image en utilisant moins de bits que la représentation originale [3]. Cela se fait en éliminant les redondances (données inutiles ou répétitives) ou en exploitant les limites de la perception humaine pour réduire la quantité d'information sans altérer significativement la qualité perçue.



Une image de Taille 96 ko



La même image, compressée à 70 Ko



La même image, compressée à 35 Ko

Figure 1: Compression d'image

3. Schéma fonctionnel de la compression des images :

Le schéma fonctionnel de la compression est présenté dans la figure2 ci-dessous :

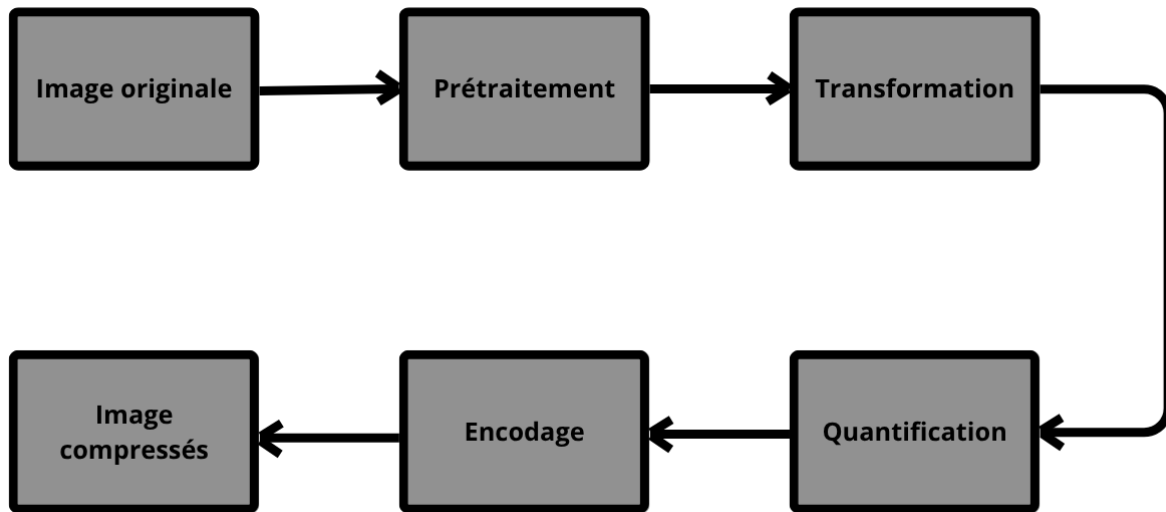


Figure 2: Schéma fonctionnel de la compression d'image avec perte

a. Prétraitement :

La première étape de la compression d'image consiste en un prétraitement, dont le but est de préparer l'image pour les étapes suivantes. Dans la plupart des cas, cela inclut la conversion de l'image de son espace colorimétrique d'origine, généralement RGB (red, green, blue), vers un espace plus adapté à la perception humaine, comme YCbCr. Dans ce nouvel espace, la composante de luminance (Y), qui correspond à la luminosité, est séparée des composantes de chrominance (Cb et Cr), représentant les couleurs. Cette séparation est utile car l'œil humain est plus sensible aux variations de luminosité qu'aux détails de couleur. De plus, l'image est souvent divisée en blocs de petite taille (souvent 8x8 pixels) pour permettre une analyse locale lors des transformations ultérieures.

b. Transformation :

Une fois l'image prétraitée, elle subit une transformation mathématique qui permet de passer du domaine spatial (où chaque pixel est défini par sa position et sa couleur) à un domaine fréquentiel. La transformation la plus courante est la DCT (Discrete Cosine Transform), utilisée dans le format JPEG. Cette transformation convertit chaque bloc de pixels en une série de coefficients représentant différentes fréquences de variation dans l'image. Les basses fréquences (variations lentes, comme des surfaces uniformes) sont généralement regroupées au début, tandis que les hautes fréquences (variations rapides, comme les détails fins) apparaissent ensuite. L'avantage de cette transformation est qu'elle permet de concentrer l'essentiel de l'information visuelle dans un petit nombre de coefficients, facilitant ainsi la compression.

c. Quantification :

L'étape de quantification suit immédiatement la transformation. Elle consiste à réduire la précision des coefficients obtenus en divisant chacun d'eux par une valeur prédéfinie (appelée matrice de quantification), puis en les arrondissant à l'entier le plus proche. Cette opération élimine une grande partie des hautes fréquences, c'est-à-dire les détails fins que l'œil humain perçoit difficilement. La quantification est l'étape où la perte d'information est introduite, ce qui en fait une compression avec perte. Plus la quantification est forte (valeurs plus élevées dans la matrice), plus la compression est efficace, mais cela se fait au détriment de la qualité visuelle de l'image.

d. Encodage :

La dernière étape du processus est l'encodage, qui consiste à représenter les coefficients quantifiés de manière plus efficace à l'aide de techniques de compression sans perte. L'une des méthodes les plus utilisées est le codage de Huffman, qui attribue des codes plus courts aux valeurs les plus fréquentes et des codes plus longs aux valeurs rares. Une autre technique est le codage RLE (Run-Length Encoding), qui compresse les séquences répétées (comme une longue série de zéros issue de la quantification). Ces méthodes permettent de réduire davantage la taille du fichier sans altérer l'information restante. Le résultat final est une version compressée de l'image, prête à être stockée ou transmise.

La différence essentielle entre le schéma de fonctionnement de la compression avec perte et celui de la compression sans perte réside dans l'étape de quantification. Dans la compression avec perte, après la transformation (comme la DCT ou la DWT), les coefficients sont quantifiés, ce qui consiste à réduire leur précision et à éliminer les détails jugés moins perceptibles par l'œil humain ; cette étape permet d'obtenir un taux de compression élevé mais entraîne une perte irréversible d'information. À l'inverse, dans la compression sans perte, il n'y a pas de quantification : l'image est simplement préparée (par filtrage ou prédiction), puis les données redondantes sont exploitées par des techniques de codage entropique (Huffman, LZW, DEFLATE, etc.), ce qui garantit que l'image reconstruite sera strictement identique à l'originale. Ainsi, la compression avec perte privilégie la réduction maximale de taille au prix de la qualité, tandis que la compression sans perte privilégie la fidélité parfaite des données au détriment du taux de compression.

4. Les objectifs de la compression d'image :

Les principaux objectifs de la compression d'images sont :

a. Réduction de la taille des fichiers :

- Diminuer l'espace de stockage nécessaire pour les images.
- Faciliter le partage et le transfert sur des réseaux à bande passante limitée.

b. Optimisation des performances :

- Accélérer le chargement des images sur les sites web ou les applications.

- Réduire la latence dans les systèmes de communication.
- c. Adaptation aux contraintes techniques :**
 - Adapter les images aux limites des dispositifs (écrans, appareils mobiles, etc.).
 - Répondre aux exigences des formats spécifiques (JPEG, PNG, WebP, etc.).
- d. Préservation de la qualité :**
 - Maintenir une qualité visuelle acceptable malgré la réduction de taille.
 - Minimiser les artefacts visuels (déformations, perte de détails, etc.).
- e. Économie de ressources :**
 - Réduire la consommation d'énergie liée au stockage et au transfert des données.
 - Optimiser les coûts associés à l'hébergement et à la bande passante.

5. Les types de la compression :

5.1 La compression d'images avec perte :

La compression avec perte est une méthode de réduction de la taille d'un fichier en éliminant une partie de ses données. Contrairement à la compression sans perte qui permet de restaurer un fichier à son état original sans perte de données, la compression avec perte supprime définitivement certaines informations du fichier.

La compression avec perte est un moyen efficace de réduire la taille des fichiers en sacrifiant une partie de leur qualité. Elle est particulièrement utile pour la gestion des archives, dans un contexte où l'espace de stockage est limité, celui de l'archivage par exemple.

5.1.1 Les méthodes de la compression avec perte :

Voici quelques méthodes de compression avec perte :

- Transformée en Cosinus Discrète (DCT - Discrete Cosine Transform).
- Transformée en Ondelettes (Wavelet Transform).
- Quantification.
- Compression Fractale.
- Compression par Modèles (Model-Based Compression).
- Compression par Prédiction (Predictive Coding).
- Compression par Masquage (Masking).

5.1.2 Les avantages et les Inconvénients :

- **Avantages :**
 - Compression plus efficace, entraînant des tailles de fichier nettement plus petites
 - Idéal pour le partage en ligne et les applications où la taille du fichier est une priorité
- **Inconvénients :**
 - Perte de qualité de l'image, pouvant entraîner une dégradation visible si la compression est trop importante
 - Impossibilité de récupérer les données perdues après décompression

5.1.3 Exemples de formats d'image utilisant la compression avec perte :

- JPEG (Joint Photographic Experts Group)
- WebP (développé par Google)

5.2 La compression d'image sans perte :

La compression d'image sans perte est une méthode de réduction de la taille des fichiers d'image sans sacrifier la qualité visuelle. Contrairement à la compression avec perte (comme JPEG), où certaines données sont irrémédiablement supprimées, la compression sans perte préserve toutes les informations originales de l'image. Cela la rend idéale pour des applications où la qualité et la fidélité des détails sont critiques, comme en photographie professionnelle, en imagerie médicale ou en archivage.

5.2.1 Les méthodes de la compression sans perte :

Voici quelques méthodes de compression avec perte :

- Codage de Huffman.
- Codage arithmétique.
- Algorithme de Lempel-Ziv (LZ77, LZ78, LZW).
- Run-Length Encoding (RLE).
- Burrows-Wheeler Transform (BWT).
- Prediction by Partial Matching (PPM).
- Entropy Coding.

5.2.2 Les avantages et les Inconvénients :

- **Avantages :**
 - Qualité d'image préservée, idéal pour les applications où la qualité est primordiale
 - Possibilité de récupérer l'image originale après décompression
- **Inconvénients :**
 - Généralement moins efficace pour réduire la taille du fichier par rapport à la compression avec perte
 - Peut entraîner des fichiers plus volumineux

5.2.3 Exemples de formats d'image utilisant la compression sans perte :

- PNG (Portable Network Graphics)
- TIFF (Tagged Image File Format)
- BMP (Bitmap)

5.3 Choisir entre compression avec ou sans perte :

5.3.1 Facteurs à prendre en compte :

- Qualité d'image souhaitée
- Taille du fichier et contraintes de stockage

- Objectif de l'utilisation de l'image (impression, partage en ligne, archivage, etc.)

6. Le Format JPEG :

6.1 Définition :

Le format JPEG, acronyme de « Joint Photographic Experts Group », est un type de fichier image qui utilise une technique de compression avec perte . Créé au début des années 1990, il a été conçu pour réduire la taille des fichiers tout en maintenant une qualité visuelle acceptable. Cette compression fonctionne en supprimant certaines données de l'image, ce qui permet de diminuer considérablement l'espace de stockage nécessaire. Si une partie de la qualité visuelle est sacrifiée, cela rend le JPEG particulièrement adapté pour des usages où la taille du fichier est cruciale, comme l'affichage sur le Web ou l'envoi d'images par e-mail. Le format JPEG est ainsi devenu une norme pour les photos numériques et les images à destination d'applications diverses.

6.2 La compression JPEG :

Le processus de compression et de décompression JPEG irréversibles comporte six étapes principales représentées ci-dessous :

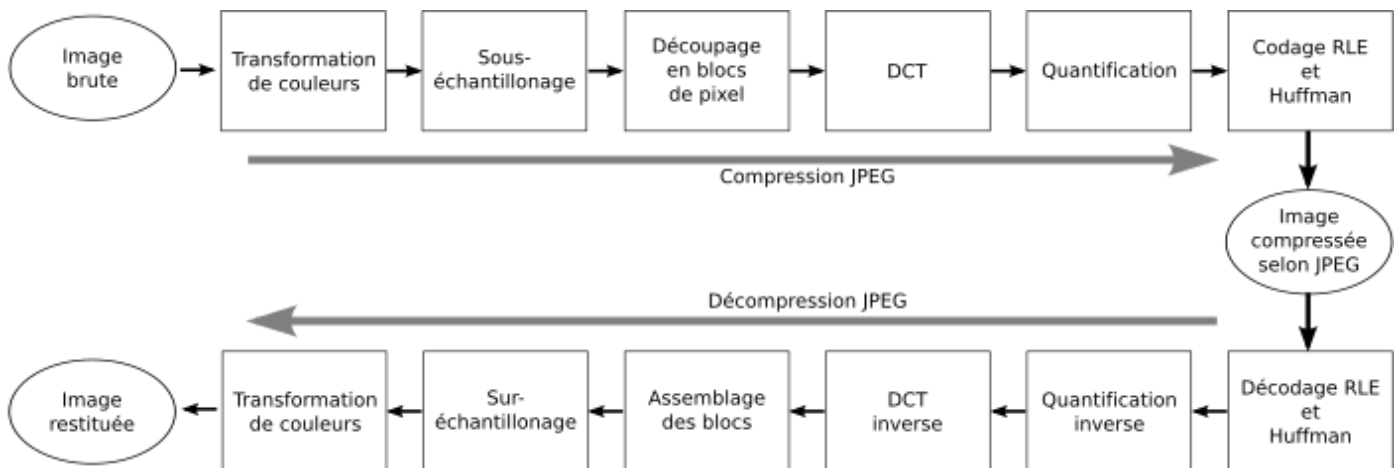


Figure 3: Organigramme de compression pour JPEG

6.2.1 Transformation des couleurs :

JPEG est capable de coder les couleurs sous n'importe quel format, toutefois les meilleurs taux de compression sont obtenus avec des codages de couleur de type luminance/chrominance car l'œil humain est assez sensible à la luminance (la luminosité) mais peu à la chrominance (la teinte) d'une image. Afin de pouvoir exploiter cette propriété, l'algorithme convertit l'image d'origine depuis son modèle colorimétrique initial (en général RVB) vers le modèle de type chrominance/luminance YCbCr. Dans ce modèle, Y est l'information de luminance, et Cb et Cr sont deux informations de chrominance, respectivement le bleu moins Y et le rouge moins Y .

6.2.2 Sous-échantillonnage de la chrominance :

Pour exploiter cette faible sensibilité de l'œil humain à la chrominance, on peut procéder à un sous-échantillonnage de ses signaux. Le principe de l'opération est de réduire de la taille de plusieurs blocs de chrominance en un seul bloc. Un traitement identique est appliqué aux blocs Cb et aux blocs Cr, tandis que les blocs de luminance (blocs Y) ne sont pas modifiés. Le sous-échantillonnage peut être réalisé selon plusieurs modes différents. Le type de mode utilisé dans une opération de compression est spécifié par la notation « J:a:b ». L'interprétation de la notation est la suivante :

L'interprétation de la notation est la suivante :

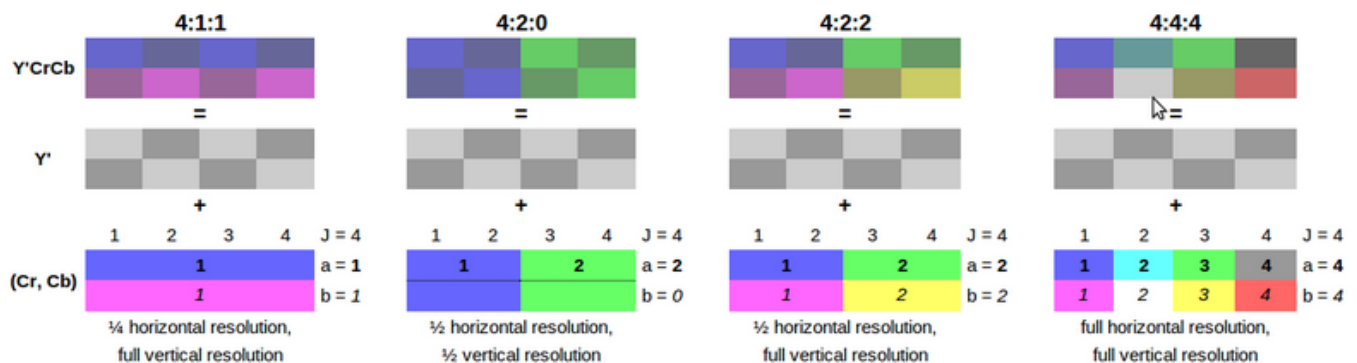


Figure 4: Sous-échantillonnage de la chrominance

J : représente la largeur de la plus petite matrice de pixels considérée (généralement 4)

a : le nombre de composantes de chrominance dans la première ligne

b : le nombre de composantes de chrominance supplémentaires dans la deuxième ligne

Cas 4:4:4 :

Ce cas correspond à l'absence d'application du sous-échantillonnage.

Cas 4:2:2 :

Ce cas correspond à un sous-échantillonnage qui conduit à une réduction d'un facteur 1/2 de la taille des blocs Cb/Cr. Pour cela, on calcule la valeur moyenne de chrominance de deux pixels voisins horizontalement.

Cas 4:2:0 :

La moyenne de la chrominance de quatre pixels voisins est calculée et stockée dans le bloc produit par le sous-échantillonnage. Les pixels voisins forment des blocs carrés de dimension 2×2 pixels. Le sous-échantillonnage 4:2:0 ajoute donc une réduction verticale à la réduction horizontale du mode 4:2:2. Il conduit à une division par 4 de la taille des blocs Cb/Cr.

Cas 4:1:1 :

Ce dernier cas est rare en compression JPEG, et est plutôt appliqué dans des applications vidéo tel que les formats NTSC DV, DVCAM, DVCPRO.

Ce type de sous-échantillonnage permet de réduire de manière importante la taille des images. Cependant, il conduit à une perte d'information qui ne peut être récupérée dans la phase de décodage de l'image ; il s'agit donc d'une opération de compression irréversible. À l'issue de cette phase, les données se présentent sous la forme de blocs de coefficients Y, Cb et Cr, le nombre de blocs Cb et Cr étant égal au quart, à la moitié ou à 100 % du nombre de blocs Y selon le mode sous-échantillonnage appliqué.

6.2.3 Découpage en blocs :

L'image est découpée en blocs de 64 (8×8) sous-pixels. Les informations de luminance n'étant pas sous-échantillonnées, chaque bloc correspond à 8×8 pixels dans l'image d'origine. Les informations de chrominance, par contre, correspondent en fonction du sous-échantillonnage à 8×8 , 16×8 ou 16×16 pixels dans l'image d'origine.

6.2.4 Transformée DCT :

La transformée DCT (*Discrete Cosine Transform*, en français transformée en cosinus discrète), est une transformation numérique qui est appliquée à chaque bloc. Cette transformation est une variante de la transformée de Fourier. Elle décompose un bloc, considéré comme une fonction numérique à deux variables, en une somme de fonctions cosinus oscillant à des fréquences différentes. Chaque bloc est ainsi décrit en une carte de fréquences et en amplitudes plutôt qu'en pixels et coefficients de couleur. La valeur d'une fréquence reflète l'importance et la rapidité d'un changement, tandis que la valeur d'une amplitude correspond à l'écart associé à chaque changement de couleur.

À chaque bloc de $N \times N$ pixels sont ainsi associées $N \times N$ fréquences

La transformée DCT s'exprime mathématiquement par :

$$\text{DCT}(i, j) = \frac{2}{N} C(i) C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \text{pixel}(x, y) \cos \left[\frac{(2x+1)i\pi}{2N} \right] \cos \left[\frac{(2y+1)j\pi}{2N} \right] \quad \dots\dots (1)$$

La constante C vaut :

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}} & \text{pour } x = 0 \\ 1 & \text{pour } x > 0 \end{cases} \quad \dots\dots\dots (2)$$

6.2.5 La quantification :

La quantification est l'étape de l'algorithme de compression JPEG au cours de laquelle se produit la majeure partie de la perte d'information (et donc de la qualité visuelle), mais c'est aussi celle qui permet de gagner le plus de place (contrairement à la DCT, qui ne compresse pas).

La DCT a retourné, pour chaque bloc, une matrice de 8×8 nombres (dans l'hypothèse que les blocs de l'image font 8×8 pixels). La quantification consiste à diviser cette matrice par une autre, appelée matrice de quantification, et qui contient 8×8 coefficients spécifiquement choisis par le codeur [4].

Chapitre 01 : Compression d'image

Le but est ici d'atténuer les hautes fréquences, c'est-à-dire celles auxquelles l'œil humain est très peu sensible. Ces fréquences ont des amplitudes faibles, et elles sont encore plus atténuées par la quantification ; certains coefficients sont même souvent ramenés à 0.

Le calcul permettant la quantification est le suivant :

$$F^*(u, v) = \left\lfloor \frac{F(u, v) + \left\lfloor \frac{Q(u, v)}{2} \right\rfloor}{Q(u, v)} \right\rfloor \cong \text{entier le plus proche} \left(\frac{F(u, v)}{Q(u, v)} \right) \quad \dots\dots\dots (3)$$

Avec : $\lfloor x \rfloor$ entier directement inférieur à x

Avec :

$F(u, v)$ → Coefficient de la transformée (DCT).

$Q(u, v)$ → Élément de la matrice de quantification.

$F^*(u, v)$ → Coefficient quantifié.

6.2.6 Codage, compression RLE et Huffman :

Le codage s'effectue en zigzag comme le montre la figure suivante et se termine par un caractère de fin :

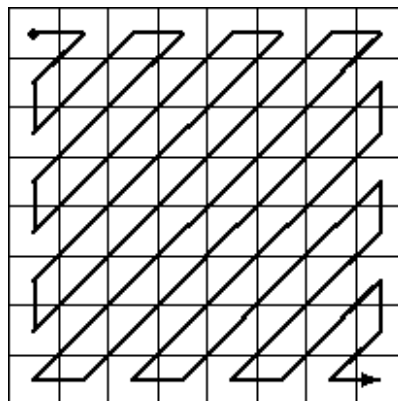


Figure 5: Ordre de codage défini par la norme JPEG

Ce résultat est ensuite compressé selon un algorithme RLE basé sur la valeur 0 (le codage RLE intervient uniquement sur cette dernière), puis un codage entropique de type Huffman ou arithmétique.

Voici le schéma de codage très simplifié suivant :

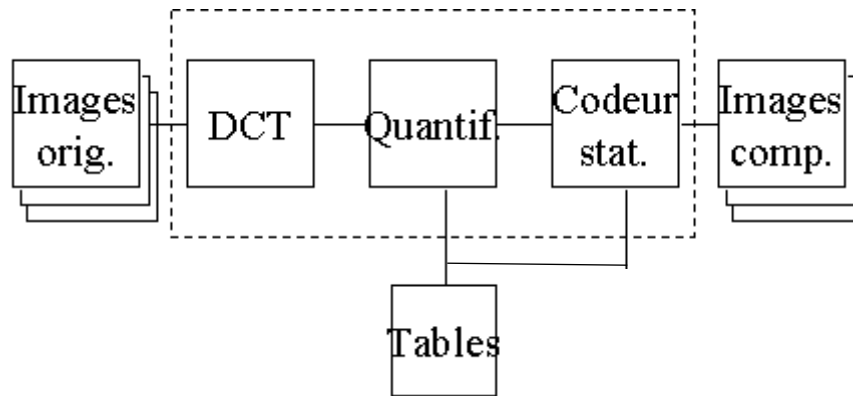


Figure 6: Schéma de codage simplifié

6.3 Étapes de la décompression JPEG :

- **Décodage RLE et Huffman :**

Le fichier JPEG est stocké sous forme d'un flux binaire compressé. La première étape consiste à décoder ce flux en utilisant les tables de Huffman (ou arithmétiques) et le codage RLE (Run Length Encoding). Cela permet de retrouver les coefficients quantifiés de chaque bloc.

- **Déquantification (Quantification inverse) :**

Les coefficients obtenus sont encore réduits à cause de la quantification effectuée lors de la compression. On les multiplie donc par la matrice de quantification utilisée à l'origine, afin d'obtenir une approximation des coefficients DCT initiaux.

- **DCT inverse (IDCT) :**

On applique la transformée en cosinus discrète inverse sur chaque bloc (généralement 8×8). Cette opération ramène les coefficients fréquentiels dans le domaine spatial, permettant de reconstruire les valeurs de pixels.

- **Assemblage des blocs :**

Une fois chaque bloc reconstruit, il faut les remettre dans leur position d'origine pour reconstituer l'image complète. À cette étape, l'image peut présenter de légers artefacts de blocs visibles.

- **Sur-échantillonnage :**

Si un sous-échantillonnage chromatique (comme 4:2:0) a été utilisé à la compression, il faut maintenant restaurer la résolution des canaux de couleur (Cb et Cr). Cela se fait par interpolation afin de les ramener à la même taille que le canal de luminance (Y).

- **Transformation de couleur :**

Enfin, on convertit l'image de l'espace YCbCr (luminance + chrominance) vers l'espace RGB (Rouge, Vert, Bleu), qui est celui utilisé par la majorité des écrans. L'image obtenue est alors lisible et visualisable par l'utilisateur.

6.4 Les artefacts JPEG :

6.4.1 Définition :

L'artefact JPEG fait référence à une distorsion notable ou à des imperfections visibles dans une image causée par la compression avec perte utilisée dans le format de fichier JPEG [5]. Lors de la compression, les données de l'image sont simplifiées pour réduire la taille du fichier, ce qui entraîne souvent une perte de certains détails et de la qualité. Ces artefacts apparaissent généralement sous la forme de flous, de zones en blocs ou de bruit, en particulier dans les zones présentant des bords nets, un contraste élevé ou des couleurs multiples.

6.4.2 Les différents artefacts JPEG :

- **Effet de blocs (Blocking) :**

L'effet de blocs apparaît lorsque l'image compressée présente une division visible en petits carrés de 8×8 pixels. Cela est dû au fait que le JPEG applique la DCT bloc par bloc. Quand le taux de compression est trop élevé, les transitions entre ces blocs deviennent perceptibles, donnant une impression de mosaïque qui dégrade fortement la qualité visuelle.

- **Flou (Blurring) :**

Le flou est un autre artefact fréquent qui se manifeste par une perte de netteté dans l'image. Lors de la quantification, les hautes fréquences responsables des détails fins et des contours précis sont supprimées ou réduites. En conséquence, l'image paraît adoucie, et les bords deviennent moins marqués qu'ils ne l'étaient dans l'original.

- **Effet d'anneaux (Ringing) :**

L'effet d'anneaux se traduit par de petites ondulations ou halos qui apparaissent autour des contours nets. Ce phénomène provient de la suppression des coefficients fréquentiels élevés lors de la compression. Au lieu de conserver des bords nets, l'image présente des oscillations parasites qui attirent l'œil et perturbent la lisibilité des détails.

- **Effet de bande (Banding) :**

L'effet de bande survient principalement dans les zones uniformes contenant des dégradés de couleurs ou de luminosité, comme un ciel bleu ou une surface éclairée. Après compression, ces transitions ne sont plus continues mais apparaissent par paliers. Le dégradé devient donc segmenté, et on distingue clairement les limites entre les différentes bandes de couleur.

- **Perte de texture :**

La perte de texture touche les régions de l'image riches en petits détails, comme l'herbe, les cheveux ou les tissus. Ces informations fines sont considérées comme secondaires par l'algorithme de compression et sont fortement simplifiées. Le résultat est une surface lissée, dépourvue des irrégularités naturelles qui donnaient du réalisme à l'image.

- **Bruit de compression :**

Enfin, le bruit de compression se manifeste par l'apparition de points parasites ou d'un grain artificiel, particulièrement visible dans les zones sombres ou très compressées. Contrairement au bruit naturel présent dans une photo, ce bruit est généré par la compression elle-même et réduit encore davantage la qualité perçue.

6.4.3 Principaux points pour éliminer les artefacts JPEG :

Pour éliminer ou réduire les artefacts JPEG, il existe plusieurs techniques et bonnes pratiques, que ce soit lors de la capture, de l'enregistrement ou du post-traitement de l'image. Voici les principaux points à considérer :

- **Utiliser une qualité JPEG élevée lors de l'enregistrement :**

Réduire le taux de compression : Lorsque vous enregistrez une image au format JPEG, choisissez une qualité élevée (par exemple, 90% ou plus). Cela minimise la perte de données et réduit les artefacts.

Éviter les réenregistrements multiples : Chaque fois que vous enregistrez une image JPEG, de nouvelles pertes de données se produisent. Utilisez des formats sans perte (comme PNG ou TIFF) pour l'édition intermédiaire.

- **Utiliser des formats sans perte pour l'édition :**

Formats recommandés : Utilisez des formats sans perte comme PNG, TIFF ou WebP (sans perte) pour enregistrer vos images pendant l'édition. Cela évite la dégradation cumulative due à la compression JPEG.

Convertir en JPEG uniquement pour la diffusion : Enregistrez l'image finale en JPEG uniquement lorsque vous êtes prêt à la partager ou à la publier.

- **Post-traitement pour réduire les artefacts :**

Plusieurs logiciels de retouche photo proposent des outils pour atténuer les artefacts JPEG. Voici quelques techniques courantes :

- a. Réduction du bruit et lissage :**

Outils de réduction du bruit : Des logiciels comme Adobe Photoshop, GIMP ou Lightroom proposent des filtres pour réduire le bruit et lisser les zones affectées par les artefacts.

Flou gaussien : Appliquez un léger flou gaussien sur les zones présentant des artefacts de blocs ou de "ringing".

b. Correction des artefacts de blocs :

Filtres de lissage : Utilisez des filtres spécifiques pour lisser les bordures entre les blocs de 8x8 pixels.

Outils de netteté : Appliquez un filtre de netteté de manière sélective pour améliorer les détails sans accentuer les artefacts.

c. Correction des artefacts de couleur :

Ajustement des couleurs : Utilisez des outils de correction des couleurs pour atténuer les distorsions de chrominance (comme les bandes de couleur ou le "color bleeding").

Dé-saturation sélective : Réduisez la saturation dans les zones où les artefacts de couleur sont visibles.

d. Réduction des artefacts de "ringing" :

Masquage flou : Utilisez des techniques de masquage flou pour atténuer les halos autour des bords nets.

Correction manuelle : Utilisez des outils comme le tampon de clonage ou la correction par patch pour retoucher manuellement les zones affectées.

- **Utiliser des algorithmes de reconstruction d'image :**

IA et apprentissage profond : Des outils modernes utilisent l'intelligence artificielle pour reconstruire les détails perdus et réduire les artefacts JPEG. Par exemple :

Topaz Labs : Des logiciels comme Topaz Denoise AI ou Topaz Sharpen AI peuvent aider à réduire les artefacts tout en préservant les détails.

Adobe Photoshop (Neural Filters) : Les filtres neuronaux de Photoshop peuvent améliorer les images compressées en réduisant les artefacts.

- **Éviter les artefacts dès la capture :**

Capter en RAW : Si vous utilisez un appareil photo, photographiez en format RAW plutôt qu'en JPEG. Le format RAW conserve toutes les données du capteur, ce qui permet une édition sans perte.

Éviter les hauts ISO : Un ISO élevé peut amplifier le bruit, ce qui peut aggraver les artefacts JPEG lors de la compression.

6.4.4 Importance de l'artefact JPEG :

L'artefact JPEG est un terme important dans le marketing numérique car il fait référence à la distorsion ou aux écarts visuels qui peuvent survenir lorsqu'une image JPEG est compressée.

La qualité étant cruciale dans les visuels marketing, la compréhension des artefacts JPEG est essentielle pour les spécialistes du marketing numérique qui s'efforcent de produire des images et du contenu visuel de haute qualité.

Lorsqu'un fichier JPEG est fortement compressé, ces artefacts deviennent plus visibles, ce qui donne souvent lieu à des images pixélisées ou floues, ce qui peut nuire à l'attrait et au professionnalisme d'une marque.

En étant conscients des artefacts JPEG, les spécialistes du marketing numérique peuvent prendre des décisions éclairées pour optimiser les images, en équilibrant la qualité visuelle avec la taille du fichier, afin de garantir que leur contenu conserve un niveau élevé de fidélité visuelle tout en se chargeant rapidement sur divers appareils et plates-formes.

6.4.5 Exemples d'artefacts JPEG :

Le terme « artefact JPEG » désigne les imperfections ou distorsions visuelles qui se produisent dans une image lorsqu'elle est enregistrée au format JPEG en raison de l'algorithme de compression utilisé. On l'observe principalement dans les images comportant un niveau de détail élevé, des contrastes prononcés ou du texte. Voici trois exemples concrets :

- **Photos de profil sur les réseaux sociaux :** lorsque les utilisateurs téléchargent leurs photos de profil sur une plateforme de réseaux sociaux, les images sont souvent compressées pour minimiser le stockage et optimiser la vitesse de chargement. Cette compression peut entraîner l'apparition d'artefacts, tels que la pixellisation ou la distorsion des couleurs, qui n'étaient pas présents dans la version originale de haute qualité de l'image.
- **Bannières publicitaires :** les spécialistes du marketing rencontrent souvent des artefacts JPEG lors de la création de bannières publicitaires numériques. Ces artefacts peuvent devenir visibles lorsque le concepteur compresse une image avec des couleurs vives et des lignes nettes pour réduire la taille du fichier. Cette opération est effectuée pour répondre aux exigences de taille des publicités d'une plateforme ou d'un site Web. Lorsque les internautes voient la bannière publicitaire, ils peuvent remarquer une pixellisation ou des zones « en blocs » dans l'image, ce qui peut affecter l'attrait visuel global de la publicité.
- **Images d'actualités en ligne ou de blogs :** les sites Web cherchant à optimiser le temps de chargement de leurs pages ont tendance à compresser les images qui accompagnent les articles d'actualité ou les billets de blog. Cela peut entraîner l'apparition d'artefacts JPEG, en particulier lorsque l'image contient du texte, des couleurs vives ou des zones à fort contraste. Dans ce cas, les lecteurs peuvent trouver la qualité de l'image inférieure, ce qui peut avoir un impact sur leur expérience globale du contenu du site Web.

7. Techniques classiques de post-traitement :

Les techniques de post-traitement sont utilisées pour améliorer la qualité des images après leur acquisition ou leur compression. Elles sont particulièrement utiles pour corriger des défauts comme les artefacts JPEG, le bruit, le flou ou les distorsions. Voici une explication des

techniques classiques de post-traitement, notamment le filtrage et l'interpolation, ainsi que leur application pour atténuer les artefacts JPEG [6].

7.1 Filtrage :

Le filtrage est une technique qui consiste à appliquer un filtre à une image pour modifier ses pixels en fonction des pixels voisins. Il est souvent utilisé pour atténuer le bruit, lisser les textures ou réduire les artefacts.

7.1.1 Types de filtres :

a. Filtre moyennneur (Mean Filter) :

Le filtre moyennneur est une opération de traitement d'images utilisée pour réduire le bruit dans une image et/ou flouter une image. Par exemple, l'application d'un filtre moyennneur sur l'image de gauche donne l'image de droite :

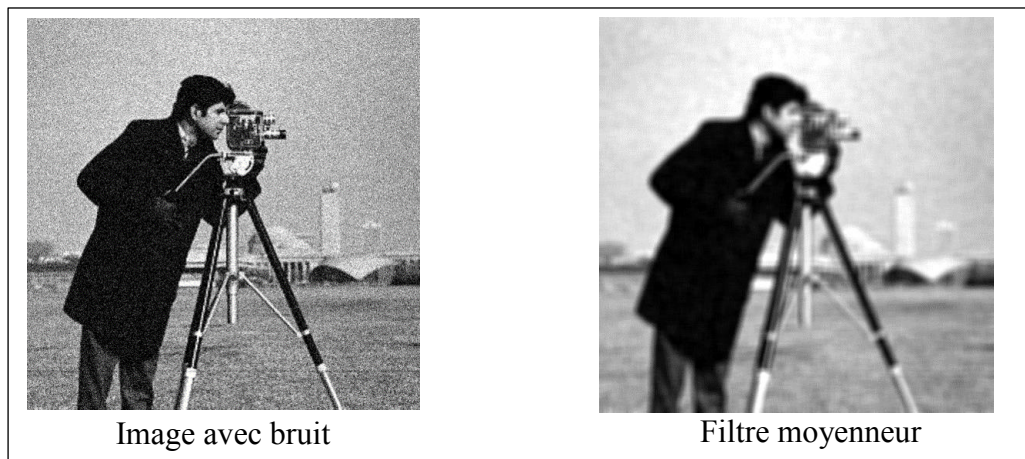


Figure 7: Filtre Moyennneur

b. Filtre médian (Median Filter) :

Le filtre médian est un filtre numérique non linéaire, souvent utilisé pour la réduction de bruit. La réduction de bruit est une étape de pré-traitement classique visant à améliorer les résultats de traitements futurs (détection débords par exemple). La technique de filtre médian est largement utilisée en traitement d'images numériques car il permet sous certaines conditions de réduire le bruit tout en conservant les contours de l'image.

Efficace pour supprimer le bruit impulsif (comme le "salt and pepper") tout en préservant les bords.

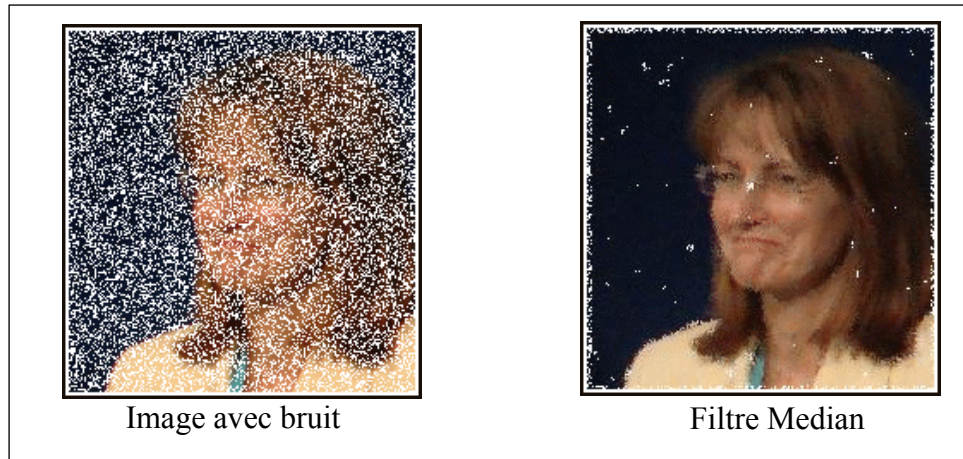


Figure 8: Filtre Median

c. Filtre gaussien (Gaussian Filter) :

Un filtre gaussien est une technique de traitement d'image largement utilisée qui applique une fonction gaussienne pour lisser ou flouter les images. Ce filtre est particulièrement efficace pour réduire le bruit et les détails d'une image. Produit un effet de flou doux, souvent utilisé pour atténuer le bruit ou les artefacts. Réduire les artefacts de "ringing" et de "mosquito noise" dans les images JPEG.

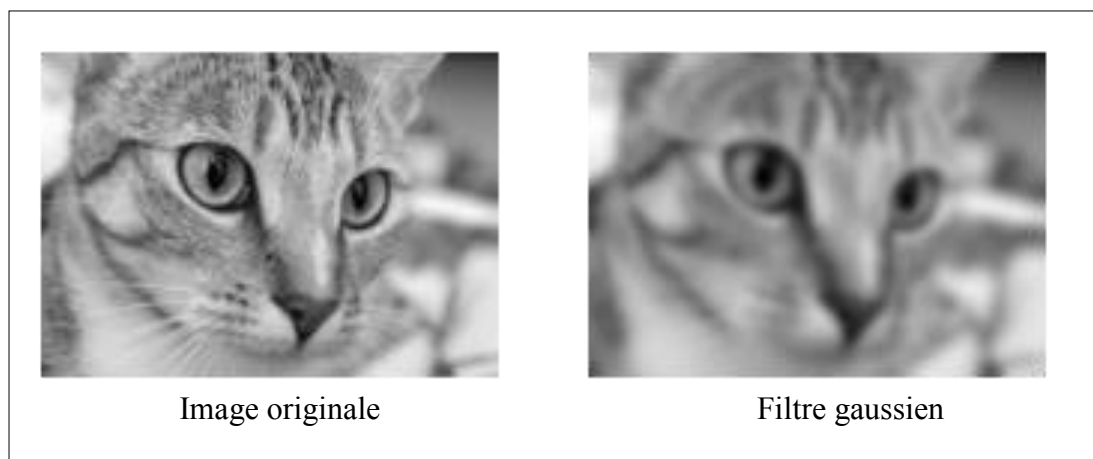


Figure 9: Filtre Gaussien

d. Filtre bilatéral (Bilateral Filter) :

Le filtre bilatéral est une technique de lissage avancée utilisée en traitement d'image pour réduire le bruit tout en conservant les contours. Contrairement aux filtres classiques (comme le flou gaussien), qui lissent uniformément l'image. Préserve les bords tout en laissant les zones uniformes.

Application : Idéal pour réduire les artefacts JPEG tout en conservant les contours nets.

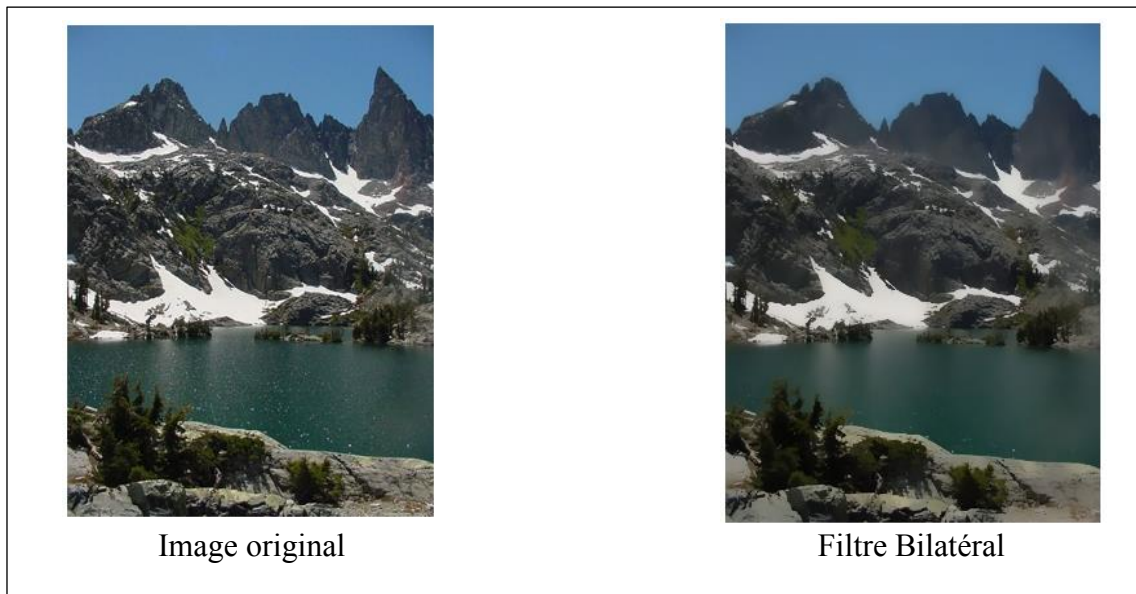


Figure 10: Filtre Bilatéral

7.2 Interpolation :

L'interpolation est une technique utilisée pour estimer les valeurs de pixels manquants ou pour redimensionner une image tout en préservant sa qualité. Elle est souvent utilisée pour corriger les distorsions ou améliorer la résolution.

7.2.1 Types d'interpolation :

a. Interpolation linéaire :

- Estime la valeur d'un pixel en fonction d'une moyenne pondérée des pixels voisins.
- Simple et rapide, mais peut produire des artefacts de flou.
- **Application** : Utilisée pour redimensionner des images ou corriger des distorsions mineures.

b. Interpolation bicubique :

- Utilise un polynôme cubique pour estimer les valeurs de pixels, ce qui donne des résultats plus précis que l'interpolation linéaire.
- Produit des transitions plus douces et préserve mieux les détails.
- **Application** : Idéale pour le redimensionnement d'images ou la correction d'artefacts de compression.

c. Interpolation par Lanczos :

- Utilise une fonction de fenêtrage pour améliorer la qualité de l'interpolation.
- Produit des résultats plus nets que l'interpolation bicubique, mais peut introduire des artefacts de "ringing".
- **Application** : Utilisée pour le redimensionnement haute qualité ou la reconstruction d'images.

d. Interpolation par apprentissage profond (Deep Learning-based Interpolation) :

- Utilise des réseaux de neurones pour prédire les valeurs de pixels manquants ou améliorer la résolution.
- Capable de produire des résultats très précis, même sur des images fortement compressées.
- **Application :** Reconstruction d'images compressées ou amélioration de la résolution (super-résolution).

7.3 Application des techniques de post-traitement aux artefacts JPEG :

Voici comment ces techniques peuvent être utilisées pour atténuer les artefacts JPEG :

a. Réduction des artefacts de blocs :

- Appliquer un filtre moyenneur ou filtre bilatéral pour lisser les transitions entre les blocs.
- Utiliser un filtre de débruitage pour réduire les distorsions tout en préservant les détails.

b. Réduction du "ringing" :

- Appliquer un filtre gaussien pour atténuer les halos autour des bords nets.
- Utiliser une interpolation bicubique pour reconstruire les zones affectées.

c. Réduction du "mosquito noise" :

- Appliquer un filtre médian ou un filtre de débruitage pour supprimer les distorsions ressemblant à du bruit.

d. Amélioration des gradients et des couleurs :

- Utiliser une interpolation bicubique ou Lanczos pour lisser les bandes de couleur et les dégradés.

e. Reconstruction des détails perdus :

- Utiliser des techniques de super-résolution basées sur l'apprentissage profond pour reconstruire les détails fins perdus lors de la compression.

8. Discussion :

Dans ce chapitre, nous avons exploré les principes fondamentaux de la compression d'image JPEG, l'un des formats les plus largement utilisés pour la compression d'images numériques. Ce format a été conçu pour offrir une réduction substantielle de la taille des fichiers d'image tout en maintenant une qualité acceptable, ce qui le rend particulièrement adapté à des applications variées comme la photographie numérique, la gestion d'images sur le web et la diffusion multimédia.

Chapitre 2 : Modèles de Deep Learning pour la restauration d'images

1. Préambule :

L'avènement du deep learning a profondément transformé le domaine de la restauration d'images, en particulier pour les tâches de débruitage, de suppression d'artefacts et d'amélioration de la qualité visuelle. Grâce à la puissance des réseaux de neurones convolutifs (CNN), les modèles modernes permettent aujourd'hui d'atteindre des performances bien supérieures aux méthodes traditionnelles, aussi bien en termes de précision que de robustesse face à différents types de bruit. Ce chapitre s'intéresse à l'impact des modèles de deep learning sur la restauration d'images, en se concentrant sur quatre architectures majeures : DnCNN, ARCNN, FFDNet et DRUNet. Chacun de ces modèles présente des stratégies uniques pour extraire, apprendre et corriger les dégradations visuelles dans les images numériques.

Par ailleurs, dans un contexte de miniaturisation des systèmes et d'applications embarquées (comme la vidéosurveillance, les drones ou les dispositifs portables), l'implémentation de ces modèles sur des plateformes à faible consommation et à faible puissance de calcul, telles que le Raspberry Pi, soulève de nouvelles problématiques. Le défi est alors de concilier les exigences computationnelles élevées de ces architectures avec les contraintes matérielles limitées du Raspberry Pi, tout en conservant des performances acceptables.

Ce chapitre vise donc à explorer en profondeur les caractéristiques et les architectures de ces modèles de débruitage d'image, à comparer leurs avantages respectifs, et à analyser leur faisabilité sur des plateformes embarquées. À travers cette étude, nous cherchons à évaluer dans quelle mesure les avancées du deep learning peuvent être rendues accessibles dans des environnements contraints, tout en conservant leur efficacité.

2. Définition des modèles de deep learning :

Le deep learning, ou apprentissage profond, est une branche du machine learning (apprentissage automatique) qui repose sur l'utilisation de réseaux de neurones artificiels profonds (deep neural networks). Un modèle de deep learning est donc un système informatique inspiré du fonctionnement du cerveau humain, constitué de plusieurs couches de neurones interconnectés, capable d'apprendre automatiquement à partir de données massives.

3. Structure générale d'un modèle de deep learning :

Un modèle de deep learning est typiquement composé de :

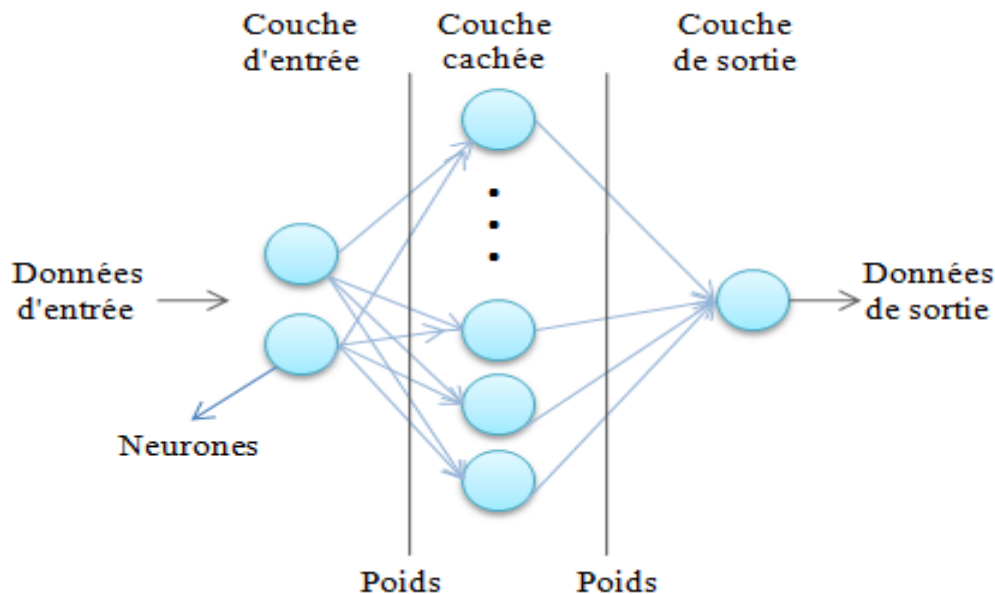


Figure 11: Structure générale d'un modèle du deep learning

- **Une couche d'entrée** : qui reçoit les données brutes (par exemple, une image).
- **Des couches cachées** : où se font les transformations et extractions de caractéristiques. Plus ces couches sont nombreuses et complexes, plus le modèle est "profond".
- **Une couche de sortie** : qui produit le résultat (ex. : image restaurée, classification, etc.).

Chaque couche contient des neurones artificiels qui effectuent des opérations mathématiques sur les données (produits scalaires, convolutions, activations, etc.).

4. Impact des modèles de deep learning sur la restauration d'images :

L'introduction des modèles de deep learning a profondément transformé le domaine de la restauration d'images, en offrant des performances largement supérieures aux méthodes traditionnelles basées sur des filtres ou des modèles statistiques.

Amélioration significative de la qualité des images restaurées

Les modèles profonds, en particulier les réseaux de neurones convolutifs (CNN), sont capables de :

- Supprimer le bruit, même à des niveaux élevés.
- Corriger les flous complexes dus au mouvement ou à une mauvaise mise au point.
- Réduire les artefacts de compression, notamment ceux causés par le format JPEG.

- Récupérer des détails fins, parfois perdus dans l'image d'origine.

Résultat : les images restaurées sont plus nettes, plus naturelles et plus fidèles, tant sur le plan visuel que sur les métriques objectives (PSNR, SSIM, etc.).

➤ **Automatisation et généralisation:**

Contrairement aux approches classiques qui nécessitaient :

- Des réglages manuels (filtre, intensité du débruitage...),
- Une connaissance précise du type de dégradation,

Les modèles de deep learning peuvent être entraînés sur de larges bases de données et généraliser à des images ou des types de bruit jamais vus auparavant.

Un seul modèle (comme DnCNN ou FFDNet) peut traiter plusieurs niveaux de bruit, ou combiner plusieurs types de dégradations.

➤ **Rapidité et efficacité (inférence en temps réel):**

Une fois entraînés, ces modèles peuvent être exécutés en temps réel sur GPU, ce qui les rend adaptés à des applications critiques :

- Surveillance vidéo,
- Imagerie médicale en direct,
- Photographie sur smartphone.

➤ **Accélération de la recherche et de l'innovation:**

Le deep learning a aussi ouvert la voie à :

- Des architectures nouvelles (UNet, GAN, Transformer),
- Des approches non supervisées ou auto-supervisées,
- L'intégration de la perception humaine via des pertes perceptuelles (ex. : VGG, LPIPS).

5. Les différents algorithmes de Deep Learning :

Plusieurs types d'algorithmes sont utilisés en apprentissage profond. Chaque algorithme possède ses propres spécificités et applications [7].

5.1 Réseaux neuronaux convolutifs (CNN) :

Également appelés ConvNets, les CNN sont constitués d'une multitude de couches chargées de traiter et d'extraire les caractéristiques des données. De manière spécifique, les réseaux neuronaux convolutifs sont utilisés pour l'analyse et la détection d'objets. Ils peuvent donc servir par exemple à reconnaître des images satellites, traiter des images médicales, détecter des anomalies ou prédire des séries chronologiques.

5.2 Réseaux neuronaux récurrents (RNN) :

Les réseaux neuronaux récurrents possèdent des connexions qui constituent des cycles dirigés. Cela permet aux sorties du LSTM d'être exploitées comme entrées au niveau de la phase actuelle. La sortie du LSTM se transforme en une entrée pour la phase actuelle. Elle peut donc mémoriser les entrées précédentes à l'aide de sa mémoire interne. Dans la pratique, les RNN sont utilisés pour le sous-titrage d'images, le traitement du langage naturel et la traduction automatique.

5.3 Réseaux de fonction de base radiale (RBFN) :

Ces algorithmes sont des réseaux neuronaux feedforward assez particuliers. Ils exploitent des fonctions de base radiales en tant que fonctions d'activation. Ils sont constitués d'une couche d'entrée, d'une couche cachée et d'une couche de sortie. Généralement, les RBFN sont utilisés dans la classification, la prédiction des séries temporelles et la régression linéaire.

5.4 Réseaux de mémoire à long et court terme (LSTM) :

Les LSTM sont des dérivés de RNN. Ils peuvent apprendre et mémoriser des dépendances sur une longue durée. Les LSTM conservent ainsi les informations mémorisées sur le long terme. Ils sont particulièrement utiles pour prédire des séries chronologiques, car ils se rappellent des entrées précédentes. Outre ce cas d'utilisation, les LSTM sont également utilisés pour composer des notes de musique et reconnaître des voix.

5.5 Réseaux adversariaux génératifs (GAN) :

Les GAN créent de nouvelles instances de données qui s'apparentent aux données d'apprentissage profond. Ils possèdent deux principaux composants : un générateur et un discriminateur. Si le générateur apprend à produire des informations erronées, le discriminateur, quant à lui, apprend à exploiter ces fausses informations. Les GAN sont généralement utilisés par les créateurs de jeux vidéo pour améliorer les textures 2D.

5.6 Machines de Boltzmann restreintes (RBM) :

C'est le professeur Geoffrey Hinton qui a développé cet algorithme. En d'autres termes, les machines de Boltzmann restreintes sont des réseaux neuronaux stochastiques constitués de deux couches : unités visibles et unités cachées. Ces réseaux artificiels sont capables d'apprendre en partant d'une distribution de probabilité sur un ensemble d'entrées. Néanmoins, il est important de souligner que la liste d'algorithmes présentés ci-dessus n'est pas exhaustive. Il en existe en effet d'autres types comme :

- Les auto-encodeurs,
- Les réseaux de croyance profonds (DBN),
- Les perceptrons multicouches (MLP).

Pour finir, les cartes auto-organisées (SOM) inventées par le professeur Teuvo Kohonen sont également une forme spécifique d'algorithmes d'apprentissage profond. Pour apprendre à utiliser ces différents algorithmes, l'idéal est de suivre une formation en data.

6. Principaux modèles de deep learning pour la restauration d'images :

6.1 DnCNN (Denoising Convolutional Neural Network) :

6.1.1 Définition:

DnCNN (Denoising Convolutional Neural Network) est un modèle de deep learning basé sur un réseau convolutionnel profond, conçu pour effectuer la restauration d'images, notamment le débruitage. Il repose sur l'apprentissage résiduel, c'est-à-dire qu'il apprend à estimer le bruit présent dans l'image, afin de le soustraire de l'image bruitée et ainsi obtenir une version restaurée.

Ce modèle se distingue par son efficacité à traiter différents niveaux de bruit avec un seul réseau et par ses performances supérieures aux méthodes traditionnelles comme BM3D.

6.1.2 Objectif du DnCNN:

Le DnCNN (Denoising Convolutional Neural Network) est un réseau de neurones profonds conçu pour :

- Supprimer le bruit (surtout gaussien) dans les images
- Restaurer la qualité visuelle de l'image
- Utiliser un apprentissage résiduel pour estimer le bruit au lieu de l'image directement

6.1.3 Architecture :

Le réseau est constitué de D couches (souvent $D = 17$ ou 20), et est divisé en 3 blocs fonctionnels :

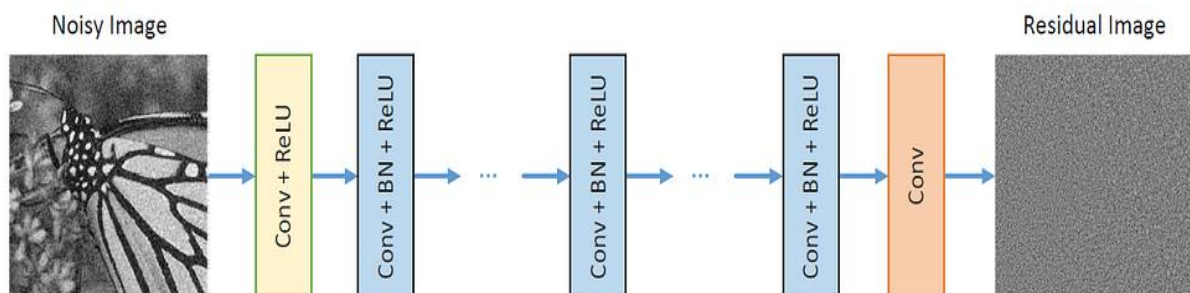


Figure 12: Architecture de DnCNN

❖ Bloc 1 – Entrée (Conv + ReLU):

La première couche du réseau est une couche de convolution suivie d'une activation ReLU (Rectified Linear Unit). Cette couche utilise 64 filtres de taille $3 \times 3 \times c$, où c représente le nombre de canaux de l'image d'entrée : $c = 1$ pour une image en niveaux de gris, et $c = 3$ pour une image couleur. Cette couche sert à extraire les caractéristiques de bas niveau telles que les bords, textures et gradients. La fonction ReLU introduit de la non-linéarité, ce qui permet au réseau de modéliser des relations complexes dans les données.

❖ Bloc 2 – Couches cachées (Conv + BatchNorm + ReLU):

Les couches intermédiaires vont de la 2^e à la $(D-1)^e$ couche. Chaque couche est composée de trois opérations successives : une convolution avec 64 filtres $3 \times 3 \times 64$, suivie d'une normalisation par lots (Batch Normalization), puis d'une activation ReLU. La batch normalization stabilise et accélère l'entraînement du réseau en normalisant les activations à chaque mini-lot, ce qui réduit le décalage de covariances interne. La ReLU, quant à elle, renforce la capacité d'apprentissage non-linéaire du réseau. Ces couches sont responsables de l'extraction de caractéristiques de plus haut niveau dans l'image bruitée, permettant au réseau de bien modéliser les structures complexes du bruit.

❖ Bloc 3 – Sortie (Conv):

La dernière couche du réseau est une simple couche de convolution, sans activation. Elle utilise c filtres de taille $3 \times 3 \times 64$ pour générer la sortie du réseau, qui correspond à une estimation du bruit présent dans l'image. Cette sortie est ensuite soustraite à l'image bruitée d'entrée pour obtenir une image débruitée. L'absence de fonction d'activation à cette étape permet une sortie en valeurs réelles, ce qui est nécessaire pour estimer précisément le bruit.

• Autres éléments importants de l'architecture :

Zéro-padding : est utilisé dans chaque couche de convolution pour préserver la taille spatiale de l'image tout au long du réseau, évitant ainsi les artefacts de bordure.

Pas de couche de pooling : est utilisé, car cela modifierait la résolution de l'image, ce qui n'est pas souhaitable pour le débruitage.

Le **champ récepteur** : du réseau augmente avec la profondeur. Il est défini comme :

$$\text{Champ récepteur} = (2d+1) \times (2d+1) \dots\dots\dots (4)$$

Par exemple, pour une profondeur $d = 17$, le champ récepteur est de 35×35 pixels, ce qui permet au réseau d'avoir un contexte global suffisant pour traiter le bruit de manière efficace.

6.1.4 Caractéristiques du modèle DnCNN :

Le modèle DnCNN se distingue par plusieurs caractéristiques techniques et conceptuelles qui le rendent particulièrement efficace pour les tâches de restauration d'image, notamment le débruitage. Ces caractéristiques concernent sa structure, sa stratégie d'apprentissage, ses performances, et sa généralisation à d'autres tâches.

▪ Apprentissage résiduel :

L'une des innovations clés de DnCNN réside dans l'usage de l'apprentissage résiduel. Au lieu d'apprendre directement l'image propre à partir de l'image bruitée, le réseau apprend à estimer le bruit présent dans l'image. L'image restaurée est obtenue en soustrayant ce bruit estimé de l'image d'entrée. Cette approche rend le problème plus simple à optimiser et améliore la stabilité de l'entraînement. Elle permet également d'accélérer la convergence du modèle lors de l'entraînement.

▪ Architecture entièrement convolutive :

DnCNN adopte une architecture entièrement convolutive, sans aucune couche de mise en sous-échantillonnage (pooling). Cette structure permet de conserver la résolution spatiale de l'image à toutes les étapes, ce qui est essentiel dans les tâches de débruitage où chaque pixel est important. De plus, les couches convolutives utilisent un padding adéquat (zéro-padding) afin de maintenir les dimensions constantes tout au long du réseau.

▪ Utilisation de la Batch Normalization :

Le modèle intègre la Batch Normalization (BN) après chaque couche convolutive (sauf la première et la dernière). La BN permet de stabiliser les activations au sein du réseau en normalisant les sorties de chaque mini-lot pendant l'entraînement. Cela a pour effet de réduire le problème du covariate shift interne, d'accélérer l'apprentissage et d'améliorer la généralisation du réseau.

▪ Activation ReLU pour la non-linéarité :

Les couches intermédiaires sont suivies d'une fonction d'activation ReLU (Rectified Linear Unit), définie comme $f(x) = \max(0, x)$. Cette fonction introduit de la non-linéarité dans le réseau tout en évitant les effets de saturation que l'on rencontre avec des fonctions comme tanh ou sigmoid. ReLU favorise également une meilleure propagation du gradient.

▪ Champ récepteur élargi :

Grâce à sa profondeur (souvent 17 à 20 couches), DnCNN dispose d'un champ récepteur élargi, ce qui lui permet de capturer un contexte spatial étendu autour de chaque pixel. Cela est essentiel pour comprendre les structures globales et locales dans l'image et pour mieux identifier les motifs de bruit.

▪ Un seul modèle pour plusieurs niveaux de bruit :

Une caractéristique remarquable de DnCNN est sa capacité à traiter plusieurs niveaux de bruit avec un seul modèle. Alors que les méthodes traditionnelles de débruitage nécessitent souvent un modèle spécifique pour chaque niveau de bruit (σ), DnCNN peut être entraîné pour gérer une plage de niveaux de bruit (ex. : $\sigma \in [0, 50]$), ce qui augmente sa flexibilité et sa praticité dans des scénarios réels.

▪ Excellentes performances en PSNR et SSIM :

En termes de performance, DnCNN surpasse largement les méthodes classiques telles que BM3D ou WNNM sur des métriques standard comme le PSNR (Peak Signal-to-Noise Ratio) et le SSIM (Structural Similarity Index). Il est considéré comme une référence dans les tâches

de débruitage d'image, aussi bien pour le bruit gaussien que pour d'autres types de perturbations.

6.2 ARCNN (Artifacts Reduction CNN) :

6.2.1 Définition:

ARCNN est un modèle de deep learning basé sur les réseaux de neurones convolutifs (CNN) conçu spécifiquement pour réduire les artefacts visuels introduits par la compression JPEG [8]. Son objectif principal est d'améliorer la qualité visuelle des images compressées, en supprimant trois types majeurs de dégradations :

- **Bloc** : discontinuités entre les blocs 8×8 .
- **Effets de résonance** : ondulations autour des bords nets.
- **Flou** : perte de netteté due à la suppression des hautes fréquences.

6.2.2 Objectifs de ARCNN :



Figure 13: Objectif de ARCNN

- Réduire les artefacts de compression JPEG
- Restaurer la netteté et les détails de l'image
- S'appuyer sur une architecture CNN peu profonde pour une exécution rapide

6.2.3 Architecture du ARCNN :

Le réseau ARCNN est composé de 4 couches principales, avec une architecture typique : (9–7–1–5) → les chiffres correspondent à la taille des filtres de chaque couche.

La figure ci-dessus présente visuellement les différentes étapes du flux de traitement de l'image.

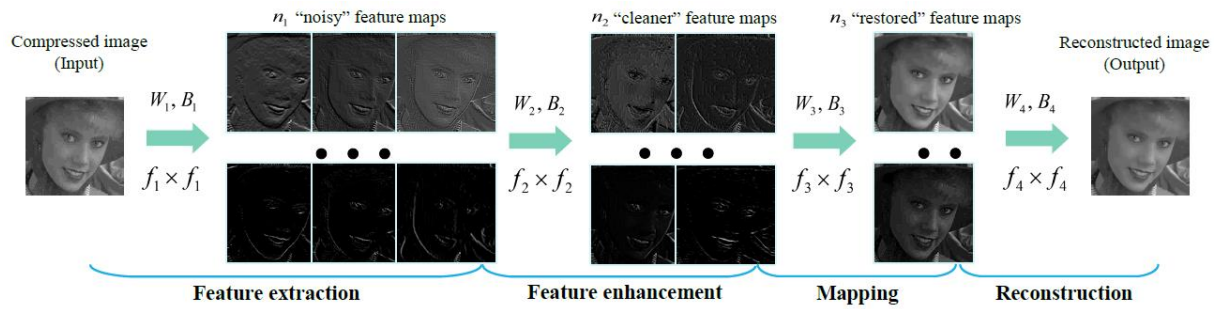


Figure 14: Architecture de ARCNN

❖ Couche 1 : Extraction (Extraction des caractéristiques)

La première couche de AR-CNN est responsable de l'extraction des caractéristiques de bas niveau à partir de l'image JPEG compressée. Elle utilise généralement 64 filtres de taille 9×9 pour parcourir l'image et détecter des motifs locaux tels que les bords, les textures ou encore les artefacts de bloc. Cette grande taille de filtre permet de capturer un contexte suffisamment large autour de chaque pixel, ce qui est crucial pour détecter les structures déformées. Cette couche est souvent suivie d'une fonction d'activation ReLU afin d'introduire de la non-linéarité dans le processus.

❖ Couche 2 : Feature Enhancement (Amélioration des caractéristiques)

La deuxième couche joue un rôle clé : elle vise à nettoyer et améliorer les caractéristiques extraites précédemment. Cette couche utilise 32 filtres de taille 7×7 , ce qui permet de raffiner les informations et de supprimer les éléments redondants ou bruités. Cette opération est essentielle pour atténuer les distorsions introduites lors de la compression JPEG. Le nom "enhancement" (amélioration) reflète bien le fait qu'elle agit comme un filtre intelligent qui renforce les bonnes structures tout en supprimant les artefacts. Une activation ReLU est également utilisée ici pour maintenir la non-linéarité.

❖ Couche 3 : Mapping (Cartographie non linéaire)

La troisième couche, dite de "mapping", agit comme un transformateur de caractéristiques. Elle prend les cartes de caractéristiques raffinées de la couche précédente et les projette dans un espace sémantique plus abstrait. Elle utilise souvent 16 filtres de taille 1×1 , ce qui permet de combiner les différentes cartes de caractéristiques sans changer la taille spatiale de l'image. Le filtre 1×1 permet aussi de réduire le nombre de paramètres, ce qui est avantageux pour la légèreté du modèle. Cette couche permet d'effectuer une transformation non linéaire locale, essentielle pour représenter la relation complexe entre les pixels dégradés et les pixels restaurés.

❖ Couche 4 : Reconstruction

Enfin, la quatrième couche est chargée de la reconstruction de l'image finale restaurée. Elle utilise une seule carte de filtre de taille 5×5 pour fusionner toutes les caractéristiques extraites et améliorées, et produire l'image de sortie. Cette sortie est une version "nettoyée" de l'image JPEG, avec une réduction significative des artefacts visuels. Contrairement aux couches précédentes, aucune activation ReLU n'est appliquée ici, car l'objectif est d'avoir une sortie linéaire représentant directement les valeurs de pixels de l'image finale.

6.2.4 Caractéristiques de AR-CNN :

Le modèle AR-CNN, proposé par Chao Dong et al. en 2015, a été conçu pour réduire les artefacts visuels causés par la compression JPEG [9]. Il s'appuie sur un réseau de neurones convolutionnels profond, spécifiquement adapté à la restauration d'image dans un contexte de compression. Voici ses principales caractéristiques, chacune accompagnée d'une explication précise :

▪ Spécialisation pour les artefacts JPEG :

AR-CNN est spécialement conçu pour traiter les artefacts de compression tels que :

- Blocs visibles dus à la compression par blocs 8×8 .
- Résonance près des bords nets causée par la quantification excessive des hautes fréquences.
- Flou global lié à la perte d'informations haute fréquence.

Cela le distingue des modèles généraux de débruitage : il est optimisé pour des dégradations spécifiques à JPEG.

▪ Architecture profonde à quatre couches convolutives :

Le réseau est constitué de quatre couches convolutives (Extraction, Amélioration, Mapping, Reconstruction), chacune remplissant un rôle fonctionnel précis dans la réduction des artefacts. Cette architecture est plus complexe que celle du SRCNN, mais reste suffisamment légère pour être utilisée en pratique, même sur des images de grandes tailles.

▪ Utilisation de filtres de tailles variées :

Chaque couche utilise des filtres de tailles différentes :

- 9×9 pour l'extraction des caractéristiques
- 7×7 pour leur amélioration
- 1×1 pour la projection (mapping)
- 5×5 pour la reconstruction

Ce choix permet au modèle de capturer à la fois le contexte global et les détails locaux, tout en maintenant un nombre de paramètres raisonnable.

▪ Apprentissage de bout en bout :

AR-CNN est entraîné de manière end-to-end, ce qui signifie que toutes les couches sont optimisées simultanément pour minimiser la perte entre l'image compressée et l'image de référence non compressée. Cette approche améliore la cohérence du modèle et la performance globale.

▪ Amélioration visuelle et numérique

Le modèle a démontré une amélioration significative en PSNR (Peak Signal-to-Noise Ratio), mais aussi en qualité visuelle perceptible. Par exemple :

- SRCNN (9–1–5) donne 28,91 dB sur des images JPEG de qualité 10

- AR-CNN(9–7–1–5) atteint 28,98dB Cela peut sembler faible en chiffre, mais visuellement, la différence est marquante, surtout près des contours.
- **Support du transfert d'apprentissage :**

Une des forces de AR-CNN est sa capacité à transférer l'apprentissage :

- D'un modèle peu profond vers un modèle plus profond.
- D'images de haute qualité vers des images de basse qualité.
- De scénarios standards vers des cas réels (ex : images compressées sur Twitter).

Cela permet de réduire le temps d'entraînement tout en améliorant la performance sur des situations variées.

- **Extension vers une version rapide (Fast AR-CNN) :**

Une version optimisée du modèle, Fast AR-CNN, a été proposée. Elle :

- Utilise des couches 1×1 pour réduire les paramètres.
- Réduit le temps d'inférence (0,067 s/image contre 0,5 s pour AR-CNN).
- Maintient une qualité visuelle proche de l'original.

Cela rend le modèle plus adapté à des systèmes embarqués ou des applications temps réel.

- **Pas de pooling, pas de padding :**

Le réseau AR-CNN ne contient pas de couches de pooling. Cela permet de préserver la résolution spatiale exacte de l'image d'entrée jusqu'à la sortie. De plus, dans la version d'origine, il n'y avait pas ou peu de padding, ce qui impliquait une sortie légèrement plus petite, mais plus précise sur les pixels prédits.

6.3 FFDNet (Fast and Flexible Denoising Network) :

6.3.1 Définition:

FFDNet est un modèle de débruitage d'images basé sur les réseaux de neurones convolutifs (CNN) développé par Kai Zhang et al. en 2018 [10]. Il a été conçu pour être à la fois flexible (pouvant traiter différents niveaux de bruit avec un seul modèle) et rapide (adapté aux applications en temps réel). Contrairement à d'autres modèles comme DnCNN qui nécessitent un modèle distinct pour chaque niveau de bruit, FFDNet introduit une carte de bruit comme entrée supplémentaire, permettant de débruiter des images de niveaux de bruit variables avec le même réseau.

6.3.2 Architecture de FFDNet :

L'architecture de FFDNet repose sur un réseau convolutionnel profond, conçu spécifiquement pour le débruitage d'images à différents niveaux de bruit, tout en assurant une

haute efficacité computationnelle. Contrairement à d'autres réseaux de débruitage classiques, FFDNet introduit une approche novatrice en intégrant une carte de bruit en tant qu'entrée du réseau, permettant ainsi une adaptation dynamique aux variations locales de bruit.

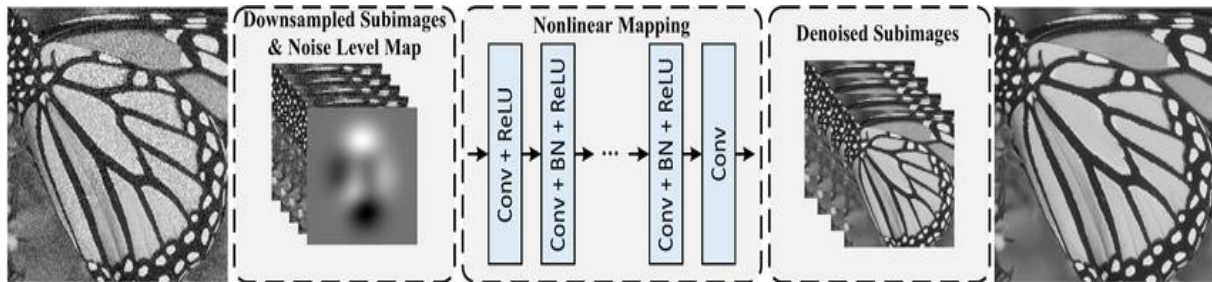


Figure 15 : Architecture de FFDNet

❖ Entrée du réseau : concaténation image + carte de bruit :

La première étape de l'architecture consiste à prendre en entrée l'image bruitée ainsi qu'une carte de bruit σ (sigma map), qui indique la force du bruit dans chaque région de l'image. Cette carte peut être uniforme (même niveau de bruit partout) ou non uniforme (niveau variable). Contrairement aux méthodes classiques qui nécessitent plusieurs modèles selon le bruit, FFDNet intègre cette information dans son entrée. L'image et la carte de bruit sont concaténées en profondeur, c'est-à-dire que chaque pixel est représenté par un vecteur multidimensionnel, combinant l'intensité et le niveau de bruit local.

❖ Sous-échantillonnage (Downsampling) :

Avant d'entrer dans les couches convolutives principales, l'image est sous-échantillonnée par blocs de 2×2 . Cette opération est réalisée à la fois sur l'image et la carte de bruit. Le but de cette étape est de réduire la taille spatiale de l'image (divisée par 4), tout en augmentant la profondeur (nombre de canaux), ce qui améliore la compacité et la vitesse du réseau. Par exemple, une image RGB est transformée de dimensions $H \times W \times 3$ en $(H/2) \times (W/2) \times 12$. Si on ajoute la carte de bruit (également découpée en 4), l'entrée du réseau devient $(H/2) \times (W/2) \times 16$.

❖ Couches convolutives internes :

Le cœur de FFDNet est constitué de 15 couches convolutives, chacune ayant une fonction spécifique dans le processus de débruitage. Les 14 premières couches appliquent une convolution 3×3 avec 64 filtres. Chacune de ces convolutions est suivie d'une Batch Normalization, qui normalise les activations pour accélérer la convergence et stabiliser l'apprentissage, et d'une fonction d'activation ReLU, qui introduit de la non-linéarité et permet de filtrer les valeurs négatives. Ces couches apprennent à extraire progressivement des caractéristiques de plus en plus complexes, allant des bords et textures de bas niveau à des représentations de bruit et de structure plus abstraites.

La 15^e couche, appelée couche de sortie, applique également une convolution 3×3 , mais cette fois avec un nombre de filtres égal au nombre de canaux de sortie souhaités ($c = 1$ pour une image en niveaux de gris, $c = 3$ pour une image couleur). Cette couche ne contient ni batch

normalization, ni fonction d'activation, car elle doit produire directement la sortie débruitée sans transformation non-linéaire supplémentaire. Le rôle de cette dernière couche est de reconstruire l'image propre à partir des caractéristiques apprises dans les couches précédentes.

❖ Reconstruction finale (Upsampling) :

Une fois l'image propre estimée dans l'espace réduit (suite au sous-échantillonnage), une opération d'agrandissement (par exemple un "PixelShuffle" ou reshape) est appliquée pour ramener l'image à sa taille d'origine. Cette opération réorganise les blocs 2×2 qui avaient été compactés au début, permettant une restitution fidèle de l'image dans sa résolution initiale. Grâce à cette technique, FFDNet évite l'usage du pooling ou du striding, ce qui est crucial pour les tâches de restauration d'image où il est important de conserver la position spatiale exacte des pixels.

6.3.3 Caractéristiques de FFDNet :

FFDNet présente plusieurs caractéristiques techniques et pratiques qui le distinguent des autres modèles de débruitage basés sur les réseaux de neurones convolutifs. Conçu pour être à la fois rapide, flexible et performant, il introduit des innovations qui le rendent efficace pour une large gamme de niveaux de bruit et de types d'images.

▪ Flexibilité au bruit variable :

L'une des caractéristiques les plus importantes de FFDNet est sa capacité à gérer différents niveaux de bruit avec un seul modèle unique. Contrairement à de nombreux modèles qui nécessitent d'entraîner plusieurs réseaux séparément pour chaque niveau de bruit ($\sigma = 15$, $\sigma = 25$, etc.), FFDNet accepte une carte de bruit en entrée. Cette carte permet au modèle d'adapter dynamiquement son comportement selon la quantité de bruit présente dans l'image, même si elle varie d'un endroit à l'autre.

▪ Vitesse élevée et efficacité computationnelle :

Grâce à l'opération de sous-échantillonnage effectuée avant les couches convolutives, FFDNet réduit la dimension spatiale de l'image d'entrée tout en augmentant sa profondeur. Cela diminue la complexité computationnelle globale du réseau, permettant une exécution plus rapide. De ce fait, FFDNet est particulièrement adapté aux systèmes temps réel ou embarqués, et affiche un temps de traitement bien inférieur à celui de modèles plus lourds comme DnCNN, tout en offrant une qualité comparable.

▪ Débruitage des images en niveaux de gris et en couleur :

FFDNet est conçu pour fonctionner aussi bien avec des images monochromes qu'avec des images RGB. Pour les images en couleurs, il traite directement les trois canaux conjointement (plutôt que canal par canal), ce qui permet de mieux préserver les structures chromatiques et d'éviter les artefacts de couleur. Cela en fait un modèle polyvalent, applicable à de nombreux cas d'usage.

▪ Formation supervisée de bout en bout :

Le réseau FFDNet est entraîné en apprentissage supervisé, où l'objectif est de minimiser la différence entre l'image propre (référence) et la sortie du réseau, souvent à l'aide d'une fonction

de perte MSE (Mean Squared Error). Ce processus apprend au réseau à reconnaître les motifs du bruit et à les retirer efficacement, tout en préservant les détails de l'image.

- **Aucune couche de pooling utilisée :**

FFDNet n'utilise aucune couche de pooling, afin d'éviter toute perte d'information spatiale. Cela est crucial pour les tâches de restauration d'image, où chaque pixel compte. L'architecture conserve la structure spatiale tout au long du réseau, garantissant une meilleure fidélité des détails lors de la reconstruction.

- **Capacité à traiter le bruit non uniforme :**

Grâce à l'entrée "carte de bruit", FFDNet est l'un des rares modèles capables de traiter efficacement les images contenant du bruit spatialement variant, c'est-à-dire des zones de bruit plus ou moins fortes selon les régions. Cette capacité est utile pour les images issues de caméras à faible luminosité ou de capteurs de qualité variable.

- **Performance élevée sur les métriques de qualité :**

Sur les jeux de données standards (BSD68, Set12, Urban100, etc.), FFDNet atteint d'excellentes performances en PSNR (Peak Signal-to-Noise Ratio) et SSIM (Structural Similarity Index), souvent proches ou supérieures à des méthodes classiques comme BM3D ou à des réseaux comme TNRD ou DnCNN, tout en étant plus rapide [11].

6.4 DRUNet (A Dilated-Residual U-Net):

6.4.1 Définition:

DRUNet (Deep Residual U-Net) est un modèle de débruitage d'images basé sur une architecture U-Net modifiée, combinée à des blocs résiduels profonds. Il est introduit dans le cadre du framework Real-ESRGAN et de l'approche DPSR (Deep Plug-and-Play Super-Resolution) pour la restauration d'images naturelles dégradées [12]. DRUNet est conçu pour être générique et modulaire, ce qui le rend applicable non seulement au débruitage, mais aussi à d'autres tâches de restauration comme le défloutage, la super-résolution, la compression d'image et l'inpainting. Il accepte une carte de bruit ou d'autres cartes de dégradation comme entrée auxiliaire, le rendant très flexible et adapté aux cas réels où les conditions de dégradation sont inconnues ou variables.

6.4.2 Architecture de DRUNet :

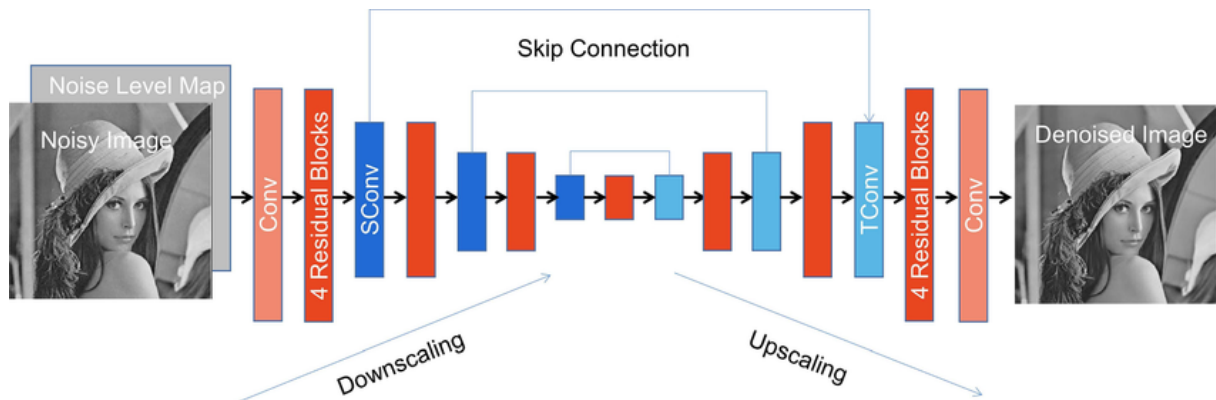


Figure 16: Architecture de DRUNet

L'architecture de DRUNet est composée de plusieurs modules hiérarchiques, organisés autour d'un schéma en forme de U. Chaque partie du réseau joue un rôle essentiel dans le traitement et la restauration des images [13]. Voici une explication approfondie de chaque composant :

- **Couche d'entrée :**

La couche d'entrée prend comme entrée une image bruitée ainsi qu'une carte de bruit sigma. Cette carte sigma représente l'intensité du bruit et est concaténée à l'image sous forme de canal supplémentaire, permettant au réseau de s'adapter aux différents niveaux de bruit. Cette entrée est ensuite traitée par une convolution 3×3 , qui augmente la dimensionnalité de l'espace des caractéristiques (typiquement à 64 canaux). Une activation ReLU est appliquée pour introduire de la non-linéarité.

- **Partie encodeur (Downsampling) :**

L'encodeur a pour rôle de réduire progressivement la taille spatiale de l'image tout en augmentant la profondeur des représentations. Il est constitué de trois niveaux, chacun comportant deux blocs résiduels suivis d'un downsampling.

Chaque bloc résiduel comprend deux convolutions 3×3 , chacune suivie d'une activation ReLU. La sortie de ces deux couches est ajoutée à l'entrée du bloc, ce qui forme une connexion résiduelle. Ces connexions permettent au réseau de mieux apprendre les différences entre l'image bruitée et l'image propre, ce qui est crucial pour les tâches de restauration.

Le downsampling est réalisé grâce à une convolution avec stride de 2, qui réduit la taille de la carte d'activation de moitié, tout en doublant le nombre de canaux. Cela permet au réseau de capturer des motifs à plus grande échelle sans augmenter la complexité computationnelle.

- **Goulot d'étranglement (Bottleneck) :**

Le goulot d'étranglement est la partie la plus profonde du réseau, où la résolution spatiale est minimale mais les représentations sont les plus riches. Il est composé de quatre blocs

résiduels consécutifs. À ce niveau, le réseau a capturé suffisamment de contexte global et peut commencer à apprendre des représentations de haut niveau abstraites.

Ce module agit comme un transformateur de l'information, où les détails locaux et globaux sont intégrés pour aider à la reconstruction précise de l'image.

- **Partie décodeur (Upsampling) :**

Le décodeur a pour mission de reconstruire l'image restaurée en augmentant progressivement la taille des cartes de caractéristiques. Chaque niveau du décodeur effectue une augmentation de la résolution spatiale (upsampling), suivi de la concaténation avec les caractéristiques correspondantes du niveau encodeur (skip connections), puis de deux blocs résiduels.

Les skip connections permettent de réintroduire les détails fins de l'image capturés dans les premières couches, ce qui est essentiel pour préserver les bords, textures et structures. L'upsampling est généralement effectué par une interpolation bilinéaire suivie d'une convolution 1×1 ou 3×3 , ce qui rend la montée en résolution plus stable et précise.

- **Couche de sortie :**

Une fois que l'image a été reconstruite dans sa taille d'origine, une dernière couche de convolution 3×3 est appliquée pour réduire le nombre de canaux à 1 (image en niveaux de gris) ou 3 (image couleur). Cette couche génère soit une carte de bruit estimé (dans le cadre d'un apprentissage résiduel), soit directement l'image restaurée. Aucune activation non-linéaire n'est appliquée à cette dernière couche, car on souhaite conserver les valeurs continues sans distorsion.

6.4.3 Caractéristiques principales de DRUNet :

DRUNet présente plusieurs caractéristiques distinctives qui expliquent ses performances exceptionnelles :

- **Blocs résiduels profonds :** permettent un apprentissage plus stable et plus rapide en facilitant la circulation des gradients.
- **Skip connections longues :** entre encodeur et décodeur, elles permettent de récupérer efficacement les détails perdus durant le downsampling.
- **Conditionnement au bruit (sigma) :** le réseau est conditionné par une carte de bruit injectée dès l'entrée, ce qui le rend adaptable à différents niveaux de bruit avec un seul modèle.
- **Structure multi-échelle (U-Net) :** la structure symétrique permet une compréhension à la fois globale et locale des images.
- **Apprentissage end-to-end :** le réseau est optimisé de bout en bout via une fonction de perte (souvent L2) entre l'image de sortie et l'image propre.

7. Contraintes liées à l'implémentation de modèles de débruitage d'image (DnCNN, ARCNN, FFDNet, DRUNet) sur Raspberry Pi :

L'implémentation de modèles de deep learning pour le débruitage d'image sur une plateforme embarquée comme le Raspberry Pi représente un défi considérable. Bien que ce mini-ordinateur soit accessible et facilement déployable, ses capacités matérielles limitées imposent plusieurs contraintes techniques pour l'exécution de réseaux neuronaux convolutifs profonds. Les modèles DnCNN, ARCNN, FFDNet et DRUNet, bien que très efficaces sur stations de travail équipées de GPU, doivent faire face à ces restrictions lorsqu'ils sont transférés vers un environnement embarqué.

7.1 Limitations en termes de mémoire et de stockage :

L'une des premières limitations rencontrées concerne la mémoire vive (RAM). Le Raspberry Pi, selon la version utilisée, offre entre 2 et 8 Go de RAM, partagée avec le système d'exploitation. Or, les modèles comme DRUNet ou FFDNet nécessitent une quantité importante de mémoire pour stocker les couches, les activations intermédiaires et les paramètres du réseau. Cela devient particulièrement problématique lorsque l'on traite des images haute résolution ou lorsqu'on utilise des batchs de données. ARCNN, étant un réseau plus compact, est moins exigeant, tout comme DnCNN, mais même ces derniers peuvent atteindre les limites de mémoire lors d'une utilisation intensive [14].

7.2 Faible puissance de calcul :

Le Raspberry Pi repose sur un processeur ARM à faible consommation énergétique, sans unités de calcul parallèles hautes performances [15]. Contrairement aux stations de travail dotées de GPU NVIDIA compatibles CUDA, le Raspberry Pi ne dispose pas de capacités d'accélération matérielle adaptées aux calculs de tenseurs complexes. En conséquence, l'exécution d'un modèle comme DRUNet peut prendre plusieurs secondes, voire minutes, par image. Des architectures plus légères comme ARCNN ou DnCNN s'exécutent plus rapidement, mais restent limitées pour des applications en temps réel.

7.3 Absence d'unité de traitement graphique (GPU) :

Un autre facteur critique est l'absence de GPU dédié. Les bibliothèques comme TensorFlow ou PyTorch utilisent normalement des GPU pour accélérer l'inférence. Sur Raspberry Pi, seul le CPU est utilisé, ce qui ralentit considérablement l'exécution. Certaines alternatives comme TensorFlow Lite ou ONNX Runtime permettent d'optimiser légèrement les performances, mais nécessitent une conversion spécifique des modèles, ce qui peut entraîner des pertes de précision ou des incompatibilités.

7.4 Complexité logicielle et compatibilité ARM :

L'installation de frameworks de deep learning sur l'architecture ARM du Raspberry Pi requiert des versions spécifiques [16]. Certaines fonctions, telles que les opérations de normalisation par lot ou les convolutions transposées, peuvent ne pas être pleinement supportées ou entraînent des ralentissements importants. Il est également difficile de compiler certains modèles avancés comme DRUNet sans ajustements techniques, tandis que des modèles plus simples comme ARCNN sont plus faciles à adapter.

7.5 Problèmes thermiques et énergétiques :

Les calculs prolongés de modèles profonds peuvent entraîner une surchauffe du Raspberry Pi, surtout sans système de refroidissement actif. Cette surchauffe provoque une réduction automatique de la fréquence du processeur (thermal throttling), ce qui ralentit encore davantage le traitement. Les modèles les plus complexes, notamment DRUNet, exacerbent ce problème, tandis que ARCNN et DnCNN sont plus adaptés à une exécution prolongée.

7.6 Dépendance à l'entraînement externe :

Le Raspberry Pi ne permet pas l'entraînement des modèles, en raison de sa faible puissance de calcul. Ainsi, tous les modèles doivent être entraînés au préalable sur une machine plus puissante, puis exportés vers le Raspberry Pi pour l'inférence [17]. Cela suppose un pipeline de développement bien défini, incluant la quantisation, la compression et parfois la simplification du modèle.

7.7 Résolution des images et contraintes d'E/S :

Le traitement d'images haute résolution implique un usage important de la mémoire et des entrées/sorties. Le débit du Raspberry Pi, limité en lecture/écriture depuis la carte SD ou une clé USB, peut devenir un goulet d'étranglement. De plus, la gestion de formats lourds tels que PNG ou TIFF peut ralentir les performances.

8. Discussion :

Au terme de ce chapitre, il apparaît clairement que les modèles de deep learning tels que DnCNN, ARCNN, FFDNet et DRUNet ont révolutionné la restauration d'images en apportant une amélioration significative de la qualité visuelle des images dégradées. Chacun de ces modèles se distingue par son architecture et ses mécanismes d'apprentissage spécifiques : DnCNN par sa profondeur et son apprentissage résiduel, ARCNN par sa simplicité et sa spécialisation dans la réduction des artefacts JPEG, FFDNet par sa capacité à gérer des niveaux de bruit variables grâce à un module adaptatif, et DRUNet par son efficacité multi-échelle et sa structure U-Net avancée.

Cependant, l'implémentation de ces modèles sur des dispositifs limités tels que le Raspberry Pi présente plusieurs défis techniques : la faible puissance de calcul, la mémoire limitée, l'absence de GPU dédié, et la nécessité d'optimiser les performances pour des applications temps réel. Ces contraintes nécessitent des compromis entre précision, temps de traitement et consommation de ressources.

Ainsi, bien que l'utilisation de ces modèles sur Raspberry Pi soit envisageable, elle exige une sélection rigoureuse du modèle, une réduction de complexité, et éventuellement le recours à des techniques comme la quantization, le pruning ou l'utilisation d'accélérateurs externes. Ces constats ouvrent la voie à de futures recherches sur l'adaptation des architectures de deep learning à des environnements embarqués, où l'équilibre entre performance et efficacité reste un enjeu central.

Chapitre 3 : Application et Résultats

1. Préambule :

Dans ce chapitre, nous présentons la démarche suivie pour la réalisation de ce travail, en partant de la description de la méthodologie adoptée jusqu'à l'analyse des résultats obtenus. Nous expliquons d'abord les choix techniques retenus, notamment le modèle ARCNN, la préparation des données et la mise en place de l'environnement d'entraînement. Nous détaillons ensuite les étapes de test et d'évaluation à l'aide d'indicateurs objectifs comme le PSNR et le SSIM, complétées par une analyse qualitative à travers des comparaisons visuelles. Enfin, une discussion générale vient interpréter les performances du modèle, souligner ses avantages et ses limites, et dégager des perspectives d'amélioration.

2. Sélection du modèle de Deep Learning :

2.1 Comparaison des architectures adaptées :(DnCNN, ARCNN, FFDNet, DRUNet) :

Les modèles de deep-learning appliqués à la restauration d'images ont considérablement évolué ces dernières années, chacun se distinguant par sa structure, ses objectifs et ses performances. Parmi eux, DnCNN, ARCNN, FFDNet et DRUNet sont parmi les plus reconnus. Le tableau ci-dessous représente une analyse comparative entre les quatre modèles :

Critère	DnCNN	ARCNN	FFDNet	DRUNet
Année	2016	2016	2018	2020
Tâche principale	Débruitage gaussien, JPEG, super-résolution	Suppression d'artefacts de compression JPEG [19]	Débruitage avec bruit variable [20]	Débruitage avancé multi-bruit
Profondeur du réseau	17 à 20 couches [18]	4 couches convolutives	Environ 15 couches (avec bruit en entrée)	U-Net profond (plus de 50 couches si on compte les branches) [21]
Principe clé	Apprentissage résiduel	Apprentissage direct (end-to-end)	Concaténation image + bruit sigma comme entrée	U-Net avec modules résiduels (ResBlocks)
Entrée du modèle	Image bruitée	Image compressée	Image + carte sigma du bruit	Image bruitée
Traitement des artefacts	Moyen	Spécifiquement conçu pour JPEG	Très bon mais générique	Excellente performance
Consommation mémoire	Moyenne	Faible (architecture légère)	Moyenne à faible	Élevée (U-Net lourd)
Adaptabilité raspberry pi	Moyenne	Bonne (réseau compact)	Bonne (surtout version grayscale)	Faible (trop lourd sans optimisation)
Vitesse d'inférence	Rapide	Très rapide	Rapide	Lente sans GPU
PSNR/SSIM (qualité)	Excellente (surtout en bruit gaussien)	Bonne pour JPEG	Très bonne	Supérieure à tous (mais plus lent)

Tableau1 : Analyse comparative entre les modèles de Deep Learning

2.2 Choix du modèle : ARCNN

On a choisi le modèle ARCNN pour les raisons suivantes :

- **Spécialisation JPEG :**

ARCNN est spécifiquement conçu pour la restauration des images compressées en JPEG, ce qui correspond parfaitement à mon projet [22]. Contrairement à d'autres modèles plus généralistes, il cible directement les artefacts de compression.

- **Architecture légère et simple :**

Avec seulement 4 couches, il est beaucoup plus léger que DRUNet ou DnCNN, ce qui facilite l'entraînement rapide et l'inférence en temps réel, même sur des plateformes modestes comme le Raspberry Pi.

- **Performance satisfaisante :**

Bien qu'il ne surpasse pas DRUNet en termes de PSNR, ARCNN offre une qualité de restauration plus que suffisante pour les images JPEG compressées à bas débit, avec un temps d'inférence rapide.

- **Facilité de mise en œuvre et d'entraînement :**

L'architecture simple facilite la compréhension, l'implémentation, et la personnalisation du modèle selon les besoins du projet, en particulier dans le cadre d'un mémoire de fin d'études.

- **Compatibilité embarquée :**

Son faible besoin en calcul permet de l'implémenter efficacement sur un Raspberry Pi, ce qui n'est pas le cas de modèles plus lourds comme DRUNet.

3. Collecte et préparation des données :

3.1 MS COCO (Microsoft Common Objects in Context):

MS COCO est un vaste ensemble de données d'images introduit par Microsoft, destiné à l'entraînement et à l'évaluation de modèles de vision par ordinateur. Son objectif principal est de fournir des images réalistes représentant des objets du quotidien dans leur contexte naturel, afin de mieux simuler des conditions réelles d'utilisation [23].

La base MS COCO est principalement utilisée pour des tâches telles que :

- La détection d'objets
- La segmentation sémantique et instance-level.
- La légende d'image (image captioning).
- La reconnaissance de poses humaines.

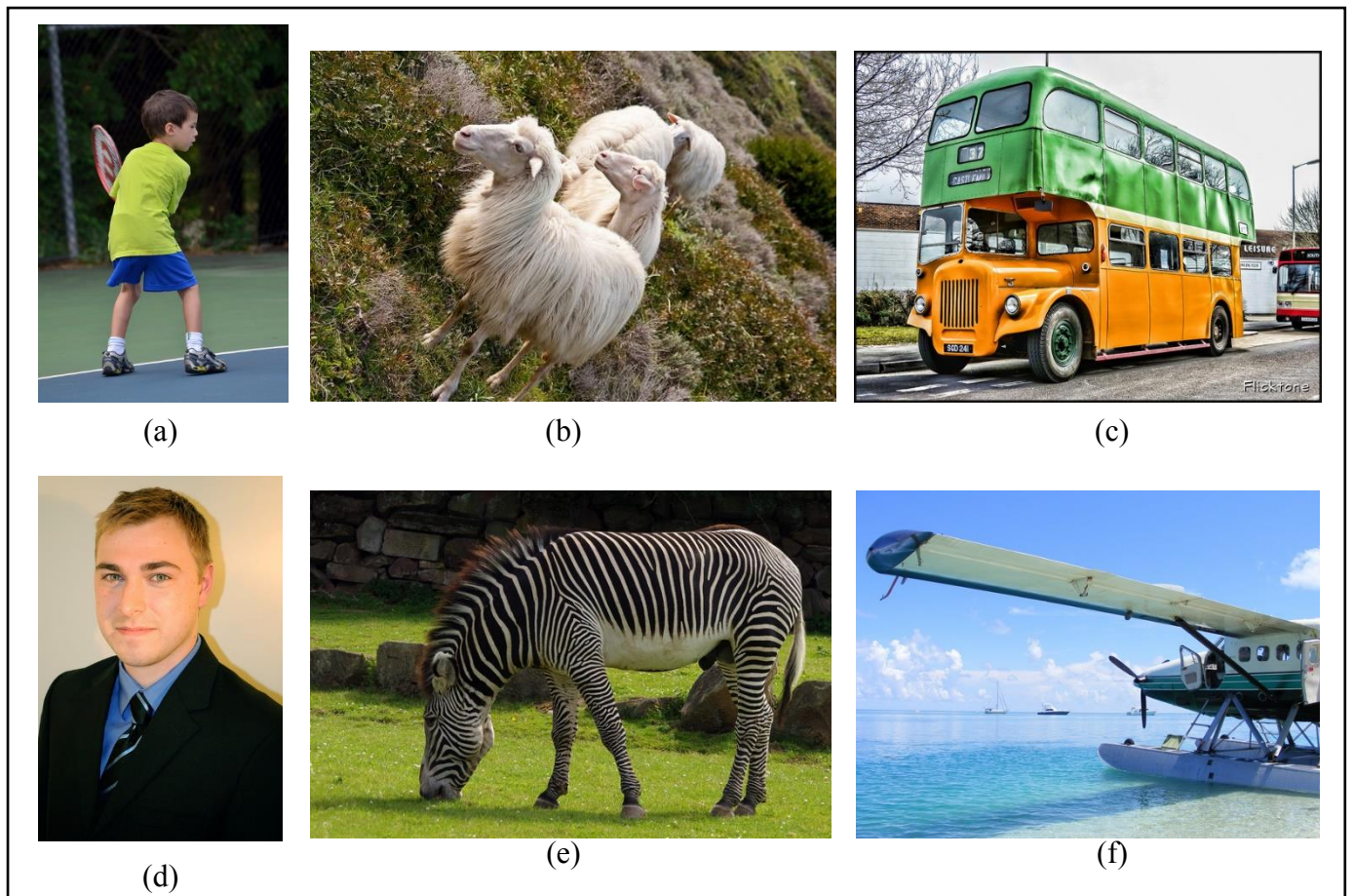


Figure 17 : Exemples d'images de la base de données MS COCO (train2014)

(a) COCO_train2014_000000015002 | (b) COCO_train2014_000000021772

(c) COCO_train2014_000000023107 | (d) COCO_train2014_0000000085624

(e) COCO_train2014_000000049987 | (f) COCO_train2014_000000025559

- **Caractéristiques principales :**

- Plus de 330 000 images, dont plus de 200 000 sont annotées [24].
- 80 catégories d'objets (personnes, animaux, véhicules, mobilier, nourriture, etc.).
- Chaque image peut contenir plusieurs objets annotés, avec des masques de segmentation précis.
- Les images proviennent de sources variées, avec un fort réalisme (milieux urbains, intérieurs, extérieurs, de jour ou de nuit).
- Le volume et la diversité des images COCO en font un excellent complément à des bases plus petites comme LIVE1 ou BSD500, notamment pour améliorer la généralisation du modèle sur des scènes complexes et variées.

Dans le cadre de la restauration d'images compressées, certaines images du set MS COCO (notamment COCO train2014) peuvent être utilisées en tant que base de données naturelle. Ce sous-ensemble contient 82 783 images différentes, couvrant une grande variété de scènes et d'objets du quotidien, ce qui en fait une ressource de référence pour l'entraînement et l'évaluation de modèles de traitement d'images.

3.2 Prétraitement des images :

Avant l'entraînement du modèle, les images de la base de données personnalisée ont fait l'objet d'un prétraitement rigoureux afin de garantir la cohérence des données et d'optimiser l'apprentissage.

3.2.1 Redimensionnement :

Les images issues de la base de données MS COCO présentent des résolutions variables, souvent très élevées, allant par exemple de 640×480 , 480×640 , 1024×768 , et pouvant dépasser les 2000 pixels de largeur dans certains cas [Lin et al., 2014]. Afin d'assurer une compatibilité avec les capacités de calcul du modèle et de normaliser les entrées du réseau, toutes les images (originales, copies et compressées) ont été redimensionnées à une taille standard de 256×256 . Ce redimensionnement a été réalisé en veillant à éviter toute déformation, soit par la conservation du ratio d'aspect avec ajout éventuel de bandes latérales, soit par un recadrage centré, permettant ainsi de préserver les zones les plus représentatives du contenu visuel.

Voici le scripte utilisé pour la Redimensionne :

```
1  from PIL import Image
2  import os
3
4  for i in range(2, 20):
5      input_folder = f"data/train/original/copy_{i}"
6      output_folder = f"data/train/images_redimensionnees_{i}"
7
8      os.makedirs(output_folder, exist_ok=True)
9
10     target_size = (256, 256)
11     image_extensions = ('.jpg', '.jpeg', '.png', '.bmp', '.tiff')
12
13     for filename in os.listdir(input_folder):
14         if filename.lower().endswith(image_extensions):
15             input_path = os.path.join(input_folder, filename)
16             output_path = os.path.join(output_folder, filename)
17
18             try:
19                 with Image.open(input_path) as img:
20                     img_resized = img.resize(target_size, Image.Resampling.LANCZOS)
21                     img_resized.save(output_path)
22                     print(f"Image redimensionnée : {output_path}")
23             except Exception as e:
24                 print(f"Erreur avec le fichier {filename} : {e}")
25
```

Figure 18: Scripte de Redimensionnement

3.2.2 Normalisation :

La normalisation d'une image est le processus qui consiste à transformer les valeurs de ses pixels dans un intervalle standardisé, généralement compris entre 0 et 1.

Cela permet de réduire les écarts d'échelle entre les données, de stabiliser l'entraînement des modèles de Deep Learning, et d'accélérer la convergence.

Dans ce projet, la normalisation des images est effectuée automatiquement lors de l'entraînement, à l'aide de la fonction `transforms.ToTensor()` de la bibliothèque PyTorch. Cette opération transforme les images en tenseurs et ramène leurs valeurs de pixels dans l'intervalle $[0, 1]$.

```
1  from torch.utils.data import Dataset
2  from PIL import Image
3  import os
4  from torchvision import transforms
5
6  class JPEGDataset(Dataset):
7      def __init__(self, jpeg_dir, original_dir, image_size=None):
8          self.jpeg_dir = jpeg_dir
9          self.original_dir = original_dir
10         self.filenamees = sorted(os.listdir(jpeg_dir))
11
12         if image_size is not None:
13             self.transform = transforms.Compose([
14                 transforms.Resize(image_size),
15                 transforms.ToTensor()
16             ])
17         else:
18             self.transform = transforms.ToTensor()
19
20     def __getitem__(self, idx):
21         fname = self.filenamees[idx]
22         jpeg = Image.open(os.path.join(self.jpeg_dir, fname)).convert("L")
23         orig = Image.open(os.path.join(self.original_dir, fname)).convert("L")
24         jpeg = self.transform(jpeg)
25         orig = self.transform(orig)
26         return jpeg, orig
27
28     def __len__(self):
29         return len(self.filenamees)
30
```

Figure 19: Scripte de la normalisation

3.2.3 Génération des paires d'entraînement :

Pour les besoins du projet, seule la base de données MS COCO a été utilisée. À partir de chaque image originale, une seule version compressée a été générée au format JPEG, en appliquant un facteur de qualité (FQ) différent pour chaque image. Les valeurs de FQ utilisées varient entre 5 et 50, selon une répartition aléatoire ou prédéfinie, afin de couvrir différents niveaux de compression.

Chaque image compressée conserve le même contenu visuel que l'original, mais avec une qualité altérée selon le FQ qui lui a été attribué. Le nommage des fichiers a été effectué de manière systématique pour faciliter le traitement automatisé.

Cette stratégie permet de constituer un ensemble diversifié de paires (image compressée, image originale), dans lequel chaque image compressée présente un niveau de dégradation spécifique lié au facteur de qualité appliqué. Une telle variété de cas est essentielle pour entraîner un modèle de restauration d'images capable de s'adapter efficacement à différents niveaux de

compression JPEG, et ainsi améliorer la qualité des images dégradées de manière robuste et cohérente.

3.2.4 Répartition des jeux de données :

L'ensemble des données a été réparti en deux parties sous-ensembles :

- Jeu d'entraînement (train) : utilisé pour l'apprentissage du modèle.
- Jeu de test (test) : utilisé pour l'évaluation finale.

Voici comment les organiser dans le dossier de projet ARCNN :

```
ARCNN/  
└─ data/  
    └─ train/  
        │   └─ images_originaux/      # Images non compressées pour l'entraînement  
        │   └─ images_compressées/    # Versions JPEG compressées des images d'entraînement  
        │  
        └─ test/  
            │   └─ images_originaux/    # Images non compressées pour le test  
            │   └─ images_compressées/  # Versions JPEG compressées des images de test
```

Figure 20: Organisation de l'ensemble des données dans le dossier ARCNN

4. Entraînement du modèle :

4.1 Environnement de développement (local) :

4.1.1 Langage de programmation (PYTHON) :

Le langage Python est un langage de programmation open source multi-plateformes et orienté objet. Grâce à des bibliothèques spécialisées, Python s'utilise pour de nombreuses situations comme le développement logiciel, l'analyse de données, ou la gestion d'infrastructures. Il n'est donc pas, comme le langage HTML par exemple, uniquement dédié à la programmation web.

Langage de programmation interprété, Python permet l'exécution du code sur n'importe quel ordinateur. Utilisable aussi bien par des programmeurs débutants qu'experts, Python permet de créer des programmes de manière simple et rapide.

Exemples d'utilisation :

- Développement web (avec Django, Flask).
- Science des données (Pandas, NumPy, Matplotlib).
- Intelligence artificielle et apprentissage automatique (TensorFlow, PyTorch).
- Automatisation et scripts.
- Applications desktop.



Figure 21: Langage de programmation PYTHON

4.1.2 Bibliothèques :

a) PyTorch :

PyTorch est une bibliothèque logicielle Python open source d'apprentissage automatique qui s'appuie sur Torch (en) développée par Meta. PyTorch est gouverné par la PyTorch Foundation. PyTorch permet d'effectuer les calculs tensoriels nécessaires notamment pour l'apprentissage profond (*deep learning*). Ces calculs sont optimisés et effectués soit par le processeur (CPU) soit, lorsque c'est possible, par un processeur graphique (GPU) supportant CUDA. PyTorch a été créé par les équipes de recherche de Facebook, précédées par les travaux de recherche de Ronan Collobert au cœur de l'équipe de Samy Bengio à l'IDIAP [25].

PyTorch permet, entre autres, de :

- Manipuler des tenseurs (tableaux multidimensionnels), de les échanger facilement avec Numpy et d'effectuer des calculs efficaces sur CPU ou GPU (par exemple, des produits de matrices ou des convolutions) .
- Calculer des gradients pour appliquer facilement des algorithmes d'optimisation par descente de gradient. PyTorch utilise la bibliothèque autograd .



Figure 22: La bibliothèque PyTorch

b) Torchvision :

Torchvision est une bibliothèque Python incontournable dans l'écosystème PyTorch, spécialement conçue pour simplifier et accélérer les projets de vision par ordinateur. Elle se distingue en fournissant des jeux de données populaires prêts à l'emploi (comme ImageNet ou

COCO), une collection riche de modèles de réseaux neuronaux profonds pré-entraînés pour des tâches variées (classification, détection d'objets, segmentation), et un ensemble complet de transformations d'images essentielles au prétraitement des données [26].

En somme, Torchvision est un pilier pour quiconque travaille avec des images ou des vidéos en PyTorch, offrant des outils standardisés qui permettent aux développeurs et chercheurs de se concentrer sur l'innovation plutôt que sur la gestion des infrastructures de base.



Figure 23: La bibliothèque torchvision

c) Pillow :

Pillow est une bibliothèque Python puissante et facile à utiliser pour le traitement d'images. Issue de la bibliothèque originale PIL (Python Imaging Library), elle permet de charger, modifier, convertir, analyser et enregistrer des images dans de nombreux formats tels que JPEG, PNG, BMP ou TIFF. Avec Pillow, on peut facilement effectuer des opérations comme le redimensionnement, la rotation, le recadrage, l'application de filtres (flou, netteté), l'ajout de texte ou la conversion de couleurs (niveaux de gris, YCbCr, etc.). Elle est largement utilisée dans des domaines comme la vision par ordinateur, le traitement d'images automatisé ou le développement d'interfaces graphiques [27].



Figure 24: La bibliothèque pillow

4.1.3 Editeur de code :

- **Visual Studio Code :**

Visual Studio Code (VS Code) est un éditeur de code source léger, open source et multiplateforme (Windows, macOS, Linux) développé par Microsoft. Il combine simplicité et puissance en offrant des fonctionnalités avancées telles que la coloration syntaxique, l'autocomplétion intelligente (IntelliSense), le débogage intégré, la gestion de versions via Git, et la personnalisation via des extensions. Basé sur Electron, il supporte une multitude de langages de programmation et frameworks grâce à son écosystème d'extensions riches. Son interface intuitive, son terminal intégré, sa prise en charge des snippets et sa capacité à exécuter des tâches automatisées en font un outil privilégié des développeurs pour la productivité et la collaboration, tout en restant performant et modulable selon les besoins.

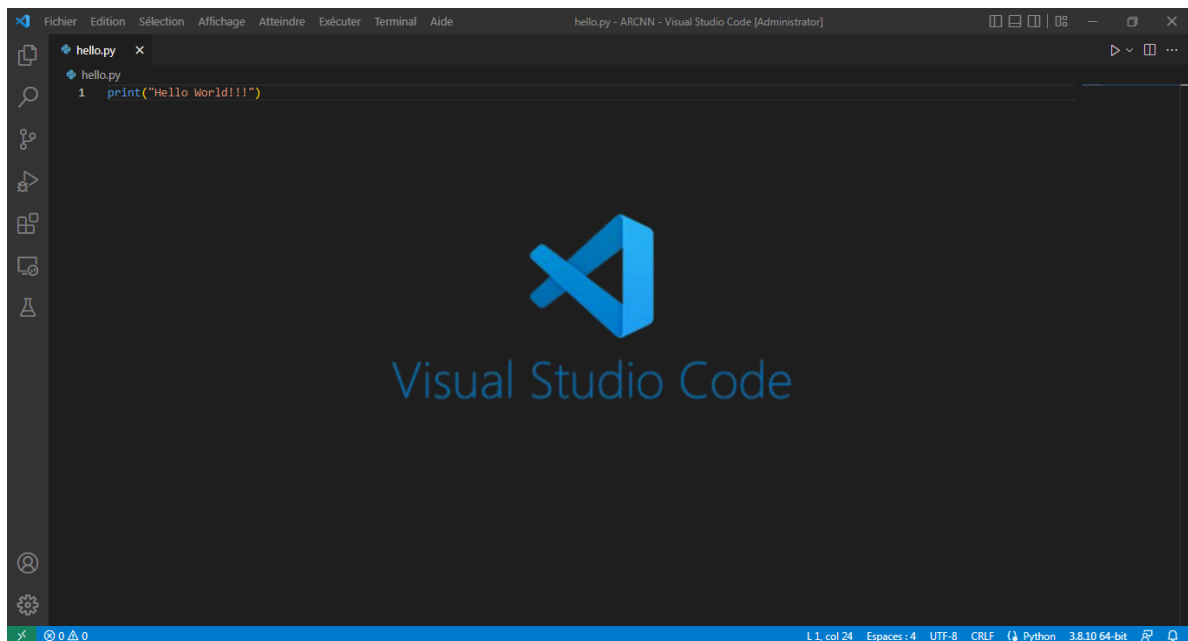


Figure 25: L'éditeur Visual Studio Code

4.2 Environnement de développement (en ligne) :

On peut également faire du développement en ligne avec l'environnement Google Colab sans avoir à installer de bibliothèques, car elles sont déjà intégrées:

Google Colab (ou Colaboratory) est un service cloud gratuit proposé par Google, basé sur Jupyter Notebook, qui permet d'écrire, d'exécuter et de partager du code Python directement dans un navigateur web. Conçu principalement pour le machine learning et l'analyse de données, Colab offre un environnement préconfiguré avec des bibliothèques populaires comme TensorFlow, PyTorch et scikit-learn. Il inclut également un accès à des GPU et TPU gratuits pour accélérer les calculs intensifs. Les notebooks sont sauvegardés automatiquement sur Google Drive, facilitant la collaboration en temps réel. Idéal pour la recherche, l'éducation et le prototypage rapide, Colab élimine les contraintes de configuration locale en fournissant une plateforme accessible et puissante dans le cloud [28].



Figure 26: Le service Google Colab

4.3 Stratégie d'entraînement :

L'objectif de l'entraînement est de permettre au modèle ARCNN d'apprendre à restaurer des images JPEG dégradées par compression, en minimisant les différences entre l'image compressée et l'image originale de référence. Pour cela, une stratégie d'entraînement bien définie est adoptée, combinant plusieurs éléments : choix des fonctions de perte, méthode d'optimisation, gestion des données et sauvegarde des poids du modèle.

4.3.1 Télécharger le fichier `arcnn.py` :

Il est nécessaire de télécharger le fichier `arcnn.py`, qui contient l'architecture complète du réseau en langage Python, sous PyTorch. Ce fichier sert de base pour entraîner un modèle personnalisé sur notre propre base de données d'images compressées. Nous avons intégré cette architecture dans notre pipeline d'entraînement en y associant notre gestion des données, nos fonctions de perte, notre plan d'apprentissage, ainsi que notre système de sauvegarde.

4.3.2 Fonctions de perte :

Deux fonctions de perte sont combinées afin de capturer à la fois l'exactitude numérique et la qualité visuelle des images restaurées :

a) MSE (Mean Squared Error) :

Cette perte mesure la différence pixel à pixel entre l'image générée et l'image originale. Elle est simple et efficace pour entraîner les réseaux à produire une sortie proche en intensité de l'image cible.

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{x}_i)^2 \quad \dots\dots\dots (5)$$

Avec :

N : le nombre total d'échantillons ou de pixels considérés (taille du vecteur ou de l'image).

x_i : la valeur réelle (valeur de référence ou *ground truth*) pour l'échantillon *i*.

$\hat{x}_i$: la valeur prédite ou reconstruite par le modèle pour le même échantillon *i*.

b) Perceptual Loss :

Cette perte compare les images non pas dans l'espace pixel, mais dans un espace de caractéristiques extraites par un réseau pré-entraîné (VGG16). Elle permet de préserver des détails visuels plus fins et d'éviter les résultats trop lissés [29].

c) Perte totale :

Les deux pertes sont combinées de la manière suivante :

$$\mathcal{L}_{total} = \mathcal{L}_{MSE} + \lambda \cdot \mathcal{L}_{perceptuelle} \quad \dots\dots\dots (6)$$

Avec $\lambda=0.05$, une valeur choisie empiriquement pour équilibrer les deux contributions.

4.3.3 Optimisation :

- a) **Optimiseur** : L'optimiseur Adam est utilisé pour sa capacité à s'adapter dynamiquement à chaque paramètre du modèle.
- b) **Taux d'apprentissage** : Un taux initial de $5e-5$ est choisi pour permettre une convergence stable.
- c) **Mini-batches** : L'entraînement est effectué par mini-batches de 8 images pour profiter d'une meilleure efficacité mémoire et stabilité de mise à jour des poids.

4.3.4 Plan d'entraînement :

a) Nombre d'époques :

Le modèle est entraîné pendant 100 époques, avec possibilité d'interrompre et de reprendre grâce aux checkpoints sauvegardés.

b) Gestion des données :

Les données d'entraînement sont organisées sous forme de paires (image compressée / image originale). À chaque époque, l'ordre des échantillons est aléatoirement réinitialisé (shuffle=True) afin d'éviter le surapprentissage lié à l'ordre des images.

c) Utilisation du GPU :

Lorsque disponible, l'entraînement s'effectue sur GPU grâce à CUDA, accélérant considérablement le traitement.

4.3.5 Sauvegarde du modèle :

À la fin de chaque époque, les poids du modèle sont sauvegardés sous deux formes :

a) Checkpoints complets (poids + état de l'optimiseur + numéro d'époque)

b) Poids seuls (nommés par époque) pour une utilisation ultérieure en inférence.

Chapitre 03 : Application et Résultats

Voici le scripte utilisé pour l'entraînement :

```
2 # train.py
3 # =====
4 import torch
5 import torch.nn as nn
6 import torch.optim as optim
7 from torch.utils.data import DataLoader
8 from models.arcnn import ARCNN
9 from utils.dataset import JPEGDataset
10 from utils.metrics import psnr
11 from perceptual_loss import PerceptualLoss
12 import os
13
14 # =====
15 # Configurations
16 # =====
17 jpeg_dir = "data/train/images_compressées"
18 orig_dir = "data/train/images_originaux"
19 checkpoint_path = "checkpoint.pth"
20 num_epochs = 100
21 batch_size = 8
22 learning_rate = 5e-5
23 lambda_percep = 0.05
24
25 # =====
26 # Device setup
27 # =====
28 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
29
30 # =====
31 # Model
32 # =====
33 model = ARCNN().to(device)
34
35 # =====
36 # Loss functions
37 # =====
38 criterion_mse = nn.MSELoss()
39 criterion_perceptual = PerceptualLoss().to(device)
40
41 # =====
42 # Optimizer
43 # =====
44 optimizer = optim.Adam(model.parameters(), lr=learning_rate)
45
46 # =====
47 # Dataset & DataLoader
48 # =====
49 dataset = JPEGDataset(jpeg_dir, orig_dir, image_size=None) # image_size=None si déjà redimensionnées
50 dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
52 # =====
53 # Load checkpoint if exists
54 # =====
55 start_epoch = 0
56 if os.path.exists(checkpoint_path):
57     print("> Chargement du checkpoint...")
58     checkpoint = torch.load(checkpoint_path, map_location=device)
59     model.load_state_dict(checkpoint['model_state_dict'])
60     optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
61     start_epoch = checkpoint['epoch']
62     print(f"> Reprise à l'époque {start_epoch}")
63 # =====
64 # Training loop
65 # =====
66 for epoch in range(start_epoch, num_epochs):
67     model.train()
68     total_loss = 0
69     for x, y in dataloader:
70         x, y = x.to(device), y.to(device)
71         output = model(x)
72
73         loss_mse = criterion_mse(output, y)
74         loss_percep = criterion_perceptual(output, y)
75         loss = loss_mse + lambda_percep * loss_percep
76         optimizer.zero_grad()
77         loss.backward()
78         optimizer.step()
79         total_loss += loss.item()
80
81     avg_loss = total_loss / len(dataloader)
82     print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss:.4f}")
83 # =====
84 # Save checkpoint
85 # =====
86 torch.save({
87     'epoch': epoch + 1,
88     'model_state_dict': model.state_dict(),
89     'optimizer_state_dict': optimizer.state_dict(),
90     'loss': avg_loss
91 }, checkpoint_path)
92
93 # Optionnel : sauvegarde du modèle seul (nommé selon l'époque)
94 torch.save(model.state_dict(), f"arcnn_epoch{epoch+1}.pth")
```

Figure 27: Scripte d'entraînement

4.4 Évaluation qualitative et quantitative (PSNR, SSIM) :

4.4.1 PSNR (Peak Signal-to-Noise Ratio):

Le PSNR (Peak Signal-to-Noise Ratio) est utilisé comme indicateur quantitatif principal pour évaluer l'efficacité de notre modèle ARCNN dans la restauration d'images JPEG compressées.

Cet indice permet d'évaluer la qualité de l'image restaurée en la comparant à l'image originale, non dégradée. Il s'agit d'une mesure objective qui repose sur l'analyse des différences d'intensité entre les deux images. Une valeur de PSNR plus élevée indique une plus grande fidélité de l'image restaurée par rapport à l'image d'origine.

Plus précisément, après avoir traité une image JPEG compressée à l'aide du réseau ARCNN, nous procédons à une comparaison entre l'image restaurée et l'image d'origine afin de calculer le PSNR. Cette démarche nous permet d'estimer de manière numérique le degré de réussite de la reconstruction réalisée par le modèle, en complément d'une analyse visuelle [30].

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right) \dots\dots\dots (7)$$

- **MAX** : valeur maximale d'un pixel (255 pour une image 8 bits).
- **MSE**: moyenne des carrés des différences entre pixels de l'image originale et de l'image restaurée.
- PSNR élevé = meilleure qualité (moins d'erreur entre les deux images).
- PSNR bas = forte dégradation.

Le tableau ci-dessus [31] donne les résultats de PSNR en dB :

PSNR (en dB)	Qualité
PSNR > 40 dB	Excellente
30 dB < PSNR < 40 dB	Bonne qualité
20 dB < PSNR < 30 dB	Moyenne
PSNR < 20 dB	Faible

Tableau 2 : Résultats de PSNR en dB et leurs qualités

Le scripte Utilisé pour calculer le PSNR :

```
1  import os
2  from PIL import Image
3  import numpy as np
4  from skimage.metrics import peak_signal_noise_ratio
5
6  # Dossiers
7  restored_dir = "data/eval/images_restaurées"
8  original_dir = "data/eval/original"
9
10 # Résultats
11 psnr_scores = []
12
13 # Préparer la liste de fichiers
14 filenames = sorted(os.listdir(original_dir))
15
16 for filename in filenames:
17     if not filename.lower().endswith((".jpg", ".jpeg")):
18         continue
19
20     orig_path = os.path.join(original_dir, filename)
21     rest_path = os.path.join(restored_dir, filename)
22
23     if not os.path.exists(rest_path):
24         print(f"✗ Image restaurée manquante : {filename}")
25         continue
26
27     orig_img = Image.open(orig_path).convert("RGB")
28     rest_img = Image.open(rest_path).convert("RGB")
29
30     if orig_img.size != rest_img.size:
31         rest_img = rest_img.resize(orig_img.size)
32
33     orig_np = np.array(orig_img).astype(np.float32) / 255.0
34     rest_np = np.array(rest_img).astype(np.float32) / 255.0
35
36     psnr = peak_signal_noise_ratio(orig_np, rest_np, data_range=1.0)
37     psnr_scores.append(psnr)
38
39     print(f"{filename} - PSNR: {psnr:.2f} dB")
40
41 print(f"\n✅ Moyenne PSNR : {np.mean(psnr_scores):.2f} dB")
42
```

Figure 28: Scripte pour calculer le PSNR

4.4.2 SSIM (Structural Similarity Index Measure) :

SSIM est une mesure perceptuelle de la qualité d'une image, que nous utilisons dans ce projet pour évaluer la fidélité visuelle de l'image restaurée par le modèle ARCNN par rapport à l'image originale.

Contrairement au PSNR qui mesure simplement les différences d'intensité pixel par pixel, le SSIM prend en compte des caractéristiques structurelles telles que :

- La luminance (éclairage).
- Le contraste.
- La structure locale de l'image.

$$\text{SSIM}(\mathbf{x}, \mathbf{y}) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \quad \dots\dots\dots (8)$$

Avec :

- μ_x, μ_y = moyennes des intensités des images x et y.
- σ_x^2, σ_y^2 = variances des images.
- σ_{xy} = covariance entre les deux images.
- C_1, C_2 = constantes pour éviter une division par zéro.

Cela rend le SSIM Structural Similarity Index Measure plus proche de la perception humaine, et donc mieux adapté à juger la qualité visuelle réelle de la restauration [32].

Le tableau ci-dessus [33] donne les résultats de SSIM entre 0 et 1 :

SSIM	Qualité
SSIM ≈ 1	Excellente qualité
SSIM ≥ 0.9	Très bonne qualité
$0.8 < \mathbf{SSIM} < 0.9$	Bonne qualité, mais de légères pertes visibles
$0.6 < \mathbf{SSIM} < 0.8$	Qualité moyenne, des détails peuvent être altérés
SSIM < 0.6	Mauvaise qualité, structure dégradée

Tableau 3 : Résultats de SSIM et leurs qualités

Le scripte utilisé pour calculer le SSIM :

```
1  import os
2  from PIL import Image
3  import numpy as np
4  from skimage.metrics import structural_similarity
5
6  # Dossiers
7  restored_dir = "data/eval/images_restaurées"
8  original_dir = "data/eval/original"
9
10 # Résultats
11 ssim_scores = []
12
13 # Préparer la liste de fichiers
14 filenames = sorted(os.listdir(original_dir))
15
16 for filename in filenames:
17     if not filename.lower().endswith((".jpg", ".jpeg")):
18         continue
19
20     orig_path = os.path.join(original_dir, filename)
21     rest_path = os.path.join(restored_dir, filename)
22
23     if not os.path.exists(rest_path):
24         print(f"✗ Image restaurée manquante : {filename}")
25         continue
26
27     orig_img = Image.open(orig_path).convert("RGB")
28     rest_img = Image.open(rest_path).convert("RGB")
29
30     if orig_img.size != rest_img.size:
31         rest_img = rest_img.resize(orig_img.size)
32
33     orig_np = np.array(orig_img).astype(np.float32) / 255.0
34     rest_np = np.array(rest_img).astype(np.float32) / 255.0
35
36     ssim = structural_similarity(orig_np, rest_np, channel_axis=-1, data_range=1.0)
37     ssim_scores.append(ssim)
38
39     print(f"{filename} - SSIM: {ssim:.4f}")
40
41 print(f"\n✓ Moyenne SSIM : {np.mean(ssim_scores):.4f}")
```

Figure 29: Scripte pour calculer le SSIM

- **Structure finale du projet :**

Le projet est structuré autour d'un dossier principal nommé ARCNN, dans lequel sont organisés les scripts, les modèles, les utilitaires et les données. Cette organisation permet une séparation claire entre l'entraînement, le test, le modèle et les données, facilitant le développement, la reproductibilité et le déploiement du système.



Figure 30: Structure finale du projet développée

5. Implémentation sur Raspberry Pi :

5.1 Matériel :

5.1.1 Raspberry Pi 4 Model B :

a) Architecture :

Architecture ARM 64 bits, basée sur le SoC Broadcom BCM2711. Elle permet l'exécution de systèmes d'exploitation modernes (Linux 64 bits) et de bibliothèques compatibles avec des applications avancées telles que le traitement d'image et le machine learning [34].

b) Processeur (CPU) :

Quadricœur ARM Cortex-A72 (ARM v8) cadencé à **1.5 GHz**. Ce processeur offre une nette amélioration des performances par rapport aux générations précédentes, avec un traitement plus rapide des opérations mathématiques et des calculs lourds.

c) Mémoire vive (RAM) (8 Go de RAM LPDDR4-3200) :

Cette capacité permet d'exécuter des tâches complexes, d'entraîner ou d'évaluer des modèles de deep learning (comme ARCNN) et de travailler avec des images haute résolution sans risque de saturation mémoire.

d) Stockage :

Stockage principal via **carte** microSD (support du boot via USB aussi disponible). Il est recommandé d'utiliser une carte microSD UHS-I rapide, ou un SSD externe via USB 3.0 pour de meilleures performances.

e) Connectivité réseau :

- Ethernet Gigabit (1 Gbps réel)
- Wi-Fi 802.11 b/g/n/ac double bande (2,4 GHz et 5 GHz)
- Bluetooth 5.0, pour la communication avec d'autres appareils ou capteurs sans fil.

f) Ports USB :

4 ports USB
:

- 2 × USB 3.0 (transferts rapides pour disques durs, SSD, caméras)
- 2 × USB 2.0

g) Sorties vidéo/audio :

- 2 ports micro-HDMI, prenant en charge :
 - 1 écran 4K à 60 fps.
 - 2 écrans 4K à 30 fps.
- Sortie audio intégrée via HDMI ou jack 3,5 mm stéréo.

h) Graphique (GPU) :

VideoCore VI, compatible OpenGL ES 3.0. Ce GPU prend en charge l'accélération matérielle pour le décodage de vidéos 4K (H.265), utile pour la visualisation de résultats restaurés.

i) Alimentation :

Port USB-C (5V / 3A). Une alimentation officielle est recommandée pour garantir la stabilité lors de l'utilisation de périphériques puissants.

j) GPIO :

40 broches GPIO, compatibles avec les versions précédentes. Elles permettent de connecter des capteurs, caméras, écrans LCD, etc., utiles pour l'intégration dans des systèmes embarqués.

k) Dimensions & poids :

- **Taille :** 85,6 mm × 56,5 mm × 17 mm
- **Poids :** ~46 g
Son format compact est parfaitement adapté pour une intégration dans des boîtiers ou des projets embarqués.

l) Systèmes d'exploitation compatibles :

- Raspberry Pi 4 OS (32 ou 64 bits)
- Ubuntu, Debian, Manjaro ARM, etc .
- Compatible avec les environnements Python, PyTorch, TensorFlow.

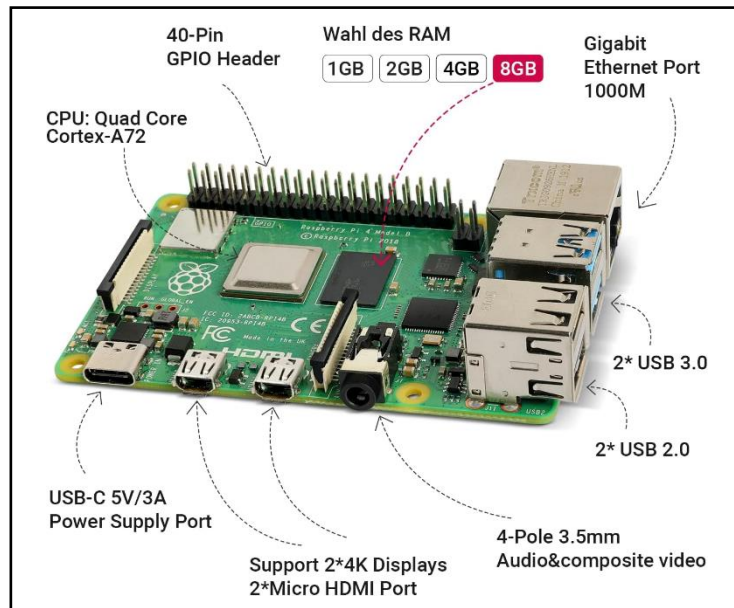


Figure 31: Raspberry Pi 4 Model B

5.1.2 Carte microSD :

La carte microSD (micro Secure Digital) est un support de stockage amovible de très petite taille, conçu principalement pour les appareils électroniques compacts comme les smartphones, les caméras, les drones, ou encore les Raspberry Pi.

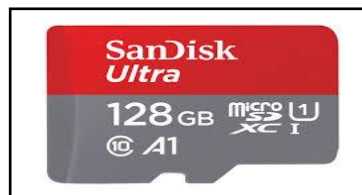


Figure 32: Carte microSD

5.1.3 Lecteur de carte SD :

C'est un petit accessoire que l'on connecte via un port USB à un ordinateur. Il permet de lire et écrire sur des cartes mémoire SD ou microSD, souvent utilisées dans les appareils photo, téléphones ou Raspberry Pi 4. Très pratique, il facilite le transfert de fichiers ou l'installation de systèmes sur une carte microSD, en particulier lorsqu'un ordinateur ne possède pas de lecteur intégré.



Figure 33: Lecture de la carte SD

5.2 Préparation et installation du système d'exploitation pour la Raspberry Pi 4 :

a) Téléchargement de Raspberry Pi Imager :

- Aller sur le site officiel : <https://www.raspberrypi.com/software/> [35].
- Télécharger Raspberry Pi Imager selon votre système (Windows, macOS, Linux).
- Installer le logiciel.

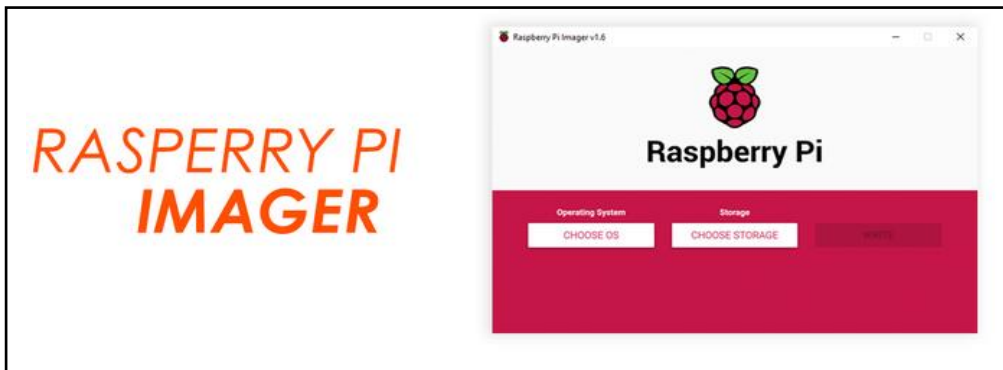


Figure 34: Raspberry Pi Imager

b) Préparation de la carte microSD :

1. Insérer la carte microSD dans le lecteur connecté à l'ordinateur.
2. Ouvrir Raspberry Pi 4 Imager.

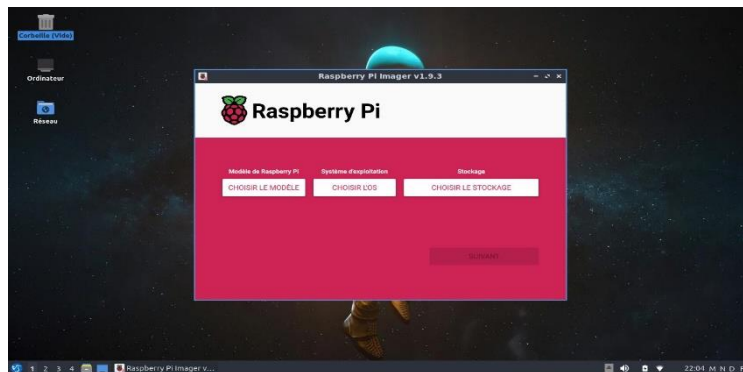


Figure 35: Ouverture de Raspberry Pi Imager

3. Choisir le **model** le carte Raspberry :

- Par exemple : Raspberry Pi 4

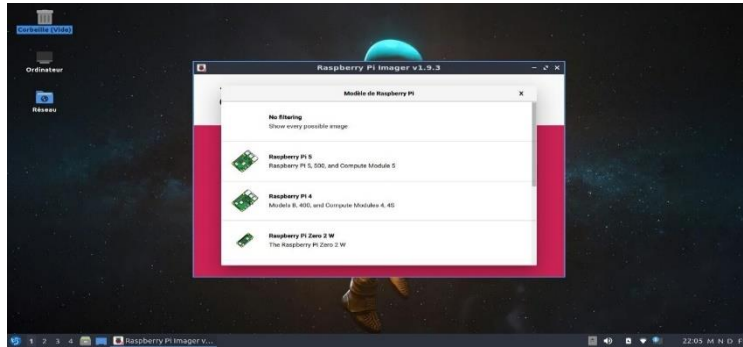


Figure 36: Le choix de model de la carte Raspberry

4. Choisir un **système** :

- Par exemple : Raspberry Pi 4 OS (64-bit)

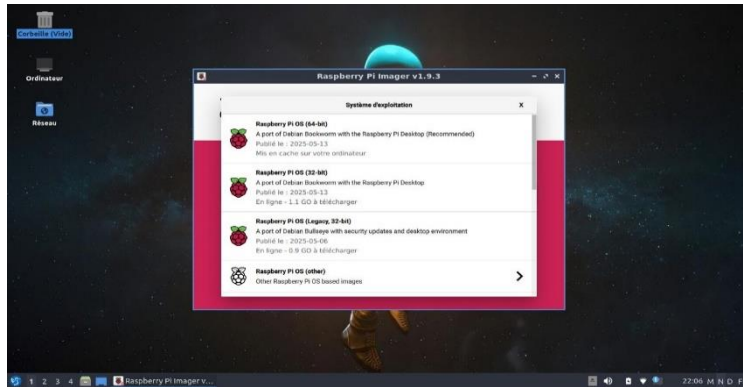


Figure 37: Le choix de système

5. Sélectionner la carte SD détectée :

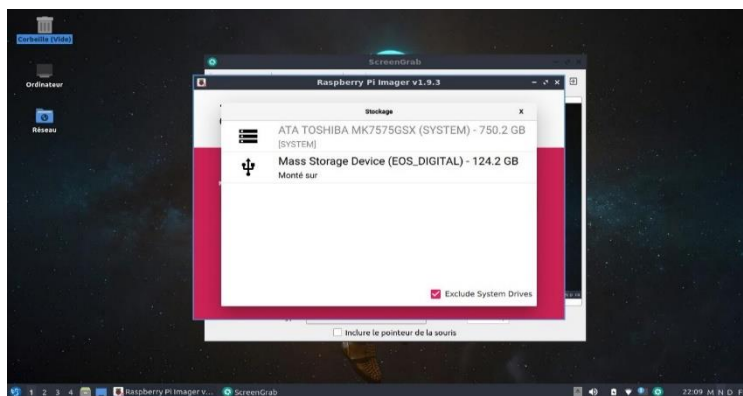


Figure 38: La sélection de la carte SD détectée

6. Cliquer sur “suivant ” :

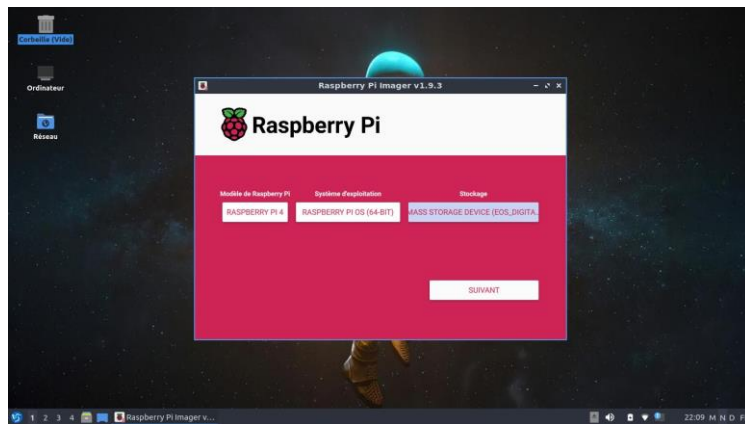


Figure 39: Cliquer “suivant ”

7. Cliquer sur “ Non ” :

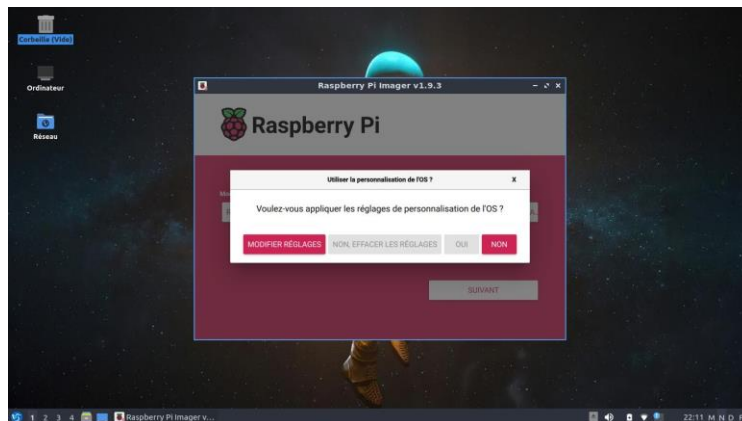


Figure 40: Cliquer “ Non ”

Puis “ Oui ” :

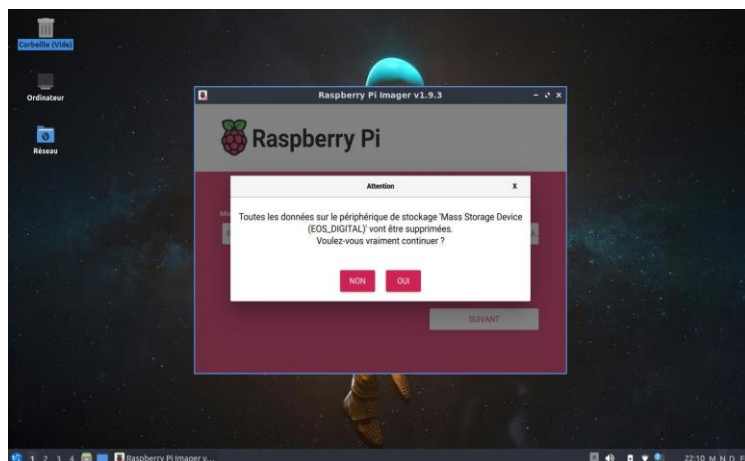


Figure 41: Cliquer “ oui ”

8. Attendre la fin de l'installation et retirer la carte en toute sécurité :

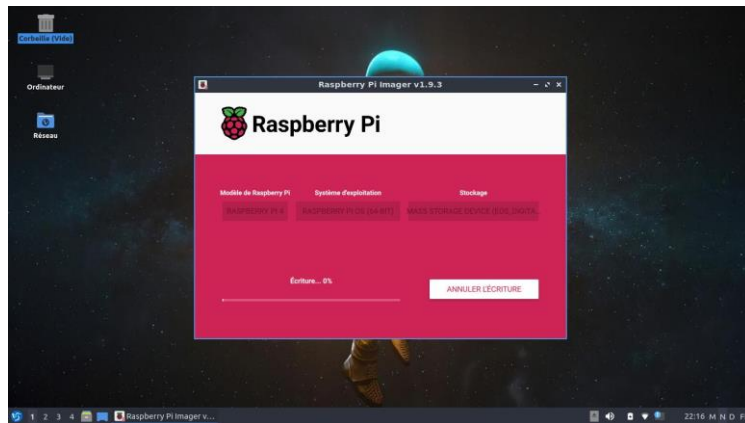


Figure 42: Installation de la carte microSD

c) Configuration initiale de la Raspberry Pi 4 :

1. Insérer la carte microSD dans la Raspberry Pi 4.
2. Brancher :
 - L'écran via le port micro-HDMI.
 - Le clavier et la souris.
 - Enfin, **l'alimentation**.
3. La Raspberry Pi 4 démarre automatiquement et charge le système.
4. Suivre les étapes de configuration initiale :
 - Choix de la langue, clavier, Wi-Fi, mise à jour, mot de passe utilisateur, etc.

5.3 Installation des bibliothèques nécessaires :

Il faut savoir qu'avant d'installer les bibliothèques nécessaires on doit d'abord installer Python et voici comment faire :

1. Ouvrez le terminal de la raspberry.
2. Exécutez cette commande “ `sudo apt install python3` ” dans le terminal et attendez la fin de l'installation.

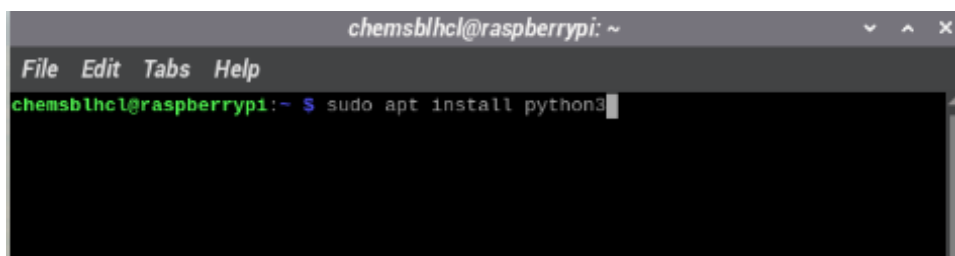
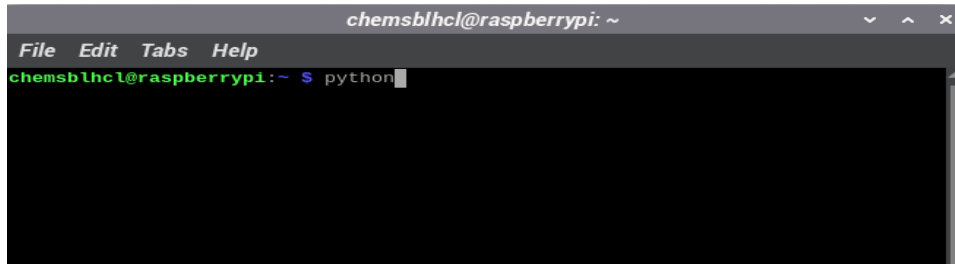


Figure 43: Exécution de la commande “ `sudo apt install python3` ”

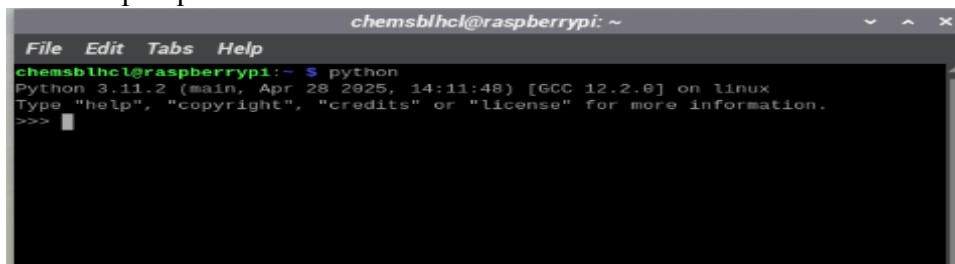
3. Vérifiez l'installation de Python avec la commande “ `python` ”:



```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
chemsblhcl@raspberrypi:~$ python
```

Figure 44: Vérification de l'installation de PYTHON

Vous verrez quelque chose comme :



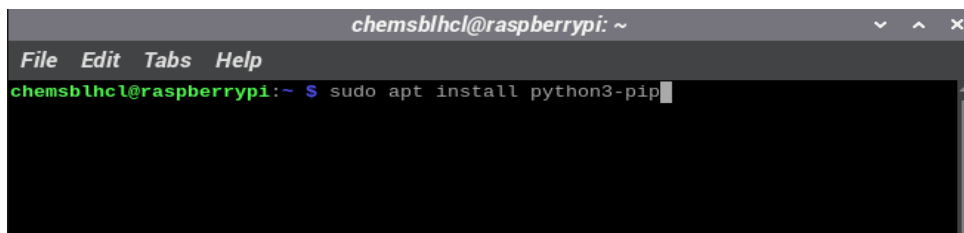
```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
chemsblhcl@raspberrypi:~$ python  
Python 3.11.2 (main, Apr 28 2025, 14:11:48) [GCC 12.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

Figure 46: PYTHON est installé

Après avoir installé Python, on doit installer le Pip (Pip signifie “ Pip installs Packages ”, est considéré aussi comment une commande qui permet d'installer, mettre à jour ou désinstaller des bibliothèques Python.

Et pour faire, suivez ces étapes :

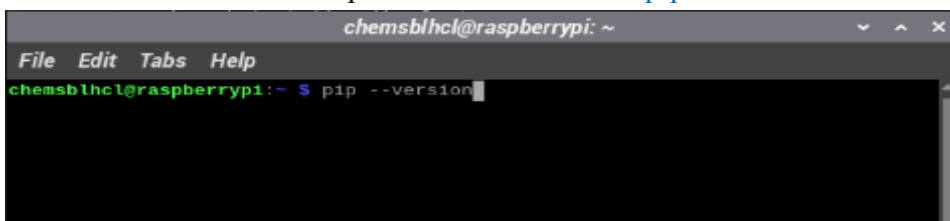
1. Ouvrez le terminal de la raspberry.
2. Exécutez cette commande “ `sudo apt install python3-pip` ” dans le terminal et attendez la fin de l'installation.



```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
chemsblhcl@raspberrypi:~$ sudo apt install python3-pip
```

Figure 46: Exécutez de la commande “ `sudo apt install python3-pip` ”

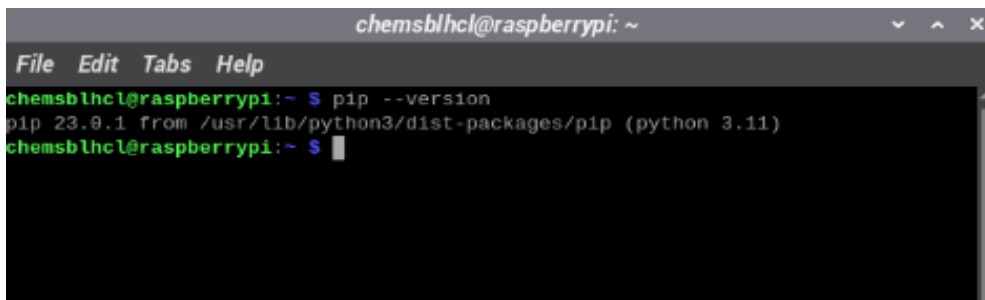
3. Vérifiez l'installation de Pip avec la commande “ `pip3 --version` ” :



```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
chemsblhcl@raspberrypi:~$ pip --version
```

Figure 47: Vérification de l'installation de Pip

Vous verrez quelque chose comme :



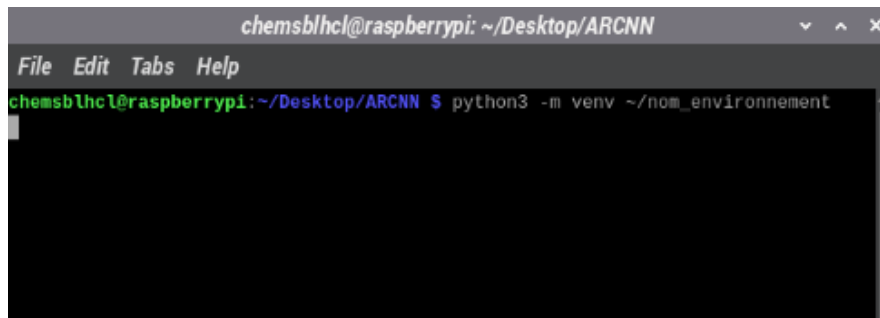
```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
chemsblhcl@raspberrypi:~ $ pip --version  
pip 23.0.1 from /usr/lib/python3/dist-packages/pip (python 3.11)  
chemsblhcl@raspberrypi:~ $
```

Figure 48: Pip est installé

A fin de l'installation de Python et le Pip, on doit créer un environnement virtuel pour qu'on pourra installer les bibliothèques nécessaires.

- **Création de de l'environnement virtuel :**

1. Ouvrez le terminal de la raspberry.
2. Exécutez cette commande “ `python3 -m venv ~/nom_environnement` ” dans le terminal et attendez la fin de la création.



```
chemsblhcl@raspberrypi: ~/Desktop/ARCNN  
File Edit Tabs Help  
chemsblhcl@raspberrypi:~/Desktop/ARCNN $ python3 -m venv ~/nom_environnement
```

Figure 49: Exécutez de la commande “ `python3 -m venv ~/nom_environnement` ”

3. Vous verrez que l'environnement a été créée sous le nom (nom_environnement):

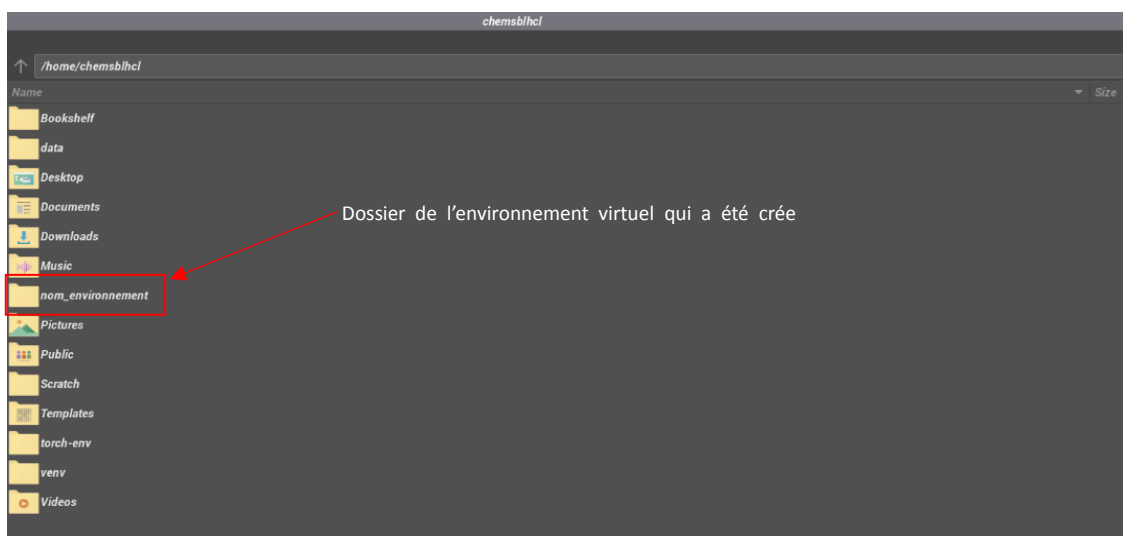
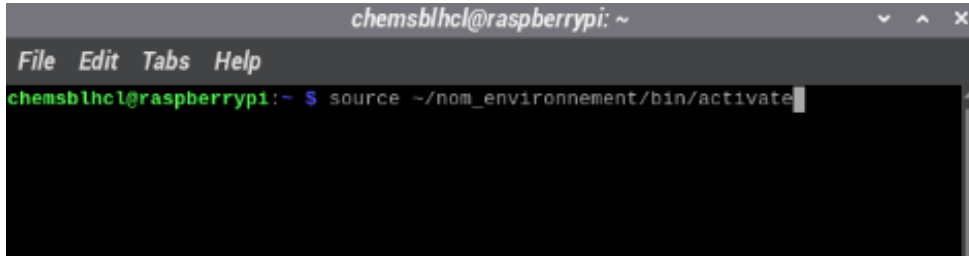


Figure 50: Dossier de l'environnement virtuel

- **Activation de l'environnement virtuel :**

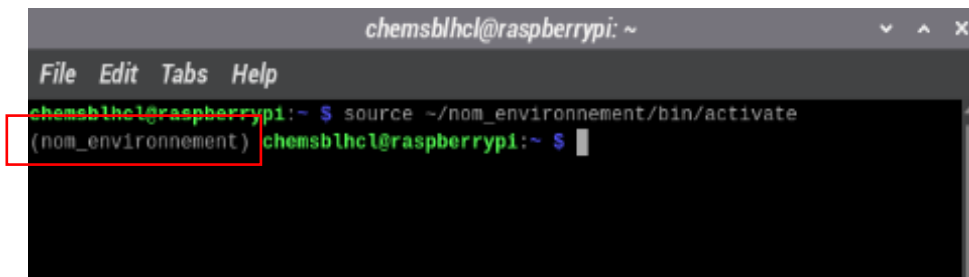
1. Ouvrez le terminal de la Raspberry.
2. Exécutez cette commande “ `source ~/nom_environnement/bin/activate` ” dans le terminal pour l'activation.



```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
chemsblhcl@raspberrypi:~ $ source ~/nom_environnement/bin/activate
```

Figure 51: Exécutez de la commande “ `source ~/nom_environnement/bin/activate` ”

Vous verrez l'apparition du nom de votre environnement avant le de l'utilisateur ce qui signifie que l'environnement est bien activé.



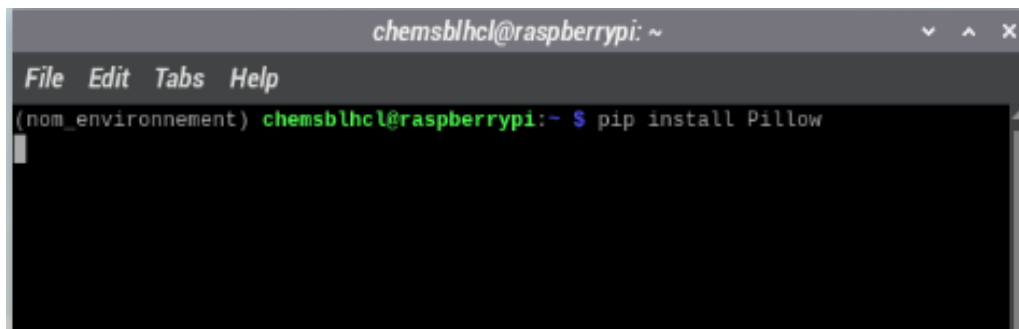
```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
(nom_environnement) chemsblhcl@raspberrypi:~ $
```

Figure 52: Environnement virtuel est activé

Après l'installation de Python, Pip et la création et l'activation de l'environnement virtuel maintenant on peut installer les bibliothèques nécessaires :

1. Bibliothèque Pillow :

Pour installer cette bibliothèque il suffit d'exécuter cette commande “ `pip install Pillow` ” et attendez la fin de l'installation.

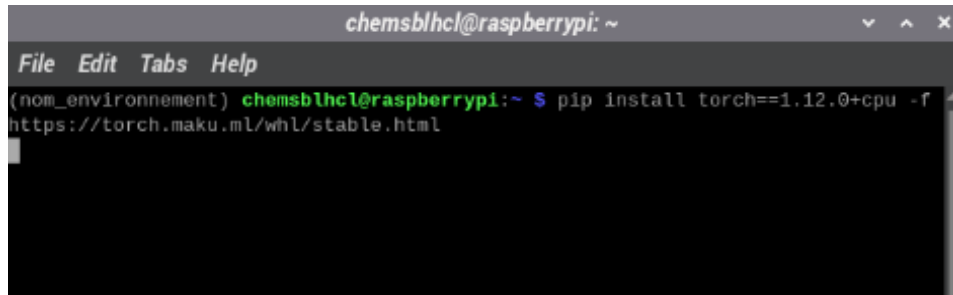


```
(nom_environnement) chemsblhcl@raspberrypi:~ $ pip install Pillow
```

Figure 53: Installation de la bibliothèque Pillow

2. Bibliothèque Pytorch :

Pour installer cette bibliothèque il suffit d'exécuter cette commande “`pip install torch==1.12.0+cpu -f http://torch.maku.ml/whl/stable.html”` et attendez la fin de l'installation.

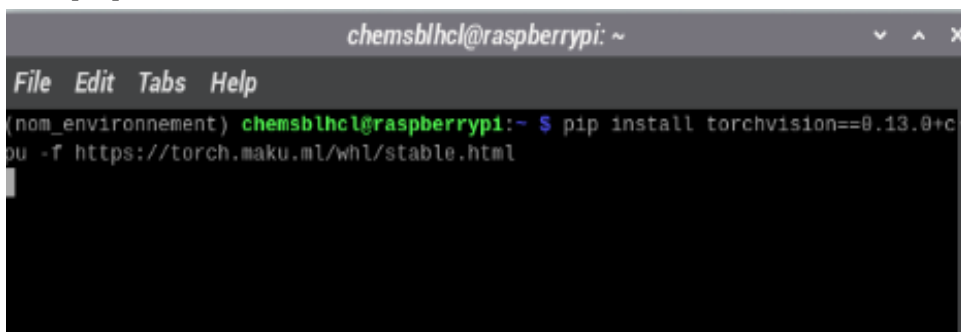


```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
(nom_environnement) chemsblhcl@raspberrypi:~ $ pip install torch==1.12.0+cpu -f  
https://torch.maku.ml/whl/stable.html
```

Figure 54: Installation de la bibliothèque Pytorch

3. Bibliothèque Torchvision :

Pour installer cette bibliothèque il suffit d'exécuter cette commande “`pip install torchvision==0.13.0+cpu -f http://torch.maku.ml/whl/stable.html”`et attendez la fin de l'installation [36].



```
chemsblhcl@raspberrypi: ~  
File Edit Tabs Help  
(nom_environnement) chemsblhcl@raspberrypi:~ $ pip install torchvision==0.13.0+cpu  
-f https://torch.maku.ml/whl/stable.html
```

Figure 55: Installation de la bibliothèque Torchvision

Note :

L'installation de ces bibliothèques diffère entre la Raspberry Pi 4 et l'ordinateur.

6. Test et résultats :

Pour évaluer concrètement les performances de notre modèle, nous avons implémenté le programme sur une carte Raspberry Pi 4 Model B, qui représente une plateforme matérielle limitée en ressources mais très utilisée dans les applications embarquées. L'objectif était de vérifier si notre approche pouvait fonctionner en conditions réelles, en dehors d'un environnement de calcul puissant.

Le travail a consisté à préparer la base de données d'images compressées (issues du set MS COCO train2014, comprenant 82 783 images), puis à effectuer un prétraitement (redimensionnement en 256×256 pixels) afin d'uniformiser les entrées du réseau. Le modèle ARCNN, déjà entraîné, a été déployé sur la Raspberry Pi. Nous avons installé les bibliothèques

nécessaires (Python, PyTorch/DirectML, OpenCV, etc.) pour assurer la compatibilité avec le matériel ARM.

Une fois le programme exécuté sur la Raspberry Pi, nous avons appliqué le réseau sur différentes images JPEG dégradées. Le système a alors produit automatiquement des images restaurées. Ces résultats ont été ensuite comparés aux images originales de référence afin d'évaluer la qualité de restauration.

Cette phase de test pratique est importante car elle montre non seulement la capacité du modèle à réduire les artefacts JPEG, mais aussi sa faisabilité sur une plateforme embarquée à faible consommation énergétique comme la Raspberry Pi, ce qui ouvre la voie à des applications dans le domaine de la vision embarquée et de l'IoT.

6.1 Résultats quantitatifs :

Pour évaluer les performances du modèle, trois images de test représentatives ont été sélectionnées. Elles illustrent différentes caractéristiques visuelles et seront utilisées pour analyser les résultats obtenus.

Voici les images à tester :



(a)



(b)



(c)

Figure 56: Images de tests

6.1.1 Évaluation avec PSNR :

Le PSNR exprime le rapport entre le signal utile et le bruit introduit par la compression et la restauration.

Un PSNR élevé indique une meilleure fidélité de l'image restaurée par rapport à l'image originale.

Image testée	Facteur de compression	PSNR l'image restaurée
A	10	29.93 dB
B	20	30.89 dB
C	40	32.91 dB

Tableau 4 : Résultats de PSNR

6.1.2 Évaluation avec SSIM :

Le SSIM mesure la similarité structurelle entre deux images.
Une valeur proche de 1 indique une forte ressemblance visuelle avec l'image de référence.

Image testée	Facteur de compression	SSIM l'image restaurée
A	10	0.8708
B	20	0.8496
C	40	0.8936

Tableau 5 : Résultats de SSIM

6.2 Résultats qualitatifs :

Pour compléter l'analyse, une comparaison visuelle est effectuée entre :

- **Image originale** (Bonne qualité).
- **Image JPEG compressée** (artefacts visibles).
- **Image restaurée par ARCNN**.

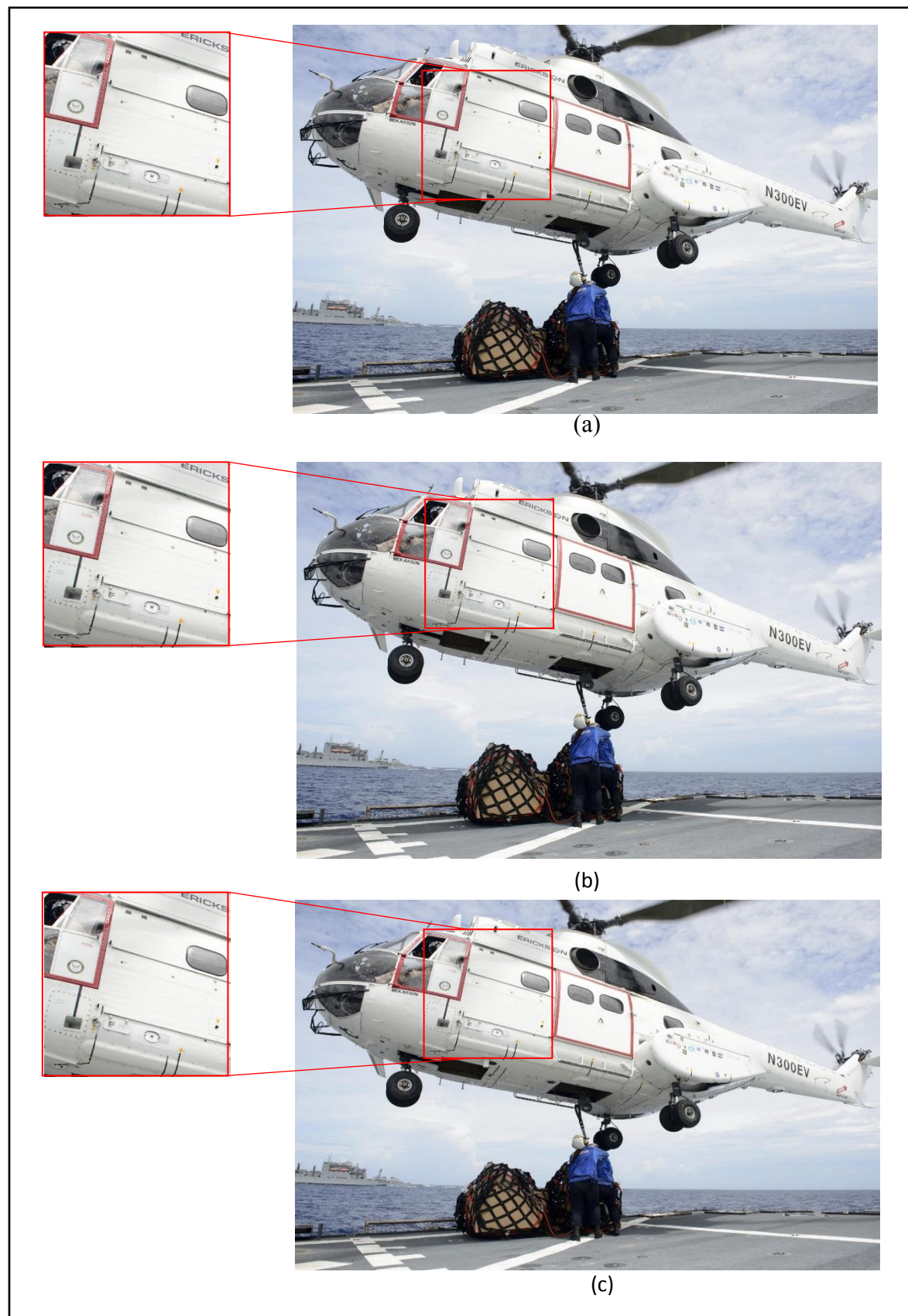


Figure 57 : Image (A) compressée $qf=10$, restaurée par ARCNN
(a) Image originale | (b) Image compressée | (c) Image restaurée



Figure 58 : Image (B) compressée $qf=10$, restaurée par ARCNN
(a) Image originale | (b) Image compressée | (c) Image restaurée

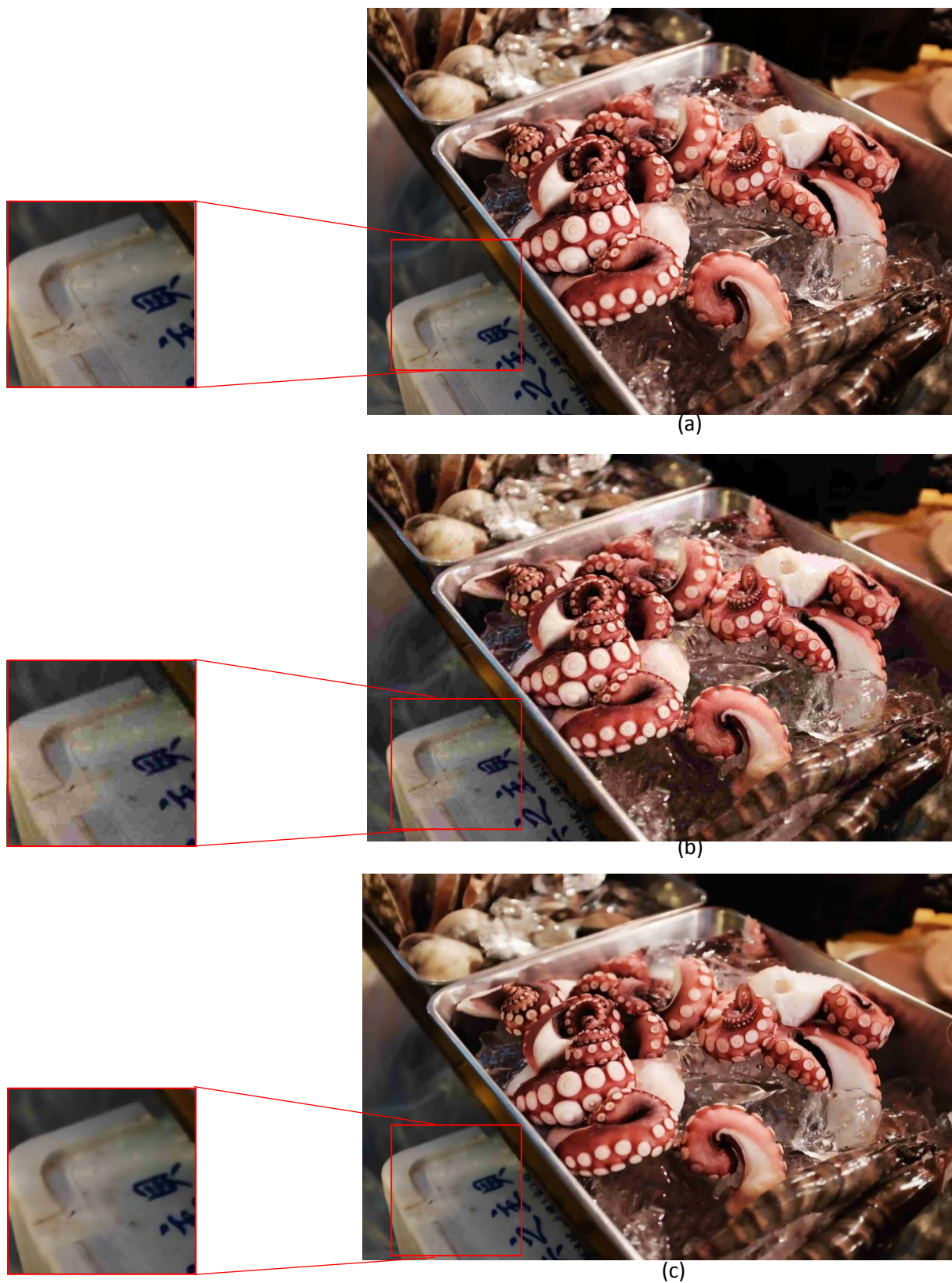


Figure 59 : Image (C) compressée qf= 10, restaurée par ARCNN
(a) Image originale | (b) Image compressée | (c) Image restaurée

6.3 Interprétation des résultats :

L'évaluation quantitative des performances du modèle ARCNN met en évidence une amélioration significative de la qualité des images compressées par rapport aux versions JPEG d'origine. Les résultats obtenus, mesurés à l'aide des indicateurs PSNR et SSIM, montrent une progression claire des valeurs, attestant de l'efficacité du modèle dans la restauration d'images dégradées.

Cette amélioration se traduit par une réduction notable des artefacts de compression, observée de manière systématique dans les différents cas de test. Plus précisément, ARCNN parvient à corriger efficacement :

- **Les effets de bloc (8×8) :** les discontinuités visibles entre les blocs de pixels, typiques de la compression JPEG, sont atténuées voire supprimées, ce qui rend l'image visuellement plus homogène.
- **Le flou induit par la perte de hautes fréquences :** le modèle permet de récupérer partiellement les détails fins et les textures, ce qui améliore la netteté des images, notamment dans les zones riches en contenu.
- **Les distorsions au niveau des contours :** les bordures d'objets et les transitions de couleur ou de luminosité sont mieux restituées, ce qui contribue à une perception plus naturelle de l'image.

L'amélioration des scores PSNR et SSIM observée dans l'ensemble des tests confirme que le modèle ne se contente pas d'un simple lissage, mais qu'il est capable de reconstruire fidèlement les informations perdues tout en respectant la structure globale de l'image.

7. Fonctionnement des deux applications développées

7.1. Application temps différé :

L'application hors temps réel développée repose sur le modèle ARCNN (Artifacts Reduction Convolutional Neural Network). Elle fonctionne à partir d'images non compressées stockées sur le système, que l'utilisateur sélectionne pour ensuite simuler une compression JPEG et évaluer les performances de restauration.

- Ouverture de l'application :

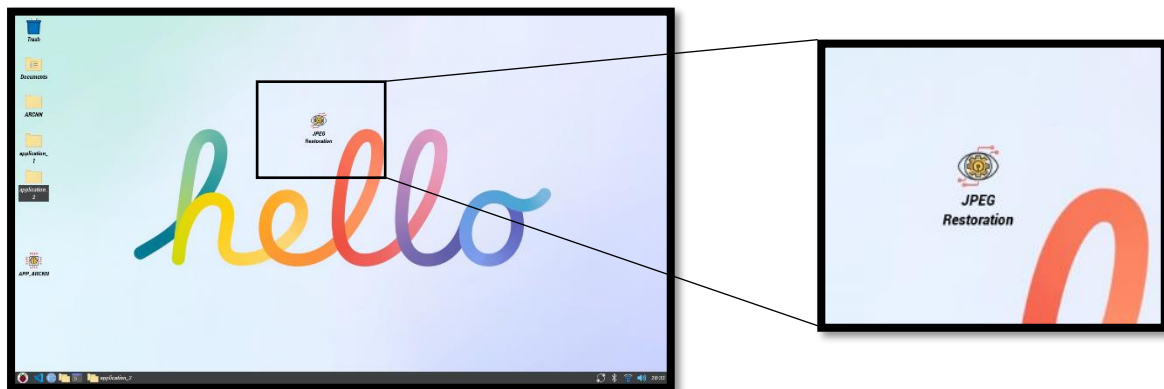


Figure 60 : Ouverture de l'application 1.

- Chargement de l'image :

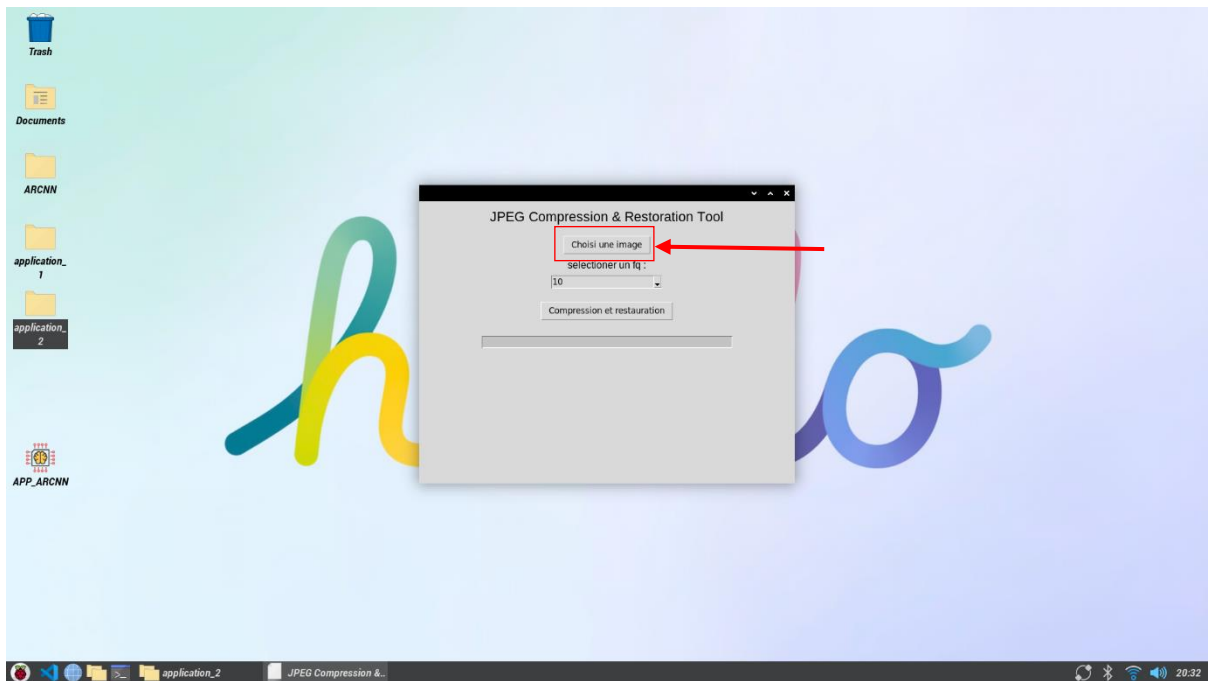


Figure 61 : Cliquer sur “Choisi une image”.

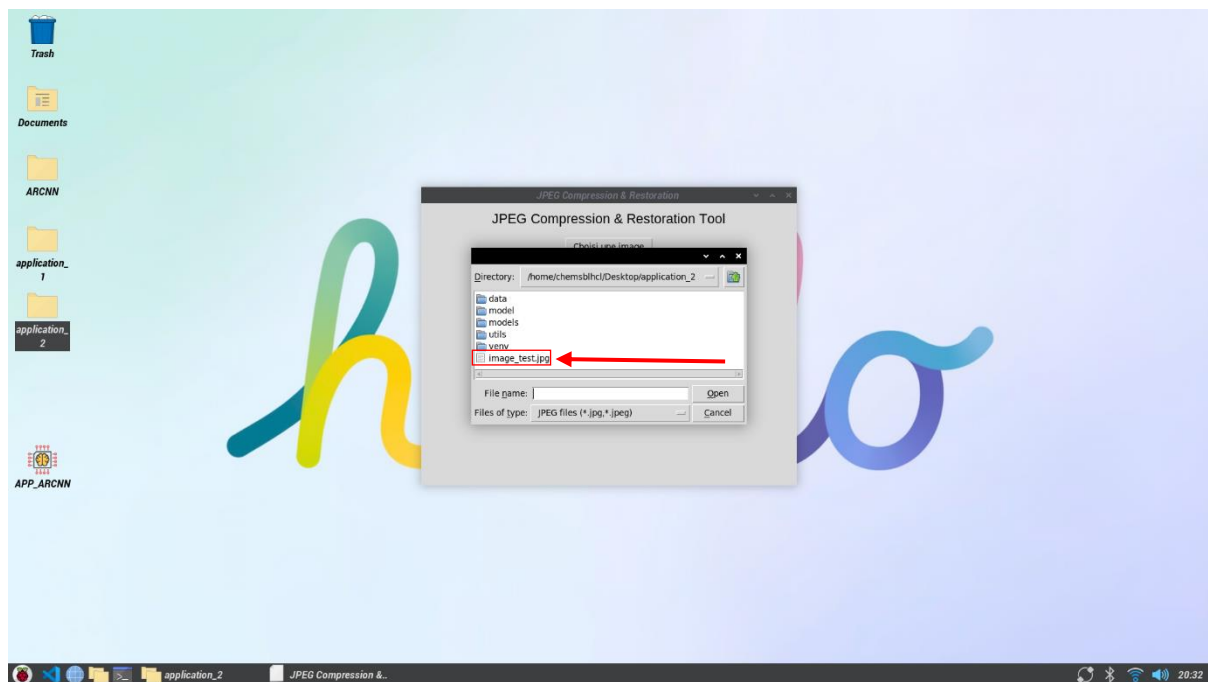


Figure 62 : Sélectionner l'image.

- Choix du facteur de qualité (QF):



Figure 63 : Choix du Facteur de compression.

- Compression et restauration avec ARCNN

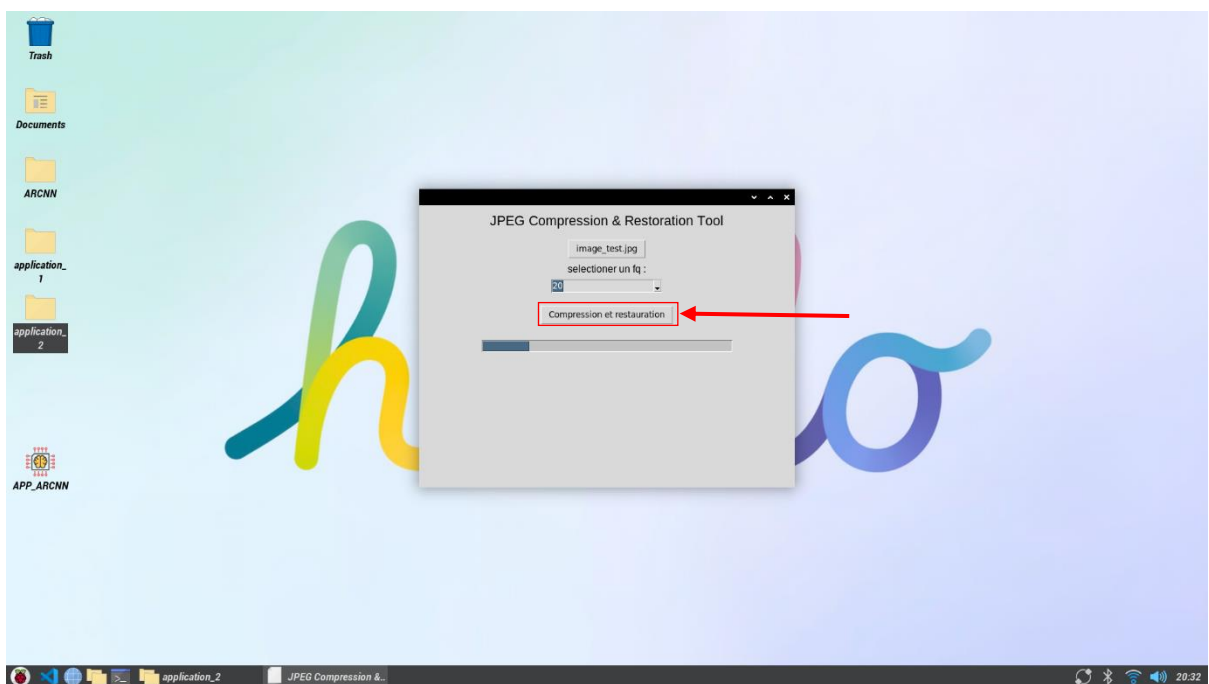


Figure 64 : Compression et Restauration de l'image.

- Production et comparaison :

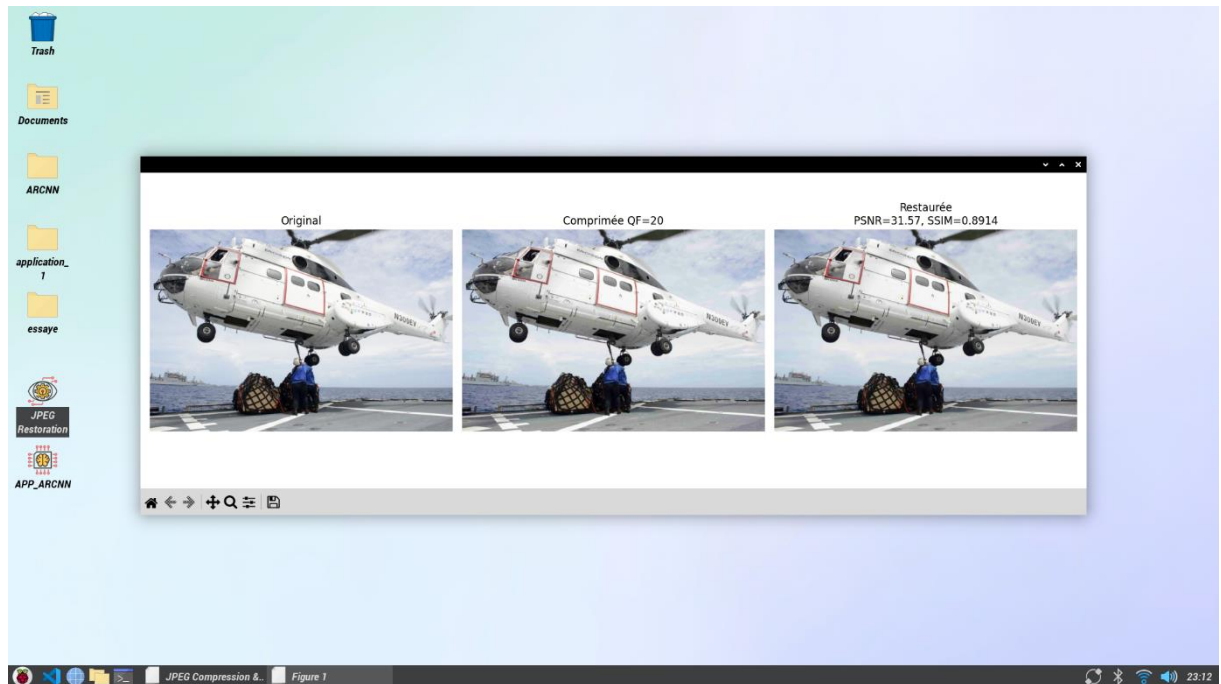


Figure 65 : Affichage de résultats.

7.2. Application temps réel :

Contrairement à l'application temps différé, cette application est conçue pour fonctionner en flux continu, ce qui permet de traiter en direct les images issues d'une caméra connectée au système. L'objectif est d'assurer une restauration des images dégradées en temps réel, avec une latence minimale.

- Ouverture de l'application :

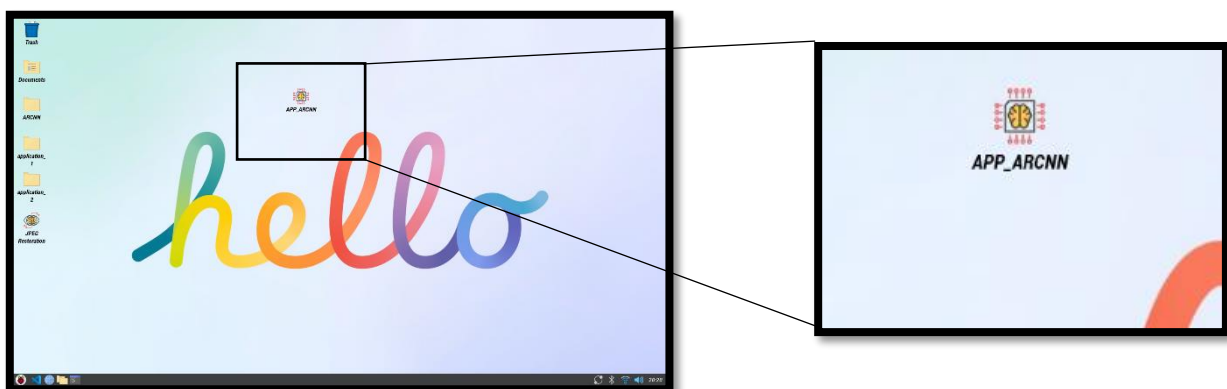


Figure 66 : Ouverture de l'application 2.

- Connexion au réseau local :

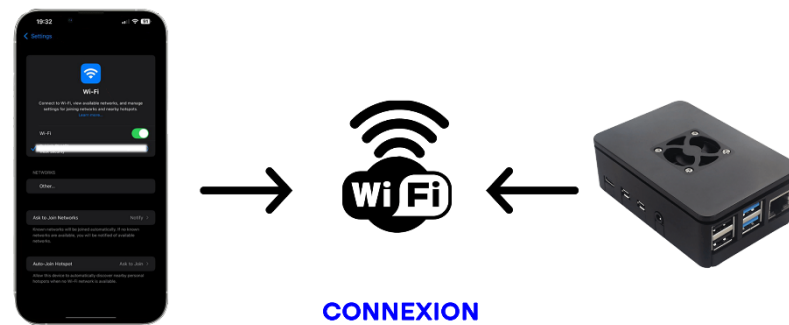


Figure 67 : Connexion au réseau local.

- Lancement du serveur sur Raspberry Pi :

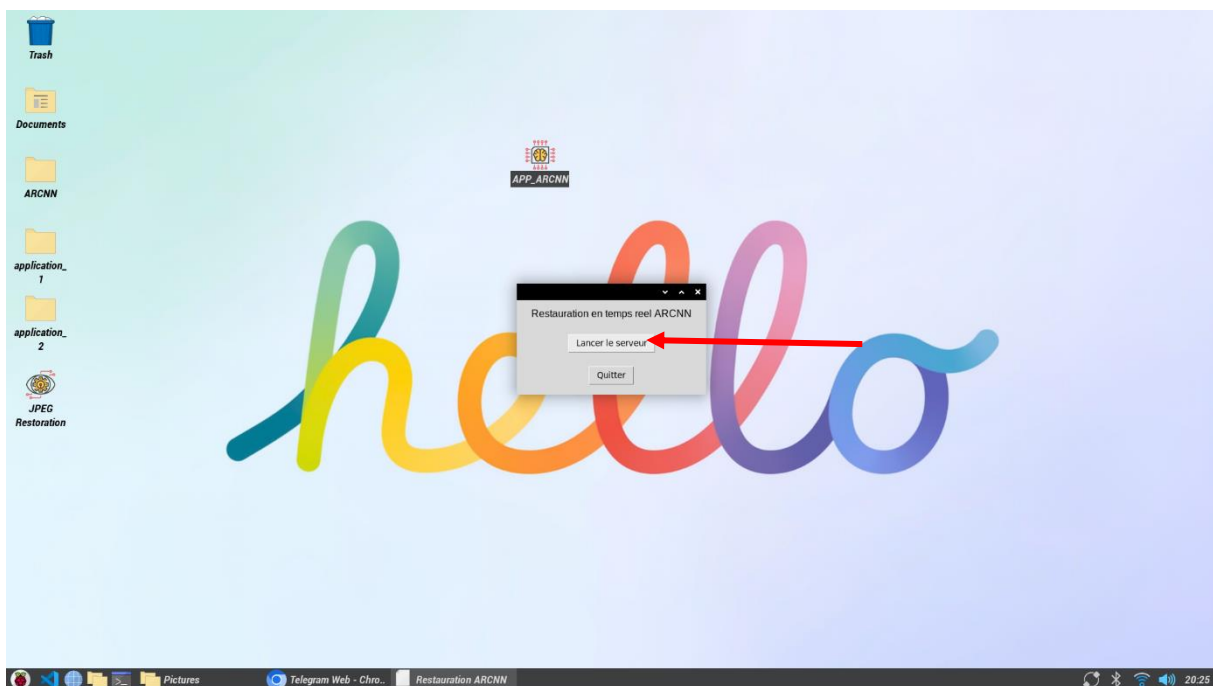


Figure 68 : Lancement du serveur.

- Acquisition et transmission des images :

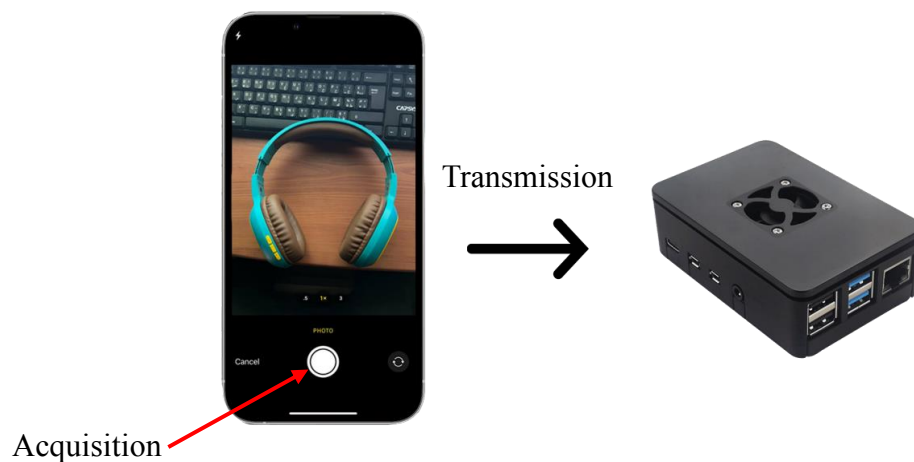


Figure 69 : Acquisition et transmission.

- Restauration et Affichage en temps réel :

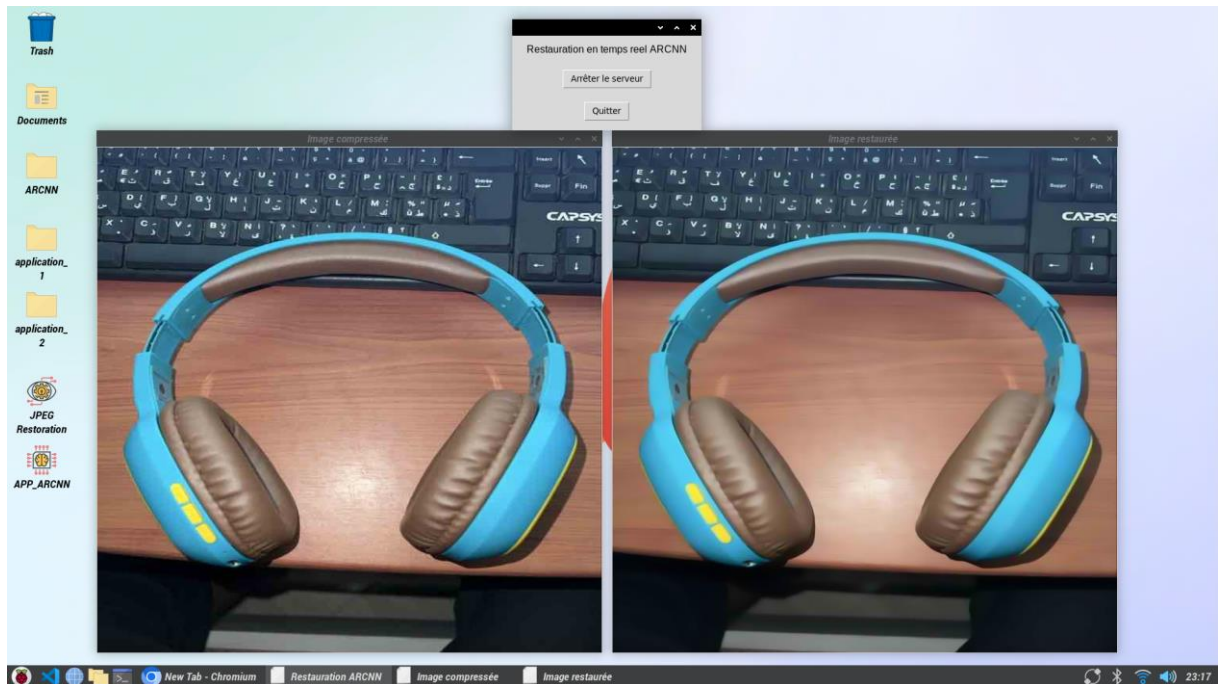


Figure 70 : Restauration et affichage du résultat.

8. Discussion :

Les résultats obtenus mettent en évidence l'efficacité du modèle ARCNN dans la restauration d'images compressées au format JPEG. L'évaluation quantitative a montré une augmentation significative des valeurs PSNR et SSIM, traduisant une meilleure fidélité des images restaurées par rapport aux versions compressées. L'évaluation qualitative confirme également une nette amélioration visuelle, avec une réduction des artefacts de blocs, une meilleure netteté et une restitution plus naturelle des contours et des textures.

Le choix du modèle ARCNN s'avère particulièrement judicieux dans notre contexte, car il conjugue simplicité architecturale, rapidité d'exécution et compatibilité avec des systèmes embarqués tels que le Raspberry Pi 4. Sa structure relativement peu profonde, composée de seulement quatre couches convolutives, lui confère une efficacité appréciable tout en réduisant les besoins en calcul et en mémoire, ce qui le distingue de modèles plus complexes comme DnCNN, FFDNet ou DRUNet. Néanmoins, malgré ces avantages, certaines limites persistent. D'une part, les performances d'ARCNN demeurent étroitement liées à la diversité et à la représentativité des données d'entraînement : un jeu de données restreint ou trop homogène risque de réduire la capacité du modèle à généraliser à différents types d'images et à divers niveaux de compression. D'autre part, bien que plus léger que d'autres architectures, son implémentation sur des dispositifs embarqués reste soumise à des contraintes matérielles importantes, telles que la puissance de calcul limitée, la mémoire restreinte et l'absence d'accélérateur graphique dédié, ce qui complique le traitement d'images en haute résolution ou en temps réel.

Ces constats ouvrent la voie à plusieurs perspectives d'amélioration, telles que l'utilisation de modèles plus légers et optimisés pour l'embarqué, l'élargissement des bases de données

d'entraînement afin de renforcer la robustesse, ou encore l'intégration de techniques hybrides combinant restauration d'images et compression adaptative.

Conclusion

Conclusion

À travers ce mémoire, nous avons étudié en profondeur le problème de la compression d'images et des artefacts visuels générés par le format JPEG, puis exploré les approches modernes basées sur le deep learning pour la restauration et l'amélioration de la qualité d'images dégradées.

Dans un premier temps, nous avons rappelé les principes fondamentaux de la compression (avec ou sans perte), ses avantages et limites, en insistant sur les artefacts typiques du JPEG tels que l'effet de bloc, le flou et les distorsions de couleur. Ensuite, nous avons analysé plusieurs architectures de réseaux de neurones dédiées à la restauration d'images (DnCNN, FFDNet, DRUNet), et retenu le modèle ARCNN comme étant le plus adapté à notre contexte expérimental et à une implémentation sur une plateforme embarquée comme le Raspberry Pi 4.

La méthodologie mise en place a permis d'entraîner et d'évaluer ce modèle à partir de jeux de données compressés et préparés, en appliquant des métriques objectives (PSNR, SSIM) et une analyse qualitative visuelle.

Les résultats ont montré une amélioration significative de la qualité des images :

- Réduction notable des effets de blocs,
- Récupération partielle des détails fins,
- Meilleure restitution des contours et transitions de couleur.

Ces améliorations ont été quantifiées et confirmées à travers les métriques objectives de qualité d'image, en particulier le PSNR (Peak Signal-to-Noise Ratio) et le SSIM (Structural Similarity Index). Après les tests réalisés sur notre modèle ARCNN, les résultats obtenus montrent une nette progression par rapport aux images compressées initialement :

- Pour un facteur de compression 10, le modèle a permis d'atteindre un PSNR de 29,93 dB et un SSIM de 0,8708, traduisant une réduction notable des artefacts et une meilleure homogénéité visuelle.
- Avec un facteur de compression 20, les performances se sont encore améliorées avec un PSNR de 31,57 dB et un SSIM de 0,8914, indiquant une récupération plus précise des détails et une restitution plus fidèle des structures locales.
- Enfin, pour un facteur de compression 40, le modèle a obtenu un PSNR de 32,30 dB et un SSIM de 0,9014, confirmant sa capacité à préserver efficacement la structure et les contrastes de l'image, tout en atténuant fortement les distorsions visuelles.

Ces résultats démontrent que le modèle ARCNN ne se limite pas à un simple lissage des blocs, mais contribue réellement à reconstruire l'information visuelle perdue, en améliorant à la fois la fidélité quantitative (PSNR, SSIM) et la qualité perceptuelle des images traitées.

En conclusion, ce travail met en évidence le potentiel des modèles de deep learning pour la restauration d'images compressées, en offrant une qualité visuelle supérieure aux méthodes traditionnelles. Toutefois, leur déploiement sur des systèmes embarqués reste un

Conclusion

défi, en raison des contraintes matérielles (mémoire, calcul, absence de GPU). Des pistes d'amélioration peuvent être envisagées, notamment l'optimisation des architectures, la compression des modèles, ou l'utilisation de méthodes hybrides combinant traitement classique et intelligence artificielle.

Ainsi, notre mémoire ouvre la voie à des perspectives prometteuses pour des applications concrètes telles que la télémédecine (où la transmission d'images médicales de haute qualité avec une bande passante limitée est cruciale pour le diagnostic à distance), la vidéosurveillance et la sécurité (permettant une meilleure analyse d'images compressées en temps réel tout en préservant les détails critiques), ou encore la photographie numérique et mobile (avec des algorithmes intégrés dans les appareils photo et smartphones pour améliorer automatiquement les images compressées).

Au-delà de ces applications directes, de nouvelles perspectives peuvent être envisagées dans des domaines variés :

- Réalité augmentée et réalité virtuelle, où la qualité des images conditionne directement l'expérience utilisateur.
- Véhicules intelligents et systèmes de transport autonomes, qui doivent traiter rapidement et efficacement des images compressées pour prendre des décisions fiables.
- Archivage et restauration numérique (musées, bibliothèques numériques), où la restauration automatique d'images anciennes ou fortement compressées constitue un atout majeur pour la conservation du patrimoine.
- Applications mobiles et objets connectés (IoT, drones, lunettes intelligentes), qui nécessitent des modèles légers et optimisés pour fonctionner sur des dispositifs contraints en ressources.
- Cloud computing et edge computing, qui offrent la possibilité de déployer des modèles de restauration d'images à grande échelle, tout en optimisant la consommation énergétique et en réduisant la latence.

À plus long terme, l'intégration de techniques avancées de compression et de débruitage dans ces environnements pourrait contribuer à une intelligence artificielle plus durable, capable de concilier qualité visuelle, performance et sobriété énergétique, répondant ainsi aux besoins croissants de la société numérique.

Références bibliographiques

- [1] Wallace, G. K. (1992). "The JPEG still picture compression standard." *Communications of the ACM*, 34(4), 30–44.
- [2] Goodfellow, I., Bengio, Y., & Courville, A. (2016). "Deep Learning." MIT Press, 800 p.
- [3] Sayood, K. (2017). "Introduction to Data Compression." 5th ed., Morgan Kaufmann, 790 p.
- [4] Pennebaker, W. B., & Mitchell, J. L. (1993). "JPEG: Still Image Data Compression Standard." Springer, 638 p.
- [5] Saha, S. (2000). "Image compression — from DCT to wavelets: A review." *ACM Crossroads*, 6(3). (Article en ligne, non paginé).
- [6] Gonzalez, R. C., & Woods, R. E. (2018). "Digital Image Processing." 4th ed., Pearson, 1168 p.
- [7] Zhang, K., Zuo, W., Chen, Y., Meng, D., & Zhang, L. (2017). "Beyond a Gaussian denoiser: Residual learning of deep CNN for image denoising." *IEEE Transactions on Image Processing*, 26(7), 3142–3155.
- [8] LeCun, Y., Bengio, Y., & Hinton, G. (2015). "Deep learning." *Nature*, 521(7553), 436–444.
- [9] Tian, C., Fei, L., Zheng, W., Xu, Y., Zuo, W., Lin, C.-W., & Zhang, D. (2020). "Deep learning on image denoising: An overview." *Neural Networks*, 131, 251–275.
- [10] Dong, C., Loy, C. C., & Tang, X. (2015). "Compression artifacts reduction by a deep convolutional network." In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 576–584.
- [11] Svoboda, P., Hradis, M., & Zemcik, P. (2016). "Compression artifacts removal using convolutional neural networks." In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, 3903–3907.
- [12] Zhang, K., Zuo, W., & Zhang, L. (2018). "FFDNet: Toward a fast and flexible solution for CNN-based image denoising." *IEEE Transactions on Image Processing*, 27(9), 4608–4622.
- [13] Buades, A., Coll, B., & Morel, J. M. (2005). "A non-local algorithm for image denoising." In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2, 60–65.
- [14] Wang, L., Li, Y., Li, K., Zhang, K., Zhong, B., & Fu, Y. (2021). "Unsupervised degradation representation learning for blind super-resolution." *IEEE Transactions on Image Processing*, 30, 6859–6871.
- [15] Ronneberger, O., Fischer, P., & Brox, T. (2015). "U-Net: Convolutional networks for biomedical image segmentation." In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 234–241.

Références bibliographiques

- [16] Lane, N. D., Bhattacharya, S., Georgiev, P., Forlivesi, C., & Kawsar, F. (2017). "An early resource characterization of deep learning on wearables, smartphones and Internet-of-Things devices." In *Proceedings of the International Workshop on Internet of Things towards Applications*, 7–12.
- [17] Deng, L., Li, G., Han, S., Shi, L., & Xie, Y. (2020). "Model compression and hardware acceleration for neural networks: A comprehensive survey." *Proceedings of the IEEE*, 108(4), 485–532.
- [18] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising," *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017.
- [19] C. Dong, Y. Deng, C. C. Loy, and X. Tang, "Compression Artifacts Reduction by a Deep Convolutional Network," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 576–584, 2015.
- [20] K. Zhang, W. Zuo, and L. Zhang, "FFDNet: Toward a Fast and Flexible Solution for CNN-based Image Denoising," *IEEE Transactions on Image Processing*, vol. 27, no. 9, pp. 4608–4622, 2018.
- [21] K. Zhang, Y. Li, W. Zuo, L. Zhang, L. Van Gool, and R. Timofte, "Plug-and-Play Image Restoration with Deep Denoiser Prior," arXiv preprint, arXiv:2008.13751, 2020.
- [22] Banbury, C. R., Reddi, V. J., Lam, M., Fu, W., et al. (2021). "Benchmarking TinyML systems: Challenges and direction." *arXiv preprint arXiv:2003.04821*, 22 p.
- [23] Han, S., Mao, H., & Dally, W. J. (2016). "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding." In *Proceedings of the International Conference on Learning Representations (ICLR)*. (En ligne, non paginé).
- [24] COCO Dataset. (2021). "Common Objects in Context (COCO)." [En ligne]. Disponible sur : <https://cocodataset.org>
- [25] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). "PyTorch: An imperative style, high-performance deep learning library." In *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 8024–8035.
- [26] TorchVision. (2023). "TorchVision Documentation." [En ligne]. Disponible sur : <https://pytorch.org/vision/stable>
- [27] Pillow. (2023). "Pillow Documentation." [En ligne]. Disponible sur : <https://python-pillow.org>
- [28] Bisong, E. (2019). "Google Colaboratory." In *Building Machine Learning and Deep Learning Models on Google Cloud Platform* (pp. 59–64). Apress, Berkeley, CA.
- [29] Johnson, J., Alahi, A., & Fei-Fei, L. (2016). "Perceptual losses for real-time style transfer and super-resolution." In *Proceedings of the European Conference on Computer Vision (ECCV)*, 694–711.
- [30] Hore, A., & Ziou, D. (2010). "Image quality metrics: PSNR vs. SSIM." In *Proceedings of the 20th International Conference on Pattern Recognition (ICPR)*, 2366–2369.

Références bibliographiques

- [31] A. Hore and D. Ziou, “*Image Quality Metrics: PSNR vs. SSIM*”, 20th International Conference on Pattern Recognition (ICPR), pp. 2366–2369, 2010.
- [32] Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). "Image quality assessment: From error visibility to structural similarity." *IEEE Transactions on Image Processing*, 13(4), 600–612.
- [33] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, “*Image Quality Assessment: From Error Visibility to Structural Similarity*”, *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.
- [34] Raspberry Pi Foundation. (2023). "Raspberry Pi 4 Model B." [En ligne]. Disponible sur : <https://www.raspberrypi.com/products/raspberry-pi-4-model-b>
- [35] Raspberry Pi Foundation. (2023). "Raspberry Pi OS Software." [En ligne]. Disponible sur : <https://www.raspberrypi.com/software>
- [36] Torch DirectML. (2023). "Precompiled Torch wheels for Windows with DirectML." [En ligne]. Disponible sur : <http://torch.maklu.ml/whl/stable.html>