Aix-Marseille Université

Débruitage d'images par modèle DnCNN et U-Net

Antoine Legendre

Mathématiques pour la Science des Données

Année Universitaire 2025-2026

Table des matières

1 Introduction

Le débruitage d'images est une tâche fondamentale en vision par ordinateur qui consiste à restaurer une image propre à partir d'une observation corrompue par du bruit. Cette problématique trouve des applications dans de nombreux domaines tels que l'imagerie médicale, la photographie numérique et la télédétection. Les approches basées sur l'apprentissage profond ont révolutionné ce domaine en surpassant les méthodes traditionnelles grâce à leur capacité à apprendre des représentations complexes des données.
Ce travail pratique a pour objectif d'étudier et de comparer deux architectures de réseaux de neurones convolutionnels pour le débruitage d'images : le DnCNN (Deep Convolutional Neural Network) et l'U-Net. Nous nous focaliserons sur le jeu de données MNIST, contenant des images de chiffres manuscrits, auquel nous ajouterons un bruit gaussien synthétique. Notre démarche consistera à implémenter ces architectures en PyTorch, à évaluer leurs performances via des métriques objectives (MSE, PSNR) et visuelles, et à analyser l'influence de différents hyperparamètres architecturaux sur la qualité du débruitage.
Nous explorerons notamment l'impact du nombre de couches, de filtres et de blocs sur les performances des modèles, tout en considérant le compromis entre complexité du modèle (nombre de paramètres) et qualité de reconstruction. Cette étude comparative nous permettra de déterminer les configurations optimales pour chaque architecture et d'identifier leurs forces et limitations respectives dans le contexte du débruitage d'images.

2 Bruitage des images

Pour entraîner les modèles de débruitage, il est nécessaire de créer un jeu de données comportant à la fois des images propres et des images bruitées. Le code ci-dessus définit une classe `NoisyMNIST` héritant de `torch.utils.data.Dataset`. Cette classe encapsule le jeu de données MNIST et ajoute un bruit gaussien aux images. Le constructeur `__init__` permet de spécifier le chemin de téléchargement `root`, le mode (`train=True` pour l'ensemble d'apprentissage ou `False` pour l'ensemble de test) et l'écart-type du bruit gaussien `noise_std`. La méthode `__len__` renvoie le nombre d'exemples du dataset, et `__getitem__` renvoie pour un indice donné `idx` un couple (image bruitée, image propre). Le bruit gaussien est généré par `torch.randn_like(clean_img) * noise_std` et ajouté à l'image originale. La fonction `torch.clamp` permet de s'assurer que les pixels restent dans l'intervalle $[0, 1]$.
Enfin, les instances de `NoisyMNIST` sont utilisées pour créer des `DataLoader` de PyTorch, permettant de charger les données par batchs de taille `batch_size=128` et de mélanger aléatoirement l'ensemble d'apprentissage. Ces `DataLoader` facilitent l'entraînement et l'évaluation des modèles de débruitage sur MNIST bruité.

3 Architecture DnCNN

3.1 Le réseau

La classe `DnCNN` définit l'architecture complète d'un réseau de débruitage convolutionnel profond (DnCNN) en héritant de `torch.nn.Module`. Le constructeur `__init__` permet de spécifier le nombre de filtres par bloc (`num_of_features`), le nombre total de couches

(`num_of_layers`) et l'utilisation ou non d'une *skip connection* (`skip_connection`). L'architecture est construite sous la forme d'une séquence de couches (`nn.Sequential`) comprenant :

— une première convolution 2D de l'image d'entrée (1 canal) vers le nombre de filtres spécifié, suivie d'une activation ReLU ;
— un ensemble de blocs intermédiaires répétant la séquence `Conv2d → BatchNorm2d → ReLU` pour `num_of_layers - 2` couches ;
— une dernière convolution ramenant les canaux de sortie à 1.

La méthode `forward` applique le passage avant sur l'entrée `x` à travers la séquence de couches. Si la *skip connection* est activée, le réseau apprend à prédire le bruit résiduel et la sortie finale est calculée comme `y = x - out`. Sinon, la sortie correspond directement à la sortie du réseau ($y = out$). Cette approche permet au DnCNN de se concentrer sur l'apprentissage du bruit plutôt que de l'image entière, facilitant ainsi l'entraînement et améliorant la qualité du débruitage.

3.2 Entraînement du modèle

Le code définit ensuite la fonction de perte et l'optimiseur utilisés pour entraîner le modèle :

— `criterion = nn.MSELoss()` : la fonction de perte choisie est l'erreur quadratique moyenne (MSE), adaptée au problème de débruitage car elle mesure directement l'écart entre l'image débruitée et l'image propre.
— `optimizer = optim.Adam(..., lr=1e-3)` : l'optimiseur Adam est utilisé avec un taux d'apprentissage de 10^{-3}, permettant une convergence rapide et stable.

L'entraînement est effectué sur 5 époques :

1. Pour chaque mini-lot, les images bruitées et propres sont transférées sur le *device* (CPU ou GPU).
2. Le modèle produit des images débruitées via un passage avant.
3. La perte est calculée par rapport aux images propres.
4. Le gradient est réinitialisé (`optimizer.zero_grad()`), rétropropagé (`loss.backward()`) puis les poids du modèle sont mis à jour (`optimizer.step()`).
5. La perte moyenne par époque est affichée.

3.3 Evaluation du modèle

Le code réalise enfin l'évaluation du modèle après son apprentissage.

— Le modèle est placé en mode évaluation avec `model.eval()`, ce qui désactive notamment la mise à jour des poids et la normalisation en mode apprentissage.
— Dans une boucle sur l'ensemble de test, on calcule deux métriques :
    1. L'erreur quadratique moyenne (**MSE**) à l'aide de `F.mse_loss`.
    2. Le **PSNR** (Peak Signal-to-Noise Ratio), mesuré en dB, qui évalue la qualité de restitution du signal propre.
— Les résultats sont moyennés sur tous les lots du *test set*.
— Enfin, quelques exemples visuels sont affichés pour comparer les images bruitées, débruitées par le modèle et les images propres de référence.

3.4 Test de différentes configurations

On décide ensuite de tester différentes configurations en faisant varier les différent paramètres du réseau afin d'estimer la configuration représentant le meilleur compromis entre nombre de paramètres et qualité du débruitage. La *skip connection* permettant une meilleure qualité du débruitage tout en gardant le même nombre de paramètres nous la laisserons activée durant tous nos tests. Nous choisissons de faire varier le nombre de couches de 1 à 8 et le nombre de filtres par couche de 1 à 64. Nous obtenons alors les graphiques suivants de la MSE et du PSNR en fonction du nombre de paramètres :

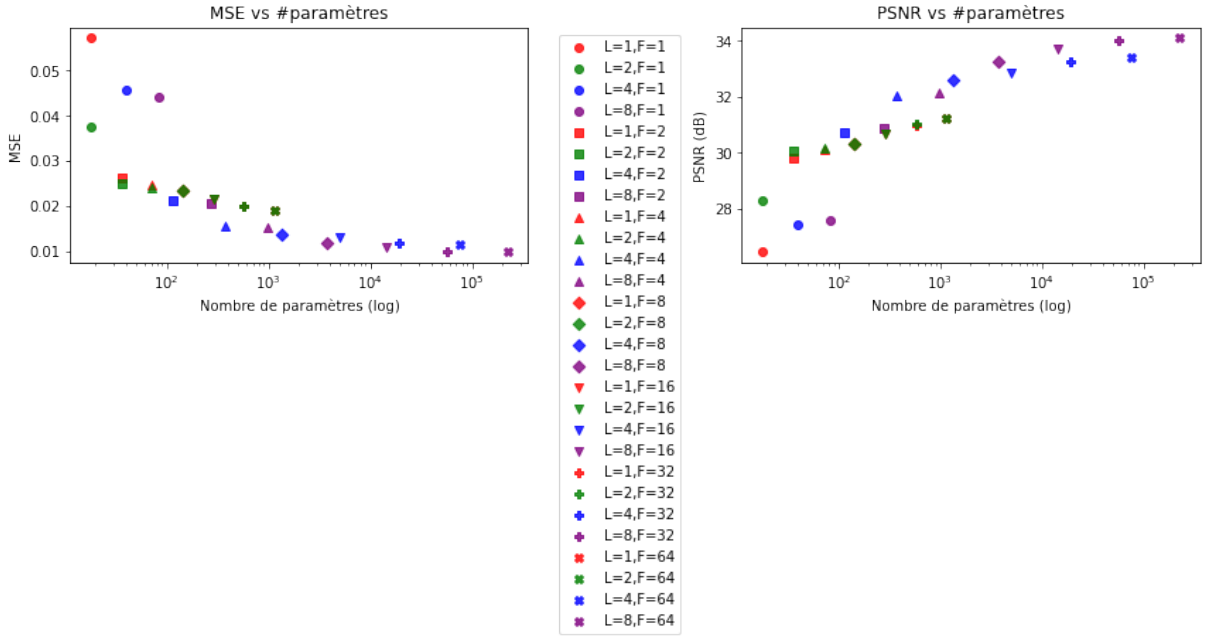

FIGURE 1 – MSE et PSNR en fonction du log du nombre de paramètres pour le modèle DnCNN

En observant les graphiques on remarque que le point ($L = 4, F = 4$ (4 couches et 4 filtres par couche) a une MSE assez faible (environ 0.015) et un PSNR assez élévé (environ 32 dB)pour un nombre de paramètres assez faible. En regardant plus précisément les résultats de cette configuration, on obtient une MSE d'environ 1.57×10^{-2}, un PSNR de 32.05 dB pour seulement 376 paramètres. Ces valeurs ($MSE \approx 10^{-2}$ et $PSNR > 30dB$) indiquent que le modèle DnCNN avec 4 couches et 4 filtres par couche est capable de réduire efficacement le bruit des images MNIST, avec une très bonne fidélité par rapport aux images originales. En reconstruisant quelques images avec cette configuration on obtient alors les images suivantes :

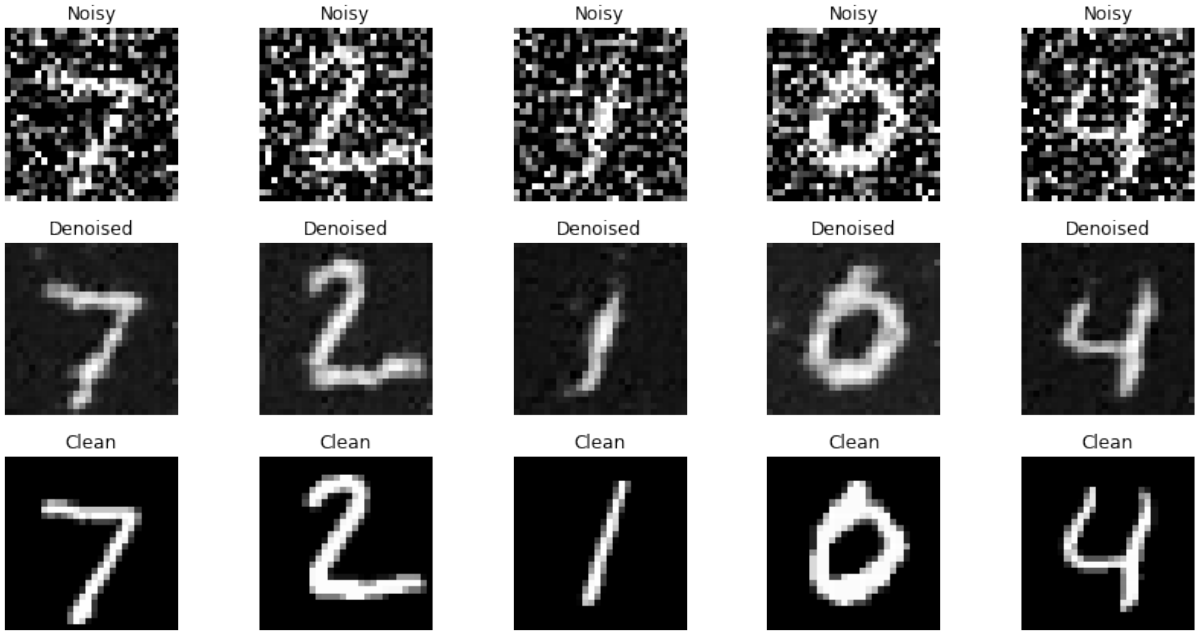

FIGURE 2 – Images bruitées, images débruitées par le modèle DnCNN et images propres

On observe bien que les images débruitées sont beacoup plus lisibles que les images bruitées et assez proches des images propres. On pourrait bien sûr augmenter le nombre de couches et le nombre de filtres par couches afin de s'approcher encore plus des images propres mais cela se ferait aux dépens du nombre de paramètres et donc du temps d'exécution du code. Enfin cette configuration composée de 4 couches et de 4 filtres par couches est suffisante pour un jeu de données assez simple comme le MNIST utilisé ici mais pourrait s'avérer insuffisante pour des jeux de données plus complexes. Il faudrait alors augmenter le nombre de couches et le nombre de filtres afin d'avoir des images débruitées de bonne qualité.

4 Architecture U-Net

4.1 Le réseau

Le code implémente la classe `UNet`, une architecture de réseau de neurones convolutionnel conçue à l'origine pour la segmentation d'images médicales, mais également très adaptée à des tâches comme le débruitage d'images. Elle repose sur une structure en forme de U, composée d'une **partie contractante (encodeur)**, d'un **goulot d'étranglement (bottleneck)** et d'une **partie expansive (décodeur)**, avec des *skip connections* entre les deux parties.

- **Partie contractante** : elle est constituée d'une succession de `ContractingBlock`. Chaque bloc applique deux convolutions 3×3 suivies de normalisation et d'une activation `ReLU`, puis d'un sous-échantillonnage via un max-pooling. À chaque niveau, le nombre de filtres est doublé, ce qui permet de capturer des caractéristiques de plus en plus complexes mais à plus petite échelle spatiale. Les sorties convolutionnelles (`x_conv`) sont sauvegardées dans une liste `skips` pour être réutilisées dans la partie expansive.
- **Bottleneck** : c'est la couche centrale du U-Net. Elle se compose de deux convolutions 3×3 avec normalisation et activation, qui extraient des caractéristiques de

haut niveau tout en se situant à la résolution la plus basse. Elle forme le "fond du U".

— **Partie expansive** : elle est symétrique de la partie contractante et composée de `ExpansiveBlock`. Chaque bloc commence par une convolution transposée (*up-convolution*) qui double la résolution spatiale, puis concatène la sortie avec la carte correspondante de la partie contractante (récupérée via `skips.pop()`). Ensuite, deux convolutions classiques affinent la reconstruction. Le nombre de filtres est divisé par deux à chaque étape, ce qui réduit progressivement la complexité des représentations.

— **Couche finale** : un simple `Conv2d` avec noyau 1×1 transforme la sortie en l'image finale, avec le nombre de canaux de sortie (`out_channels`) désiré (par exemple 1 pour une image MNIST débruitée).

Le passage dans le `forward` suit donc ce schéma : l'entrée traverse successivement les blocs contractants (`skips` mémorise les sorties utiles), puis le bottleneck. Ensuite, les blocs expansifs reconstruisent l'image en exploitant à la fois les informations globales (descendues jusqu'au bottleneck) et les détails locaux (issus des skip connections). L'architecture permet ainsi de combiner contexte global et précision spatiale.

4.2 Test de différentes configurations

L'entraînement et l'évaluation du modèle se faisant de manière similaire au modèle DnCNN, nous allons pouvoir passer au test de différentes configurations du modèle U-Net. Pour cela, on va faire varier le nombre de blocs contractants et expansifs de 1 à 3 (1 bloc contractant et 1 bloc expansif à 3 blocs contractants et 3 blocs expansifs) ainsi que le nombre de filtres de départ de 1 à 64 et on va tracer les graphiques de la MSE et du PSNR en fonction du nombre de paramètres. On obtient alors le graphique suivant où L est le nombre de blocs et F le nombre de filtres de départ :

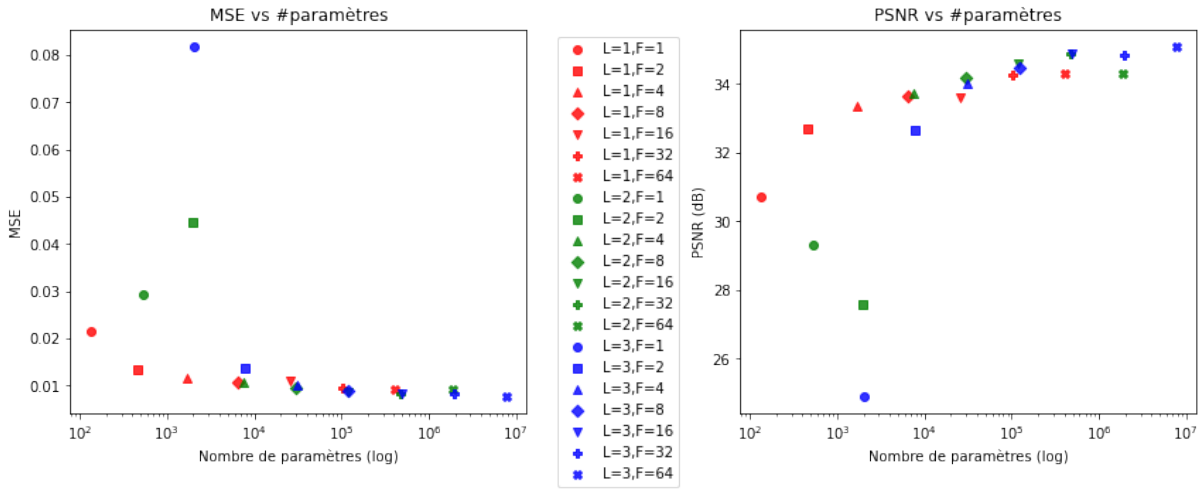

FIGURE 3 – MSE et PSNR en fonction du log du nombre de paramètres pour le modèle U-Net

On remarque que le point correspondant a 1 bloc contractant et 1 bloc expansif et 2 filtres de départ a une MSE assez faible (environ 0.014) et un PSNR assez élevé (environ 32.5 dB) pour un nombre assez faible de paramètres. En regardant plus précisément les résultats de cette configuration, on note une MSE d'environ 1.35×10^{-2} et un PSNR de

32.70 dB, ce qui sont des résultats très satisfaisants pour seulement 463 paramètres. Ce modèle ayant peu de paramètres il va nous permettre de débruiter efficacement les images. La figure suivante présente le résultat de ce débruitage pour 5 images :

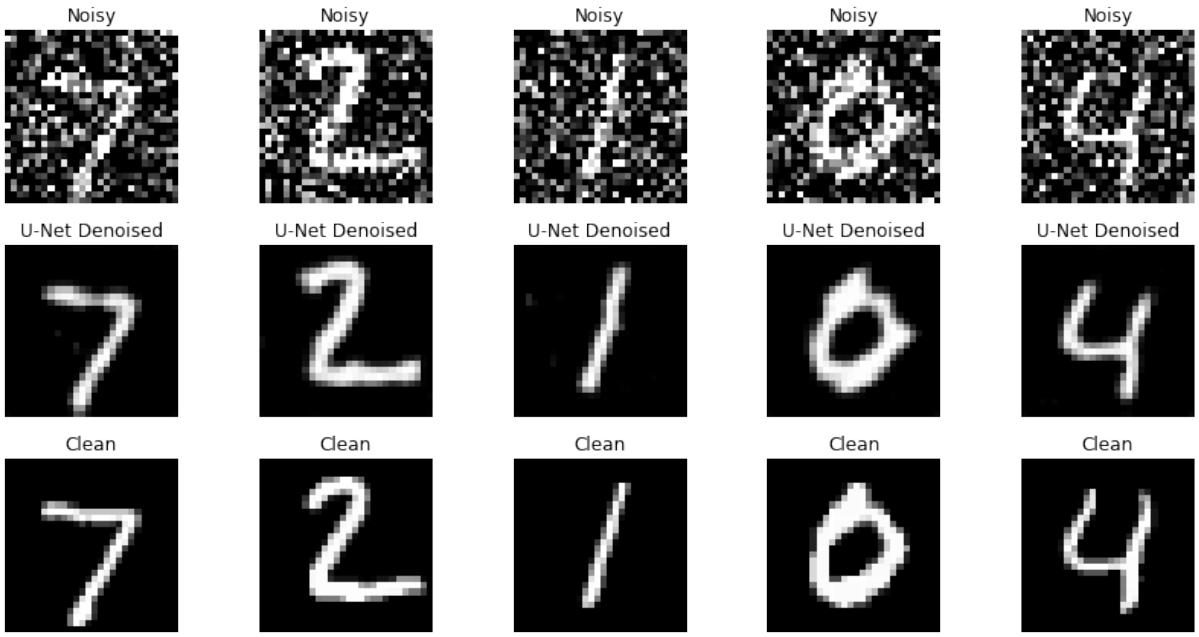

FIGURE 4 – Images bruitées, images débruitées par le modèle U-Net et images propres

Les images débruitées sont lisibles et assez proches des images propres. On remarque que malgré un nombre de paramètres assez similaire au réseau DnCNN avec 4 couches et 4 filtres par couche (376 pour le DnCNN contre 463 pour le U-Net) le résulat sur ces 5 images semble meilleur avec le U-Net ce qui est confirmé par les valeurs observées de MSE (1.57×10^{-2} contre 1.35×10^{-2}) et de PSNR (32.05 dB contre 32.70 dB). Le modèle U-Net semble donc plus adapté pour le débruitage de ce type d'images et serait encore plus adapté pour des images plus complexes en augmentant le nombre de blocs (on a ici un unique bloc) et le nombre de filtres de départ (seulement 2 ici).
Par exemple, en modifiant légèrement l'architecture du réseau afin de l'adapter à un jeu de données plus complexe tel que CIFAR-10, il est possible d'obtenir un modèle capable de traiter des images en couleurs de dimension 32×32 sur trois canaux. Dans ce cas, le U-Net conserve sa structure en "encodeur-décodeur" avec des connexions de type *skip connections*, mais le nombre de filtres de départ est augmenté afin de mieux capturer la richesse spectrale et spatiale du jeu de données. L'entraînement est réalisé en injectant un bruit gaussien additif sur les images d'entrée et en minimisant l'erreur quadratique moyenne entre les images débruitées et les images propres. Les résultats obtenus montrent que le modèle parvient non seulement à restaurer efficacement les détails visuels mais également à améliorer les métriques quantitatives telles que le MSE et le PSNR, confirmant ainsi la pertinence de l'utilisation du U-Net pour des tâches de débruitage sur des ensembles de données plus variés et réalistes. Par exemple pour le jeu de données CIFAR-10 et pour un réseau U-Net composé de 3 canaux d'entrée et de sortie, 3 blocs et 32 filtres d'entrées (1 927 459 paramètres pour des images de taille 32×32) on obtient une MSE de 2.12×10^{-3} et les images suivantes :

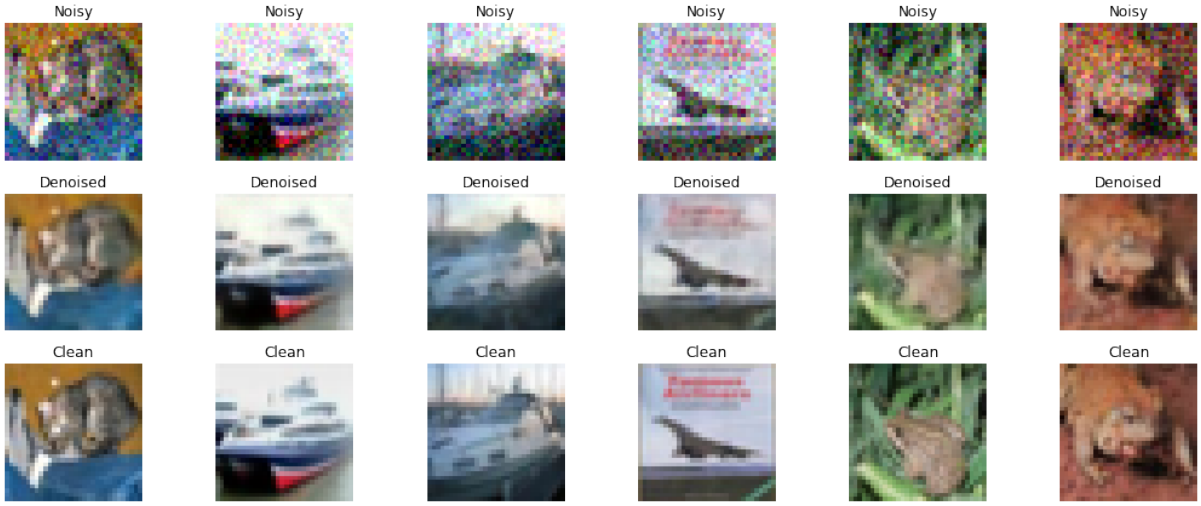

FIGURE 5 – Images bruitées, images débruitées par le modèle U-Net et images propres

On remarque bien que les images débruitées par le U-Net sont très proches des images propres et de très bonne qualité comme le suggérait la faible valeur de la MSE.

5 Conclusion

Ce travail nous a permis de mettre en évidence l'efficacité des réseaux de neurones convolutionnels pour le débruitage d'images. Dans un premier temps, nous avons constaté que le DnCNN, même avec un nombre limité de paramètres, parvient à produire des résultats satisfaisants sur le jeu de données MNIST. Toutefois, la comparaison avec le U-Net a montré que ce dernier obtient des performances supérieures (MSE plus faible et PSNR plus élevé) pour un nombre de paramètres comparable.
Cette supériorité s'explique notamment par l'architecture en encodeur-décodeur avec *skip connections*, qui permet de combiner des informations globales et locales lors de la reconstruction. Nous avons également observé que l'augmentation du nombre de blocs et du nombre de filtres de départ renforce encore les capacités de débruitage du U-Net, particulièrement pour des données plus complexes comme CIFAR-10.
En conclusion, si le DnCNN constitue un modèle efficace et léger pour des données simples, le U-Net apparaît comme une solution plus robuste et extensible, adaptée à des images variées et réalistes. Une perspective naturelle de ce travail serait d'évaluer des architectures encore plus profondes, ou d'explorer d'autres fonctions de perte, afin d'améliorer davantage la qualité visuelle des images reconstruites.

Annexes

Bruitage

```python
class NoisyMNIST(Dataset):
    def __init__(self, root="./data", train=True, noise_std=0.5):
        self.mnist = datasets.MNIST(
            root=root, train=train, download=True,
            transform=transforms.ToTensor()
        )
        self.noise_std = noise_std

    def __len__(self):
        return len(self.mnist)

    def __getitem__(self, idx):
        clean_img, _ = self.mnist[idx]
        noise = torch.randn_like(clean_img) * self.noise_std
        noisy_img = torch.clamp(clean_img + noise, 0., 1.)
        return noisy_img, clean_img
```

```python
batch_size = 128
train_dataset = NoisyMNIST(train=True, noise_std=0.5)
test_dataset = NoisyMNIST(train=False, noise_std=0.5)

train_loader = DataLoader(train_dataset, batch_size=batch_size,
   shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
   shuffle=False)
```

Le réseau DnCNN

```
class DnCNN_Block(nn.Module):
    def __init__(self, num_of_features=4):
        super(DnCNN_Block, self).__init__()

        self.conv = nn.Conv2d(
            in_channels=num_of_features,
            out_channels=num_of_features,
            kernel_size=3,
            padding=1,
            bias=False
        )

        self.bn = nn.BatchNorm2d(num_of_features)

        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        y = self.conv(x)
        y = self.bn(y)
        y = self.relu(y)
        return y
```

```python
class DnCNN(nn.Module):
    def __init__(self, num_of_features=4, num_of_layers=4, skip_connection=True):
        super(DnCNN, self).__init__()

        self.skip_connection = skip_connection

        layers = []

        layers.append(nn.Conv2d(1, num_of_features, kernel_size=3, padding=1, bias=False))
        layers.append(nn.ReLU(inplace=True))

        for _ in range(num_of_layers - 2):
            layers.append(nn.Conv2d(num_of_features, num_of_features, kernel_size=3, padding=1, bias=False))
            layers.append(nn.BatchNorm2d(num_of_features))
            layers.append(nn.ReLU(inplace=True))

        layers.append(nn.Conv2d(num_of_features, 1, kernel_size=3, padding=1, bias=False))

        self.dncnn = nn.Sequential(*layers)

    def forward(self, x):
        out = self.dncnn(x)

        if self.skip_connection:
            y = x - out
        else:
            y = out

        return y
```

Entraînement

```python
model = DnCNN(num_of_features=4, num_of_layers=4, skip_connection=True)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

```
criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(), lr=1e-3)

num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    for noisy_imgs, clean_imgs in train_loader:
        noisy_imgs, clean_imgs = noisy_imgs.to(device), clean_imgs.
           to(device)

        outputs = model(noisy_imgs)

        loss = criterion(outputs, clean_imgs)

        optimizer.zero_grad()
        loss.backward()

        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len
       (train_loader):.6f}")
```

Evaluation

```
def psnr(clean, denoised):
    mse = F.mse_loss(denoised, clean, reduction='sum') / clean.
       numel()
    n = clean[0].numel()
    return 10 * torch.log10(4 * n / (mse * clean.size(0)))
```

```
model.eval()
test_mse, test_psnr = 0.0, 0.0

with torch.no_grad():
    for noisy, clean in test_loader:
        noisy, clean = noisy.to(device), clean.to(device)

        denoised = model(noisy)

        mse_batch = F.mse_loss(denoised, clean, reduction="mean").
           item()
        test_mse += mse_batch

        psnr_batch = psnr(clean, denoised).item()
        test_psnr += psnr_batch

test_mse /= len(test_loader)
test_psnr /= len(test_loader)

print(f"MSE (test): {test_mse:.6f}")
print(f"PSNR (test): {test_psnr:.2f} dB")

noisy, clean = next(iter(test_loader))
noisy, clean = noisy.to(device), clean.to(device)
with torch.no_grad():
    denoised = model(noisy)

num_show = 5
fig, axes = plt.subplots(3, num_show, figsize=(12, 6))
for i in range(num_show):
    axes[0, i].imshow(noisy[i].cpu().squeeze(), cmap="gray")
    axes[0, i].set_title("Noisy")
    axes[0, i].axis("off")

    axes[1, i].imshow(denoised[i].cpu().squeeze(), cmap="gray")
    axes[1, i].set_title("Denoised")
    axes[1, i].axis("off")

    axes[2, i].imshow(clean[i].cpu().squeeze(), cmap="gray")
    axes[2, i].set_title("Clean")
    axes[2, i].axis("off")

plt.tight_layout()
plt.show()
```

Résultats pour le DnCNN

TABLE 1 – Résultats du DnCNN pour différentes configurations (L = nombre de couches, F = nombre de filtres)

L	F	Paramètres	MSE	PSNR (dB)
1	1	18	0.057212	26.44
	2	36	0.026296	29.81
	4	72	0.024545	30.11
	8	144	0.023488	30.30
	16	288	0.021517	30.68
	32	576	0.020103	30.97
	64	1152	0.018914	31.24
2	1	18	0.037576	28.26
	2	36	0.024954	30.04
	4	72	0.024179	30.17
	8	144	0.023381	30.32
	16	288	0.021580	30.67
	32	576	0.019998	31.00
	64	1152	0.019056	31.21
4	1	40	0.045749	27.41
	2	116	0.021297	30.73
	4	376	0.015697	32.05
	8	1328	0.013859	32.59
	16	4960	0.013157	32.82
	32	19136	0.011932	33.24
	64	75136	0.011495	33.40
8	1	84	0.044189	27.56
	2	276	0.020615	30.87
	4	984	0.015413	32.13
	8	3696	0.011925	33.25
	16	14304	0.010760	33.69
	32	56256	0.009947	34.03
	64	223104	0.009813	34.09

Le réseau U-Net

```python
class ContractingBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ContractingBlock, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
        self.pool = nn.MaxPool2d(2)

    def forward(self, x):
        x_conv = self.conv(x)
        x_down = self.pool(x_conv)
        return x_conv, x_down
```

```python
class ExpansiveBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ExpansiveBlock, self).__init__()
        self.upconv = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, skip):
        x = self.upconv(x)
        if x.size()[2:] != skip.size()[2:]:
            diffY = skip.size()[2] - x.size()[2]
            diffX = skip.size()[3] - x.size()[3]
            x = F.pad(x, [diffX // 2, diffX - diffX // 2,
                          diffY // 2, diffY - diffY // 2])
        x = torch.cat([x, skip], dim=1)
        x = self.conv(x)
        return x
```

```python
class UNet(nn.Module):
    def __init__(self, in_channels=1, out_channels=1, num_layers=1, features_start=2):
        super(UNet, self).__init__()

        self.contracting_blocks = nn.ModuleList()
        self.expansive_blocks = nn.ModuleList()

        features = features_start
        for i in range(num_layers):
            self.contracting_blocks.append(ContractingBlock(in_channels, features))
            in_channels = features
            features *= 2

        self.bottleneck = nn.Sequential(
            nn.Conv2d(in_channels, features, kernel_size=3, padding=1),
            nn.BatchNorm2d(features),
            nn.ReLU(inplace=True),
            nn.Conv2d(features, features, kernel_size=3, padding=1),
            nn.BatchNorm2d(features),
            nn.ReLU(inplace=True)
        )

        for i in range(num_layers):
            self.expansive_blocks.append(ExpansiveBlock(features, features // 2))
            features //= 2

        self.final_conv = nn.Conv2d(features, out_channels, kernel_size=1)

    def forward(self, x):
        skips = []
        for block in self.contracting_blocks:
            x_conv, x = block(x)
            skips.append(x_conv)

        x = self.bottleneck(x)

        for block in self.expansive_blocks:
            skip = skips.pop()
            x = block(x, skip)

        return self.final_conv(x)
```

Résultats pour le U-Net

TABLE 2 – Résultats du U-Net pour différentes configurations (L = nombre de couches, F0 = nombre de filtres de base)

L	F0	Paramètres	MSE	PSNR (dB)
1	1	134	0.021416	30.70
	2	463	0.013530	32.70
	4	1709	0.011671	33.34
	8	6553	0.010842	33.66
	16	25649	0.011005	33.60
	32	101473	0.009471	34.25
	64	403649	0.009338	34.31
2	1	528	0.029369	29.33
	2	1963	0.044443	27.57
	4	7557	0.010704	33.72
	8	29641	0.009628	34.18
	16	117393	0.008736	34.60
	32	467233	0.008231	34.86
	64	1864257	0.009318	34.32
3	1	2028	0.081680	24.92
	2	7811	0.013652	32.66
	4	30645	0.010020	34.00
	8	121385	0.008969	34.48
	16	483153	0.008214	34.87
	32	1927841	0.008247	34.85
	64	7701825	0.007830	35.08

Application à CIFAR-10

```
transform = transforms.Compose([
    transforms.ToTensor(),
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=
                                           transform)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=
                                          transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers
                                             =2)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
                                         shuffle=False, num_workers
                                            =2)
```

```
def add_gaussian_noise(images, mean=0.0, std=0.1):
    noise = torch.randn_like(images) * std + mean
    noisy_images = images + noise
    return torch.clamp(noisy_images, 0., 1.)
```

```
class ContractingBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
        self.pool = nn.MaxPool2d(2)

    def forward(self, x):
        x_conv = self.conv(x)
        x_down = self.pool(x_conv)
        return x_conv, x_down
```

```
class ExpansiveBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.upconv = nn.ConvTranspose2d(in_channels, out_channels,
            kernel_size=2, stride=2)
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, skip):
        x = self.upconv(x)
        if x.shape != skip.shape:
            diffY = skip.size()[2] - x.size()[2]
            diffX = skip.size()[3] - x.size()[3]
            x = F.pad(x, [diffX // 2, diffX - diffX // 2,
                          diffY // 2, diffY - diffY // 2])
        x = torch.cat([skip, x], dim=1)
        return self.conv(x)
```

```python
class UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, num_layers=3, features_start=32):
        super().__init__()
        self.contracting_blocks = nn.ModuleList()
        self.expansive_blocks = nn.ModuleList()

        features = features_start
        for i in range(num_layers):
            self.contracting_blocks.append(ContractingBlock(in_channels, features))
            in_channels = features
            features *= 2

        self.bottleneck = nn.Sequential(
            nn.Conv2d(in_channels, features, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(features, features, 3, padding=1),
            nn.ReLU(inplace=True)
        )

        for i in range(num_layers):
            self.expansive_blocks.append(ExpansiveBlock(features, features // 2))
            features //= 2

        self.final_conv = nn.Conv2d(features, out_channels, kernel_size=1)

    def forward(self, x):
        skips = []
        for block in self.contracting_blocks:
            x_conv, x = block(x)
            skips.append(x_conv)
        x = self.bottleneck(x)
        for block in self.expansive_blocks:
            skip = skips.pop()
            x = block(x, skip)
        return self.final_conv(x)
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
unet = UNet().to(device)
```

```
criterion = nn.MSELoss()
optimizer = optim.Adam(unet.parameters(), lr=1e-3)

num_epochs = 5
for epoch in range(num_epochs):
    unet.train()
    running_loss = 0.0
    for imgs, _ in trainloader:
        imgs = imgs.to(device)
        noisy_imgs = add_gaussian_noise(imgs).to(device)

        outputs = unet(noisy_imgs)
        loss = criterion(outputs, imgs)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(trainloader):.6f}")
```

```
unet.eval()
dataiter = iter(testloader)
imgs, _ = next(dataiter)
imgs = imgs.to(device)
noisy_imgs = add_gaussian_noise(imgs).to(device)

with torch.no_grad():
    denoised = unet(noisy_imgs)

num_show = 6
fig, axes = plt.subplots(3, num_show, figsize=(15,6))
for i in range(num_show):
    axes[0,i].imshow(np.transpose(noisy_imgs[i].cpu().numpy(), (1,2,0)))
    axes[0,i].set_title("Noisy")
    axes[0,i].axis("off")

    axes[1,i].imshow(np.transpose(denoised[i].cpu().numpy(), (1,2,0)))
    axes[1,i].set_title("Denoised")
    axes[1,i].axis("off")

    axes[2,i].imshow(np.transpose(imgs[i].cpu().numpy(), (1,2,0)))
    axes[2,i].set_title("Clean")
    axes[2,i].axis("off")

plt.tight_layout()
plt.show()
```