Filière Informatique

Assignment 2

Groupe D :
Tanguy Pemeja

Encadrant :
Aurélie Bugeau

Version du 15 Mai 2020

1 Question 1

En premier lieu, nous allons générer des images bruitées à partir de la database afin de pouvoir les débruiter plus tard grâce à un CCN.

1.1 Code

Dans cette première question, nous définissons une classe `NoisyBSDSDataset` basée sur la bibliothèque `PyTorch` afin de faire bruiter chacune de nos images grâce à des calculs plus optimisés. Cette classe contient 4 méthodes :

1. `__init__` qui permet de récupérer les images du set
2. `__len__` qui retourne le nombre de fichiers composant ce set
3. `__repr__` qui retourne les caractéristiques de notre classe
4. `__getitem__` qui renvoie par image du set, l'image et sa version bruitée

La fonction `myimshow` quant à elle, permet d'afficher les images à partir des images retourner par la classe.

1.2 Résultat

Afin de générer nos sets de données, nous allons bruiter chacune des images des sets avec :

```
train_set = NoisyBSDSDataset(dataset_root_dir, mode='train')
test_set = NoisyBSDSDataset(dataset_root_dir, mode='test', image_size=(320, 320))
val_set = NoisyBSDSDataset(dataset_root_dir, mode='val', image_size=(320, 320))
```

Nous pouvons observer le résultat obtenu grâce à notre commande `myimshow`, ainsi nous pouvons afficher l'image à l'index 1 du set de test ainsi que ça version bruitée sur l'image 1 ci-dessous.

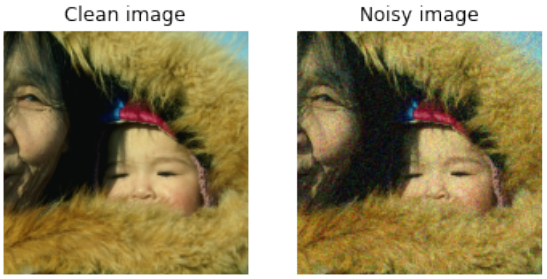

FIGURE 1 – Bruitage d'une image du set de test

2 Question 2

Nous allons maintenant utiliser le fichier `nntools.py` afin de réaliser notre CNN. Mais nous devons d'abord définir une nouvelle classe `NNRegression` afin de définir notre filtre pour les couches de convolutions de nos images ainsi que d'implémenter la méthode `criterion`, notre fonction de coût. Cette fonction est défini par la fonction de perte d'entropie croisée `MSELoss` de la bibliothèque `PyTorch`. Celle-ci est implémentée afin de calculer l'erreur quadratique moyenne entre notre image calculée à partir de l'image bruité et celle d'origine.

3 Question 3

Nous allons ici définir une dernière classe `DnCNN` basé sur la classe `NNRegression` qui sera notre réseau de neurones, et qui implémente la méthode `forward` qui calcule les valeurs de retour en fonction des valeurs d'entrée. Ce calcule s'effectue suivant la méthode de la figure 2 suivante :

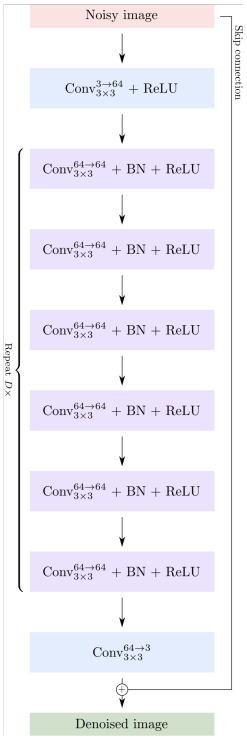

FIGURE 2 – Méthode de calcul de l'image nette

4 Question 4

Dans cette partie, nous allons étendre une classe afin de pouvoir améliorer l'entraînement en définissant des points de sauvegarde et en combinant ces sauvegardes.

4.1 Code

Nous définissons une classe `DenoisingStatsManager` basée sur la classe `StatsManager` de notre fichier `nntools.py` afin de garder en mémoire les informations apprises durant l'époque de l'entraînement. Cette nouvelle redéfinit les méthodes suivantes :

1. `__init__` qui initialise notre classe
2. `init` qui initialise notre restauration des informations
3. `accumulate` qui somme la qualité de la restauration de notre apprentissage avec la formule $PSNR = 10log_{10}\frac{4n}{\|y-n\|_2^2}$ à chaque itération
4. `summarize` qui renvoie le couple formé par la perte accumulé et la qualité de la restauration le tout divisé par le nombre de sauvegarde que nous avons fait pour chacune de nos itérations

4.2 Expérience

Pour réaliser l'entraînement de notre réseau de neurones, nous utilisons la classe `Experiment` de notre fichier `nntools.py`. Pour se faire, nous utilisons la commande :

```
exp1 = nt.Experiment(dncnn, train_set, val_set, adam, stats_manager,
batch_size=B, output_dir="denoising1", perform_validation_during_training=True)
```

Nous fournissons en argument :

1. le réseau de neurones définit par la classe `DnCNN` initialisé avec un $D = 6$ pour le calcul défini figure 2

2. nos sets de données : d'entraînement et de validation
3. un optimisateur de calcul `Adam` défini par la bibliothèque `PyTorch`
4. notre classe de sauvegarde `DenoisingStatsManager`
5. la taille de nos batchs
6. l'emplacement de notre sauvegarde
7. un booléen indiquant si on calcule aussi les statistiques de notre set de validation pour chaque époque

5 Question 5

Dans le dossier `denoising1`, nous pouvons trouver 2 fichiers : `config.txt` et `checkpoint.pth.tar`. Le premier correspond à un résumé des caractéristiques de notre expérience décrite précédemment avec notamment :

1. les caractéristiques du réseau de neurones : la fonction de coût, le découpage de la convolution et des batchs
2. les informations sur nos sets d'image (entraînement et validation)
3. les paramètres de notre optimisateur de calcul

Quant au second fichier, il correspond à une archive contenant les points de sauvegardes décrites précédemment. Il permet de sauvegarder notre expérience sur le disque afin de pouvoir l'utiliser pour ne pas devoir tout refaire à chaque fois.

6 Question 6

Dans cette question, nous allons faire tourner notre expérience sur 30 époques afin de voir comment notre réseau de neurones à débruiter les images.

6.1 Code

Dans un premier temps, nous définissons une fonction d'affichage `plot` afin de pouvoir obtenir un graphe contenant les courbes comparant l'écart ainsi que la qualité de restauration de notre apprentissage entre les sets d'entraînement et de validation en fonction des époques.

6.2 Résultat

Grâce à la commande `run` sur notre expérience, nous obtenons les courbes décrites précédemment sur la figure 3 suivante :

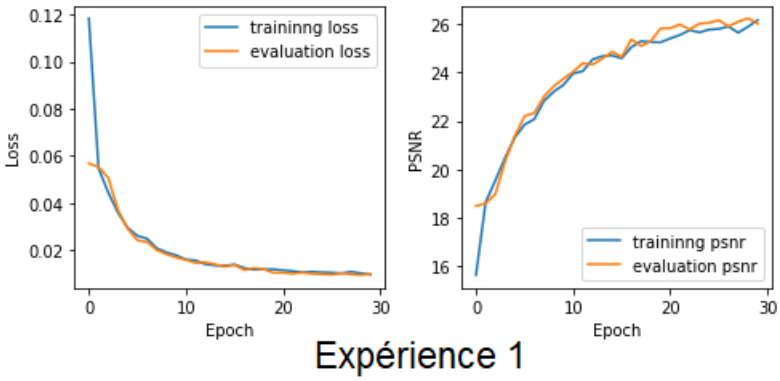

FIGURE 3 – Courbes comparative entraînement / validation

Nous pouvons voir que l'écart ainsi que la qualité de restauration de notre apprentissage suivent la même évolution entre l'entraînement et l'évaluation durant les différentes époques. Celle-ci est décroissante pour l'écart entre l'image de sortie et l'image souhaitée, et croissante pour la restitution au fur et à mesure des époques. Cela montre bien que notre réseau apprend à chaque itération et réduit donc son erreur entre son image de sortie et l'image originale sans bruitage.
Dans un second temps, nous affichons aussi le comparatif 4 d'une image du fichier test d'entrée du réseau de neurones et son image débruité en sorti du réseau.

FIGURE 4 – Comparatif avant/après d'une image du set test dans le réseau de neurones

7 Question 7

Dans cette question, nous analysons visuellement la qualité de notre entraînement.

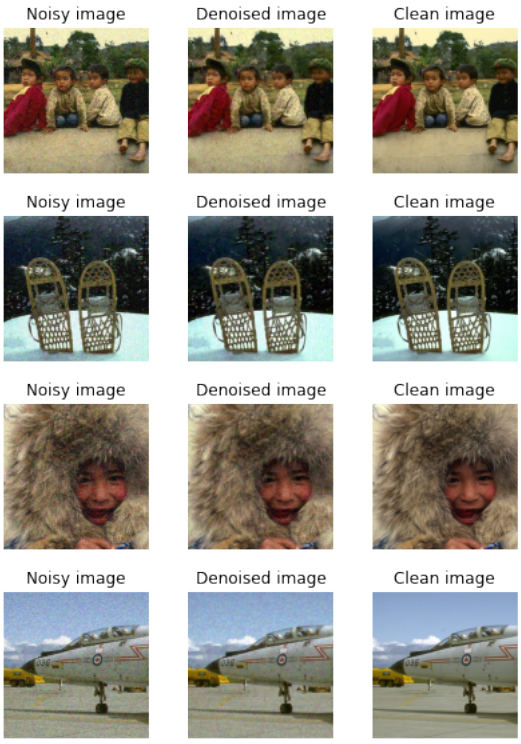

FIGURE 5 – Comparatif du rendu de 4 images du set de test

Sur la figure 5, nous pouvons observer que notre image débruitée en sortie du réseau est plus net que celle en entrée sur les différentes lignes d'images. Toutefois la qualité n'est pas non plus au même niveau que l'image d'origine. On peut notamment observer que la qualité du ciel sur la dernière ligne sur l'image débruité n'est pas aussi nette que sur l'image d'origine.

8 Question 8

Afin de déterminer le nombre de paramètre, nous utilisons les fonctions de la bibliothèque `PyTorch` :

```
for name, param in dncnn.named_parameters():
    print(name,param.size())
```

Les résultats obtenus correspondent aux différentes couches de convolutions du réseau défini dans `DnCNN(D)` avec :

```
(1)     self.conv.append(nn.Conv2d(3, C, 3, padding=1))
            for k in range(D):
(2)           self.conv.append(nn.Conv2d(C, C, 3, padding=1))
(3)     self.conv.append(nn.Conv2d(C, 3, 3, padding=1))
```

Grâce à cela, on peut déterminer le nombre de valeur de poids qu'il y a sur chacune des couches de convolution. Ainsi lorsqu'on ne spécifie pas la valeur de C et qu'on laisse l'initialisation à 64, nous obtenons les nombres de paramètres suivants :

$$\begin{cases} (1) \rightarrow 64 * 3 * 3 * 3 = 1728 \\ (2) \rightarrow 64 * 64 * 3 * 3 = 36864 \\ (3) \rightarrow 3 * 64 * 3 * 3 = 1728 \end{cases}$$

Ainsi, nous obtenons le nombre de paramètre : $3456 + 36864 * D$. Et donc pour une valeur de D=6, nous avons 224640 paramètres pour définir les poids de notre réseau.
Sachant qu'à chaque couche de convolution la taille du carré nécessaire pour calculer le pixel final augmente de 2 et étant donné que nous avons 2 couches de convolutions, le nombre de pixels nécessaires pour calculer 1 pixel avec D couches vaut : $(1 + 2 * (D + 2))^2$. Ainsi pour que $33 * 33 = (1 + 2 * (D + 2)) * (1 + 2 * (D + 2))$, D vaut 16. Et si D=16, alors nous avons besoin de $3456 + 36864 * 14 = 519552$ paramètres.

9 Question 9

Nous allons ici utiliser une nouvelle classe `UDnCNN` de la même manière que la classe `DnCNN` qui sera notre nouveau réseau de neurones, et qui ré-implémente la méthode `forward`, qui calcule les valeurs de retour en fonction des valeurs d'entrée. Ce calcule s'effectue suivant la nouvelle méthode de la figure 6 suivante :

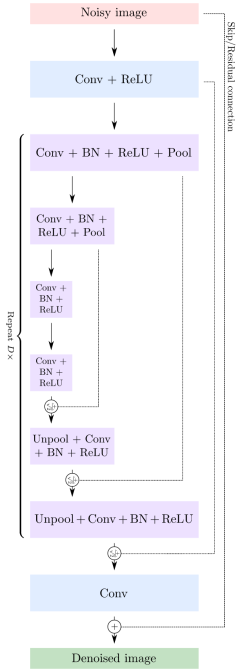

FIGURE 6 – Nouvelle méthode de calcul de l'image nette

10 Question 10

Dans cette question, nous allons faire tourner notre nouvelle expérience sur 30 époques afin de voir comment notre nouveau réseau de neurones a débruité les images, et pouvoir comparer avec l'expérience précédente.

10.1 Code

Nous définissons notre nouvelle expérience avec les mêmes caractéristiques que la précédente avec seulement le calcul des valeurs qui diffère. On définit de la même manière un répertoire de sauvegarde afin de pouvoir avoir un point de récupération si jamais on veut de nouveau utiliser le réseau.

10.2 Résultat

Avec la commande `run` sur notre nouvelle expérience, nous avons de nouvelles courbes afin de les comparer avec les précédentes.

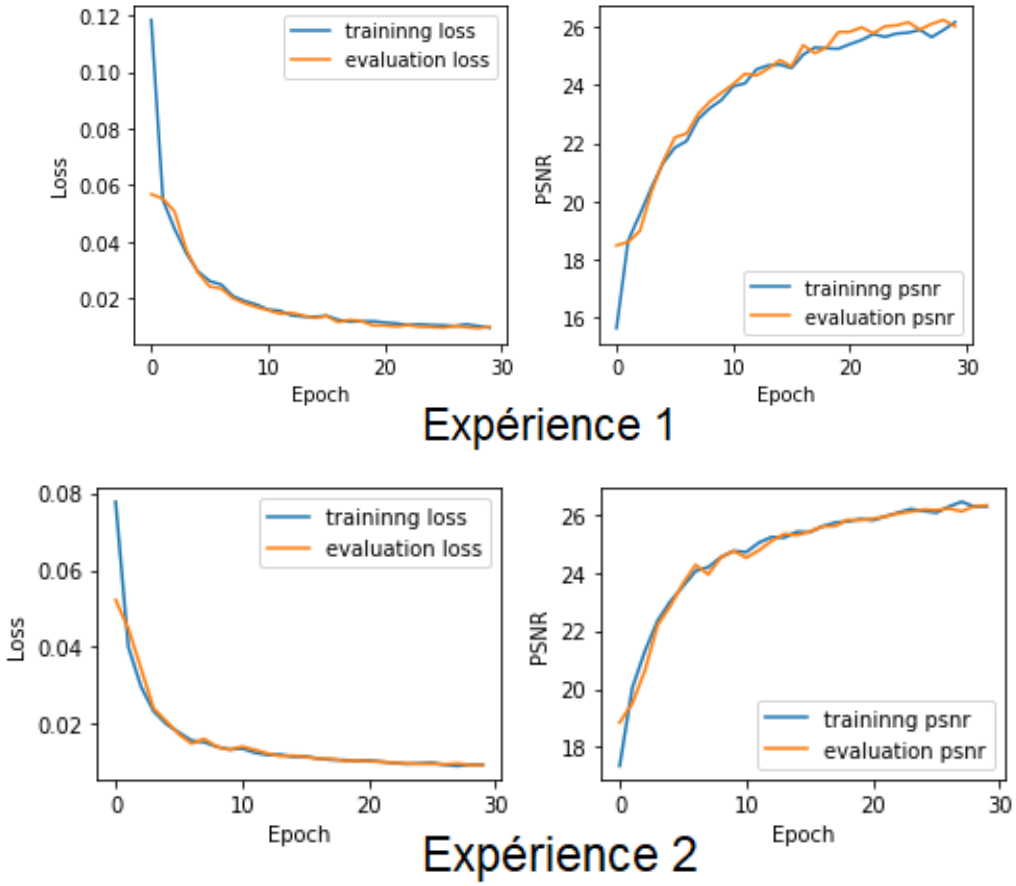

FIGURE 7 – Comparaison entre les 2 expériences

D'après nos résultats ci-dessus (Figure 7), nous pouvons observer que les 2 expériences sont très proches. Les courbes suivent les mêmes trajectoires et convergent vers les mêmes valeurs. On peut noter que les seules différences sont au début, pour les premières époques. En effet le début de l'expérience 2 obtient de meilleur résultat que le début de l'expérience 1.

Cette proximité des 2 expériences se retrouve dans leurs résultats qui sont indistinguables, en prouve les images retournées par nos expériences sur la figure 8.

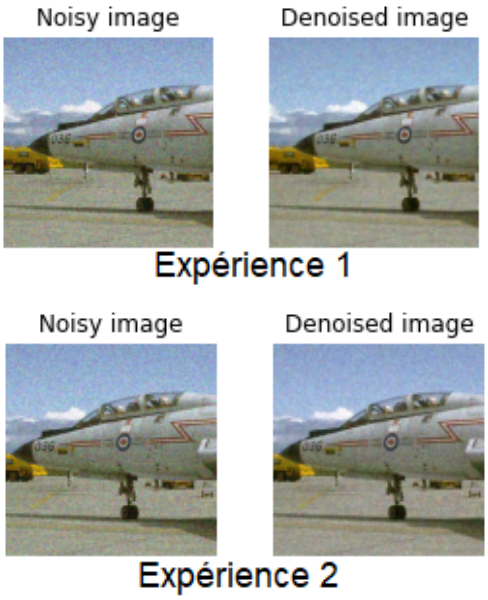

FIGURE 8 – Comparaison entre les images retournées par nos expériences

11 Question 11

Notre nouveau réseau de neurones est composé des mêmes couches de convolutions, ainsi de la même manière le nombre de paramètre est de $3456 + 36864 * D$. Et donc pour D=6, nous avons 224640 paramètres. Mais contrairement au réseau précédent, notre calcul utilise `F.max_pool2d` et `F.max_unpool2d` de la bibliothèque `PyTorch`. Ces fonctions, double et divise par deux le nombre de pixels nécessaires en plus. Ainsi suivant l'étape où on est le nombre varie de la manière suivante :

1. for k in range(int(D/2)-1) : le nombre de pixel nécessaire en plus double
2. pour k = int(D/2) - 1 et k = int(D/2), le nombre de pixel en plus reste le même
3. for k in range(int(D/2)+1, D) : le nombre de pixel en plus est divisé par 2

Au final, on obtient l'équation pour le nombre de pixels nécessaires pour 1 pixel pour une profondeur de D suivante :

$$nb\_pixel = (\sum_{k=0}^{int(D/2)-1} 2^{k+1} + 2 * 2^{int(D/2)} + \sum_{k=int(D/2)+1}^{D} 2^{int(D/2)-(k-int(D/2))})^2$$

$$nb\_pixel = (\sum_{k=1}^{int(D/2)} 2^{k} + 2 * 2^{int(D/2)} + \sum_{k=1}^{int(D/2)-1} 2^{int(D/2)-k})^2$$

$$nb\_pixel = (\sum_{k=1}^{int(D/2)} 2^{k} + 2 * 2^{int(D/2)} + \sum_{k=1}^{int(D/2)-1} 2^{k})^2$$

$$nb\_pixel = (2 * \sum_{k=1}^{int(D/2)-1} 2^{k} + 3 * 2^{int(D/2)})^2$$

Toutefois il ne faut pas oublier les couches de convolutions au début et à la fin qui augmentent ce nombre de 2 ainsi que le pixel en lui-même. Nous obtenons donc le nombre :
$receptive\_field = (1 + 2 + 2 * \sum_{k=1}^{int(D/2)-1} 2^{k} + 3 * 2^{int(D/2)})^2$ Pour notre réseau où D=6, nous avons donc $receptive\_field = (1 + 2 + 2 * \sum_{k=1}^{2} 2^{k} + 3 * 2^{3})^2 = (1 + 2 + 2 * (2 + 4) + 3 * 8)^2 = 39^2$