GROUPE INP La Prépa

Promo P31 - 2ème année
Année Universitaire 2024-2025

Compte-rendu du Projet Réseaux de Neurones

Noreddine BELMEGUENAÏ
Adrien CANDELA
Rémy GAUDIN

Avril 2025

Table des matières

1 Réseau dense général

Dans cette première partie, nous avons cherché à créer un réseau de neurones avec p couches intermédiaires. En faisant varier la dimension du réseau, on peut étudier l'impact de la taille du réseau, du nombre de neurones de chaque couche sur la capacité d'apprentissage de notre réseau de neurones. On procède par descente de gradient pour trouver l'erreur minimale.

1.0.1 Question 1

Étant donnée une liste `dimensions`, implémenter le réseau de neurones dense associé.

cf. ficher de code fournit

1.0.2 Question 2

Déterminer le nombre de paramètres présents dans chacun des réseaux donnés par les dimensions suivantes, c'est-à-dire le nombre de scalaires présents dans toutes les matrices de W (on négligera les biais b).

☞ **Réseau n°1**

$$\texttt{[2,3,3,3,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,3) \\ W_2 \text{ taille } (3,3) \\ W_3 \text{ taille } (3,3) \\ W_4 \text{ taille } (3,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 6 \text{ scalaires} \\ W_2 = 9 \text{ scalaires} \\ W_3 = 9 \text{ scalaires} \\ W_4 = 3 \text{ scalaires} \end{cases} \Longrightarrow 27 \text{ paramètres}$$

☞ **Réseau n°2**

$$\texttt{[2,7,7,7,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,7) \\ W_2 \text{ taille } (7,7) \\ W_3 \text{ taille } (7,7) \\ W_4 \text{ taille } (7,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 14 \text{ scalaires} \\ W_2 = 49 \text{ scalaires} \\ W_3 = 49 \text{ scalaires} \\ W_4 = 7 \text{ scalaires} \end{cases} \Longrightarrow 119 \text{ paramètres}$$

☞ **Réseau n°3**

$$\texttt{[2,15,15,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,15) \\ W_2 \text{ taille } (15,15) \\ W_3 \text{ taille } (15,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 30 \text{ scalaires} \\ W_2 = 225 \text{ scalaires} \\ W_3 = 15 \text{ scalaires} \end{cases} \Longrightarrow 270 \text{ paramètres}$$

☞ **Réseau n°4**

$$\texttt{[2,3,15,15,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,3) \\ W_2 \text{ taille } (3,15) \\ W_3 \text{ taille } (15,15) \\ W_4 \text{ taille } (15,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 6 \text{ scalaires} \\ W_2 = 45 \text{ scalaires} \\ W_3 = 225 \text{ scalaires} \\ W_4 = 15 \text{ scalaires} \end{cases} \Longrightarrow 291 \text{ paramètres}$$

☞ **Réseau n°5**

$$\texttt{[2,15,15,3,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,15) \\ W_2 \text{ taille } (15,15) \\ W_3 \text{ taille } (15,3) \\ W_4 \text{ taille } (3,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 30 \text{ scalaires} \\ W_2 = 225 \text{ scalaires} \\ W_3 = 45 \text{ scalaires} \\ W_4 = 3 \text{ scalaires} \end{cases} \Longrightarrow 303 \text{ paramètres}$$

☞ **Réseau n°6**

$$\texttt{[2,40,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,40) \\ W_2 \text{ taille } (40,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 80 \text{ scalaires} \\ W_2 = 40 \text{ scalaires} \end{cases} \Longrightarrow 120 \text{ paramètres}$$

☞ **Réseau n°7**

$$\texttt{[2,20,20,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,20) \\ W_2 \text{ taille } (20,20) \\ W_3 \text{ taille } (20,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 40 \text{ scalaires} \\ W_2 = 400 \text{ scalaires} \\ W_3 = 20 \text{ scalaires} \end{cases} \Longrightarrow 460 \text{ paramètres}$$

☞ **Réseau n°8**

$$\texttt{[2,5,4,4,4,4,1]} \Longrightarrow \begin{cases} W_1 \text{ taille } (2,5) \\ W_2 \text{ taille } (5,4) \\ W_3 \text{ taille } (4,4) \\ W_4 \text{ taille } (4,4) \\ W_5 \text{ taille } (4,4) \\ W_6 \text{ taille } (4,1) \end{cases} \Longrightarrow \begin{cases} W_1 = 10 \text{ scalaires} \\ W_2 = 20 \text{ scalaires} \\ W_3 = 16 \text{ scalaires} \\ W_4 = 16 \text{ scalaires} \\ W_5 = 16 \text{ scalaires} \\ W_6 = 4 \text{ scalaires} \end{cases} \Longrightarrow 291 \text{ paramètres}$$

1.0.3 Question 3

Construire les réseaux précédents et les entraîner sur les données `(X_train, T_train)`.

cf. ficher de code fournit

1.0.4 Question 4

Commenter les résultats obtenus. On pourra notamment regarder :

- L'évolution de l'erreur d'entropie.
- Le taux de précision maximal obtenu sur les données de test.
- La difficultée à obtenir un pas de descente faisant converger le réseau.
- L'influence de la répartition de dimensions (par exemple `[2,5,8,1]` ou `[2,8,5,1]`).

Proposer des explications aux résultats obtenus.

Le pas de descente `lr` doit être optimisé pour que le réseau converge le plus vite possible pour sans que jamais il ne diverge. S'il est trop petit, le réseau prendra trop de temps et d'itérations pour converger, il se bloquera même parfois dans des points fixes nécessitant une variation plus élevée pour être dépassée. Si `lr` est trop grand, le réseau ne convergera pas car il manquera de précision : dans l'exemple ci-contre, un `lr` trop grand oscillerais sur les parois. (voir FIGURE 1.1)

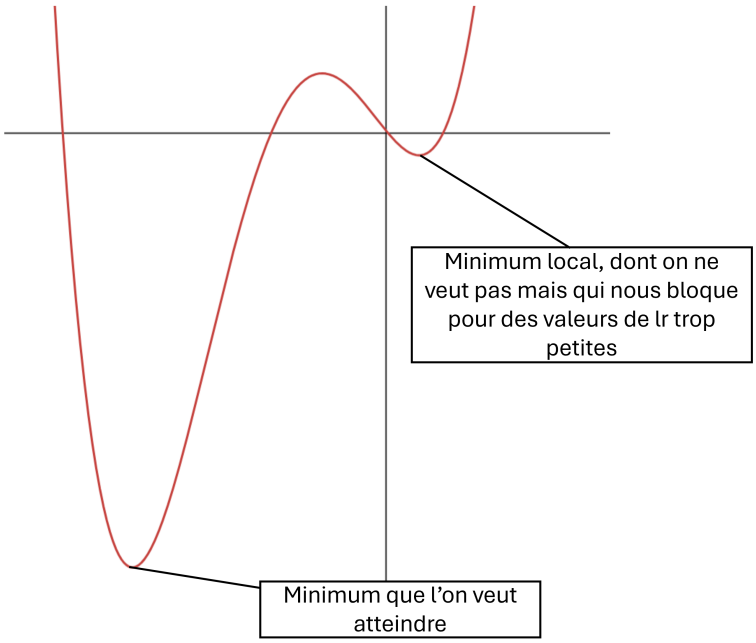

Figure 1.1 – Exemple de minimum local bloquant la convergence

C'est pourquoi trouver le bon pas de descente est très important car c'est une condition nécéssaire à la convergence du réseau. Pour chaque réseaux (variant suivant ses dimensions), nous avons déterminé le `lr` donnant les meilleurs résultats.

`Dimensions`	Meilleur `lr`	Meilleur résultat (sur 10 000 itérations)	Nombre de scalaire dans la dimension
`[2, 3, 3, 3, 1]`	$0,0015$	$80,35$	27
`[2, 7, 7, 7, 1]`	$0,00075$	$90,7$	119
`[2, 15, 15, 1]`	$0,00075$	$93,05$	270
`[2, 3, 15, 15, 1]`	$0,00075$	$81,25$	279
`[2, 15, 15, 3, 1]`	$0,00060$	$97,9$	303
`[2, 40, 1]`	$0,000075$	$77,1$	120
`[2, 20, 20, 1]`	$0,00075$	$90,15$	460
`[2, 5, 4, 4, 4, 4, 1]`	$0,001$	$82,2$	82

Figure 1.2 – Meilleur `lr` pour chaque réseaux

On a pensé à dynamiser le pas de descente, c'est-à-dire le modifier pendant les itérations d'entraînements, en fonction des écarts entre les dernières erreurs d'entropie. Nous avons remarqué que l'erreur diminue de moins en moins vite et nous avons voulu diminuer `lr` pour augmenter la précision du programme et lui permettre de repartir.

```
if abs(suite_erreur[-2] - suite_erreur[-1]) < 0.075:
    print('###')
    lr *= 2/3
```

Figure 1.3 – Ici on compare si la différence des deux dernières erreurs est trop petite, et si c'est le cas on diminue `lr`. On affiche dans la console si on modifie `lr` avec le `print`

Après 20-25 tests de cette méthode, on a pu remarquer que l'on obtenait des meilleurs résultats pour les réseaux qui fonctionnaient déjà très bien (on à gagné 5% de taux de précision en moyenne), mais que pour les réseaux moins efficaces, cette méthode ne changeait pas grand chose. On a aussi remarqué que modifier lr de beaucoup (*2/3 au lieu de par exemple *0.995) était plus efficace car la seconde méthode demandait généralement de modifier lr plusieurs fois d'affilée.

Après avoir étudié l'influence du pas de descente sur la convergence du réseau, nous avons voulu mesurer l'impact du choix du nombre de couches et de leurs tailles respectives, et de l'ordre qu'on leur donnait.

On remarque d'abord que les réseaux avec des W de grande taille ne sont pas les meilleurs convergents. Par exemple `[2,20,20,1]` converge moyennement alors que `[2,15,15,3,1]` converge très bien. Un réseau avec des grosses sous-couches comme `[2,20,20,1]` ne converge pas bien car la mise à jour de ses couches avec `updateWb` n'est pas efficace, les paramètres de départs étant bien trop aléatoires.

Le nombre de sous-couches ne semble pas être déterminant non plus car `[2,5,4,4,4,4,1]` n'est pas vraiment bon non plus. Encore une fois, c'est la mise à jour de W et de b qui n'est pas pertinente car chaque sous-couche va être modifiée de manière trop différente les unes des autres : les sous-couches ne sont, en quelque sorte, pas assez synchronisées entre-elles.

On remarque aussi que `[2,15,15,3,1]` et `[2,3,15,15,1]` n'ont pas du tout les mêmes résultats. On a essayé sur d'autres réseaux avec des similarités plus marquantes : `[2,8,6,4,1]` et `[2,4,6,8,1]`. Le premier des deux converge beaucoup mieux que le second (moyenne du taux de précision à 90% pour le premier contre 82% pour le second). On en a déduit que la forme de réseau importe beaucoup et qu'un réseau avec des couches de plus en plus petites permet d'affiner les résultats quand nous sommes proches de la prédiction finale `Y`.

Finalement, avec notre meilleur réseau (`[2,15,15,3,1]`) on obtient ceci (Figure 1.4 et Figure 1.5) :

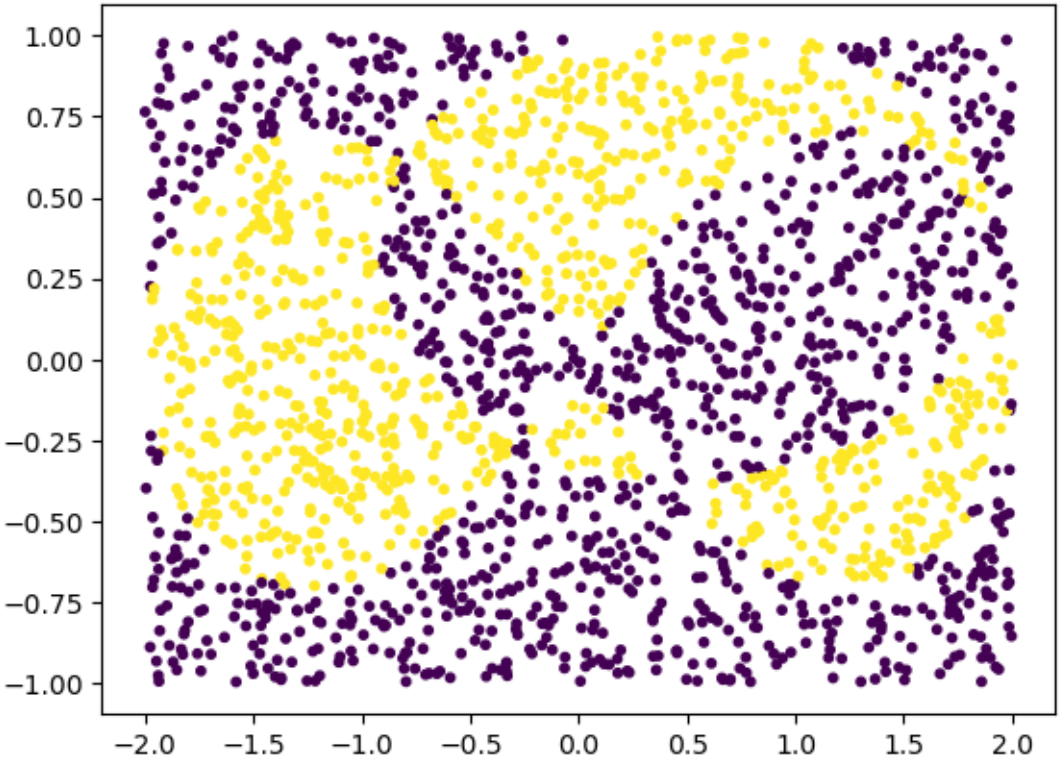

FIGURE 1.4 – Avec un taux de précision de 99%...

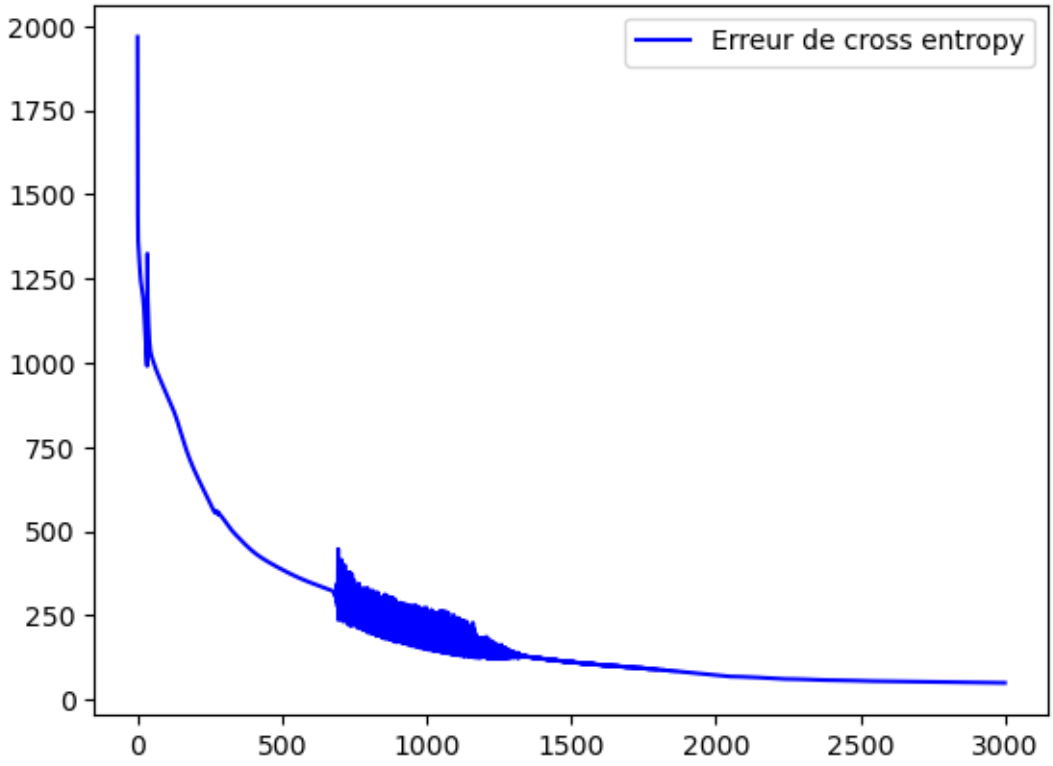

FIGURE 1.5 – Erreur d'enrtopie en fonction du nombre d'itérations

2 Problème de classification à $K \geq 3$ classes

2.1 Adaptation de la régression logistique

2.1.1 Question 1

Expliquer pourquoi minimiser la fonction J permet d'avoir un réseau qui réalise une bonne classification des points de données.

On utilise la fonction d'erreur d'entropie qui donne :

$$J = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk}\ln(y_{nk})$$

t_{nk} vaut 1 si le point n est de classe k, et y_{nk} correspond à la probabilité que le point n soit dans la classe k d'après la prédiction.

- Si la prédiction est bonne, y_{nk} est proche de 1 pour k tel que $t_{nk} = 1$ et faible pour les autres k, donc $t_{nk}\ln(y_{nk}) \approx 0$.
- Si la prédiction n'est pas bonne, y_{nk} est faible pour k tel que $t_{nk} = 1$ donc $t_{nk}\ln(y_{nk}) < 0$ donc cela ajoute une quantité négative à J, qui révèle bien une erreur de prédiction. La fonction ln diverge rapidement vers $-\infty$ en 0 donc la quantité $\ln(y_{nk})$ est importante. Ainsi la fonction d'erreur d'entropie J devient rapidement importante si il y a des erreurs de prédictions, et plus elle est faible, plus les points sont correctement classés.

2.1.2 Question 2, 3 et 4

cf. fichier de code fournit

2.1.3 Question 5

Expliquer comment obtenir la matrice C de prédiction de classes à partir de la matrice Y. Écrire une fonction `predit_classe(Y)` qui renvoi la matrice C.

La matrice Y est de dimension (N, K) avec $\forall (n,k) \in [\![1,N]\!] \times [\![1,K]\!], Y[n,k]$ la probabilité du point n d'appartenir à la classe k. Ainsi, on souhaite transformer cette matrice en une matrice C de dimension $(N,)$ avec $\forall n \in [\![1,N]\!], C[n]$ le numéro de la classe du point n. De cette manière, la matrice C permet ensuite d'associer, pour chacun des points, la couleur (arbitraire) correspondant à sa classe. On procède par itération sur chacune des lignes de Y :

1. $\forall n \in [\![1,N]\!]$, On appel la fonction de numpy `np.argmax(Y[n])` qui donne directement l'indice k de la colonne dans laquelle la valeur (qui correspond à la probabilité d'appartenir à la classe k) est maximale. Ainsi `np.argmax(Y[n])` donne la classe à laquelle le point n "à le plus de chance d'appartenir".
2. On associe directement cette classe au point n (`C[n]=np.argmax(Y[n])`).

2.1.4 Question 6

Appliquer l'algorithme de régression logistique au jeu de données `probleme_4_classes`. Commenter les résultats obtenus.

Pour le problème à 4 classes, on obtient :

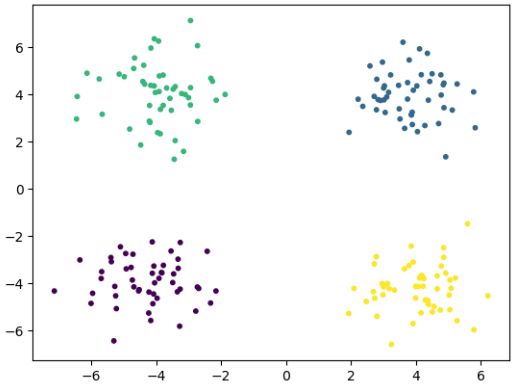

Figure 2.1 – Régression logistique appliquée au jeu de données `probleme_4_classes`.

On voit que les points sont correctement classées, le taux de précision affiche 100%. De plus l'algorithme parvient à traiter parfaitement le problème avec seulement des dizaines d'itérations. L'erreur d'entropie passe en 10 itérations de J 10^3 à $J < 10^{-2}$. On en conclut que les points étant déjà bien séparés dans l'espace, et le problème étant résolvable avec des droites de séparations simples, l'algorithme n'a aucun problème à classer les points.

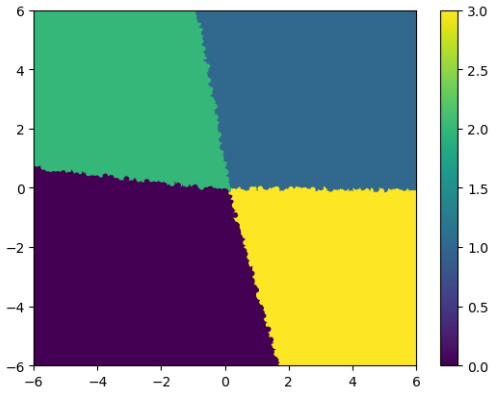

Figure 2.2 – Fonction `repartition_proba` appliquée au jeu de données `probleme_4_classes`.

Voici le résultat de notre fonction `repartition_proba` qui donne un aperçu de la séparation des zones attribuées à chaque classe. Nous voyons bien que la classification n'est pas un motif complexe, et peut être gérée avec des droites (problème linéaire) ce qui explique la facilité et la rapidité de l'algorithme de régression logistique.

2.1.5 Question 7 et 8

- Appliquer l'algorithme de régression logistique au jeu de données `probleme_5_classes`. Commenter les résultats obtenus.
- Pour le problème à 5 classes, tracer sur un même graphique les droites de séparations de chaque classe et les nuages de points de données. Commenter le résultat obtenu.

Pour le problème à 5 classes, la diminution de l'erreur d'entropie n'est pas aussi drastique, mais le programme reste très efficace. On obtient la bonne classification :

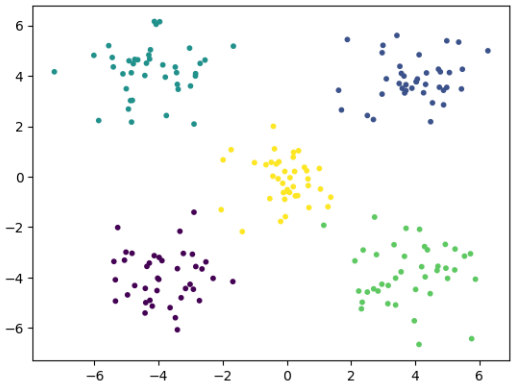

Figure 2.3 – Régression logistique appliquée au jeu de données `probleme_5_classes`

Pour cette fois-ci, l'erreur d'entropie passe de J 10^3 à $J < 10^{-2}$ en un minimum de 50 itérations. Cela s'explique par le fait que les données à classifier restent relativement basiques : les points sont déjà organisées par paquets que l'on peut séparer par des droites. Si on prend notre fonction `repartition_proba` et qu'on l'applique à notre jeu de données, on obtient cette fois-ci :

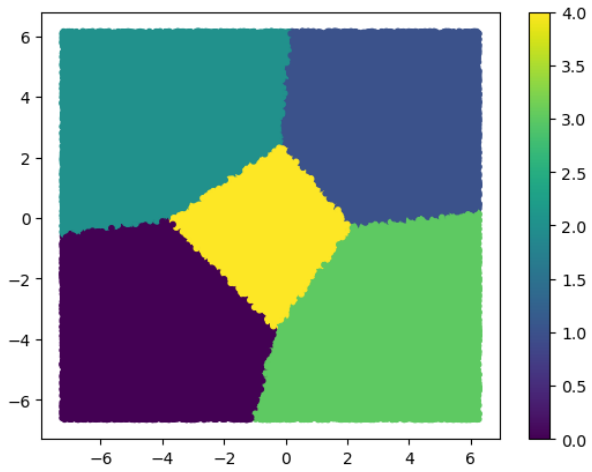

Figure 2.4 – Fonction `repartition_proba` appliquée au jeu de données `probleme_5_classes`.

Ceci illustre bien la résolution légèrement plus complexe que le problème à 4 classes mais tout de même facile pour l'algorithme les points étant déjà bien séparés dans l'espace. On peut

également représenter la droite de séparation entre deux classes, qui correspond à l'ensemble des points pour lesquels la probabilité d'être dans une classe ou l'autre est égale.

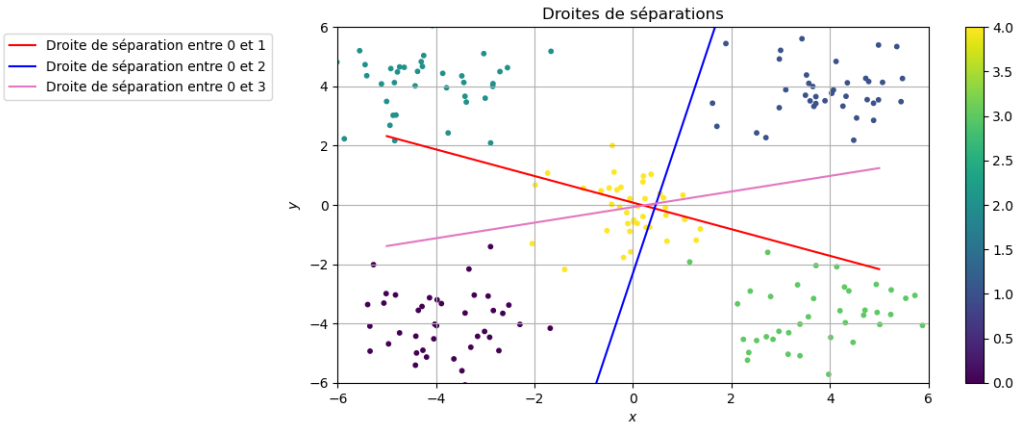

Figure 2.5 – Droites de séparations entre les différentes classes

On le voit bien, ce jeu de données ne s'éloigne pas trop de la linéarité, ce qui le rend plus facile à traiter de manière efficace.

Remarque

La fonction softmax manipule des valeurs potentiellement très grandes a cause de l'exponentielle, ce qui présente un risque d'Overflow. Pour résoudre ce problème, on ne calcule pas

$$Softmax(A)$$

mais

$$Softmax(A - \texttt{np.max(A)})$$

c'est à dire en retranchant d'abord à toutes les valeurs de A le max de A. De cette manière, la valeur maximale dans A - `np.max(A)` est 0, et donc lorsque l'on calcule l'exponentielle terme à terme, les valeurs sont toutes inferieures ou égales à 1.
Pourquoi peut-on faire ca ?

$$\text{Softmax}(A_i) = \frac{e^{A_i}}{\sum_j e^{A_j}}$$

$$\text{Softmax}(A_i - \max(A)) = \frac{e^{A_i - \max(A)}}{\sum_j e^{A_j - \max(A)}}$$

$$= \frac{e^{A_i} e^{-\max(A)}}{\sum_j e^{A_j} e^{-\max(A)}}$$

Or $e^{-\max(A)}$ ne dépend pas de j donc :

$$= \frac{e^{-\max(A)} e^{A_i}}{e^{-\max(A)} \sum_j e^{A_j}}$$

$$= \frac{e^{A_i}}{\sum_j e^{A_j}} = \text{Softmax}(A_i)$$

2.2 Adapatation des réseaux de neurones denses

2.2.1 Question 1

Visualiser le jeu de données `probleme_6_classes`.

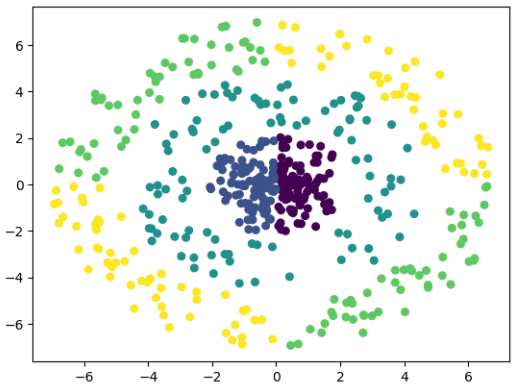

Figure 2.6 – Jeu de données `probleme_6_classes`.

Rien qu'en visualisant le jeu de donnée, on voit que le motif de répartition des classes est beaucoup plus complexe, et ne peut pas se résoudre de manière linéaire (avec des droites). On construit donc un réseau dense pour gagner en non-linéarité.

2.2.2 Question 2

Construire un réseau de neurones dense qui classifie da manière satisfaisante ce problème.

Nous avons commencé par créer un réseau n'utilisant que *softmax* comme fonction d'activation dans toutes les couches, pour essayer. Cela correspond à notre première version de `predit_proba`, tandis que la deuxième version utilise *sigma* comme fonction d'activation des couches cachées du réseau et *softmax* uniquement pour la prédiction. Nous verrons par la suite quelle version est la plus adaptée et pourquoi.

Dimensions du réseau : Choisir de bonnes dimensions est primordial pour l'efficacité du réseau dans la tache attribuée. On peut choisir :

1. Un réseau large (beaucoup de neurones par couche)
2. Un réseau profond (beaucoup de couches)

En théorie, un réseau profond permet de traiter des problèmes plus complexes, en particulier ceux qui sont peu linéaires. En effet, chaque couche applique une fonction d'activation non linéaire, ce qui ajoute un degré supplémentaire de non-linéarité à chaque ajout de couche.
Cependant, en testant l'éfficacité du réseau pour quelques dimensions, par exemple `[2,50,5]`, `[2,40,30,20,5]` et `[2,40,30,30,20,5]`, on obtient les résultats suivants :

`Dimensions`	`[2,50,5]`	`[2,40,30,20,5]`	`[2,40,30,30,20,5]`
Taux de précision (en %)	$85-95$	$70-80$	$60-70$

Figure 2.7 – Taux de précision (en %) pour différents réseaux

Ce sont des résultats approximatifs et empiriques mais la tendance générale est que au-delà de 2 couches le réseau est moins performant. C'est étonnant, mais on peut l'expliquer sans doute par la nature de la fonction Softmax :

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \quad i = 1, 2, \ldots, n$$

Ainsi pour un x_i grand, son image sera proche de 1 tandis que pour un x_i petit, son image sera proche de 0. Certaines classes peuvent donc prédominer sur les autres et cela peut conduire à des gradients de plus en plus petit. Ainsi, lors de l'étape d'optimisation en rétropropagation, les poids évoluent lentement voir pas du tout, ce qui peut empecher de traiter correctement les données. De cette facon, plus le réseau est profond, plus les gradients deviennent faibles ce qui ralentit l'apprentissage, certains neurones n'ayant presque aucun effet. Ce phénomène intervient aussi avec la version 2 qui utilise *sigma*.

- Analyse et optimisation
Il semble donc que dans notre cas, un réseau large soit à privilégier.

— ★ **Version 1 :** *Softmax*

En faisant de nombreux essais, on a trouvé les dimensions `[2,43,5]` étant les plus efficaces, et nous sommes parvenus à atteindre un taux de précision de 92,2 % et l'affichage suivant :

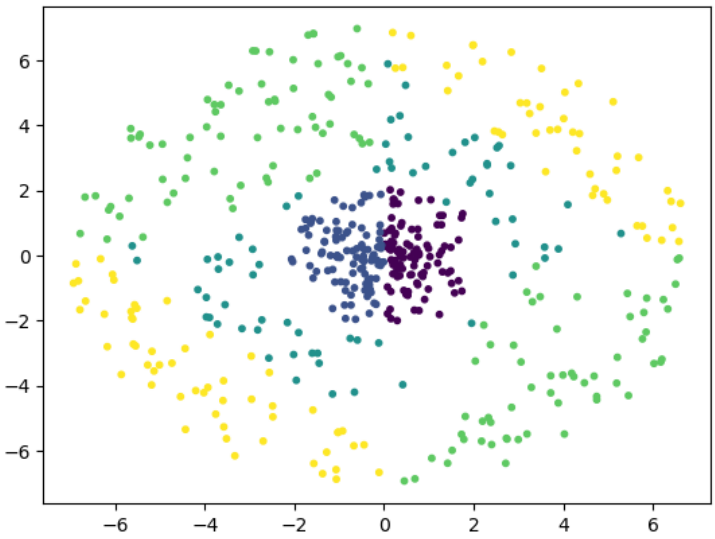

Figure 2.8 – Prédiction du réseau dense `[2,43,5]` avec la **Version 1**.

La prédiction est correcte, seulement quelques points sont mal placés. Cependant l'algorithme tel qu'il est n'est pas parfait, et ne parvient certainement pas à atteindre une précision de 100 % malgré de nombreuses itérations. L'erreur d'entropie diminue de manière stable mais ne descend pas dans des valeurs très faibles. Voici un exemple de l'évolution de la perte avec la **Version 1** :

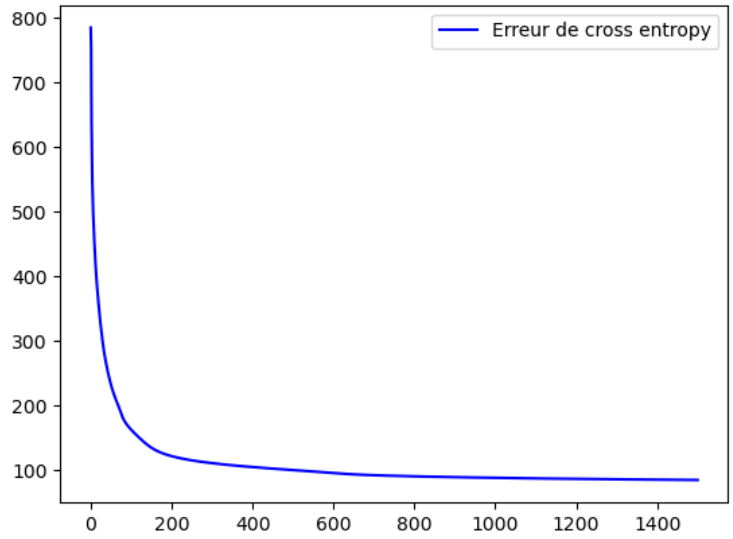

Figure 2.9 – Evolution de l'erreur d'entropie en fonction du nombre d'itérations.

— ★ **Version 2 :** *Softmax + sigma*

On observe que cette version est très efficace avec des dimensions de réseau large, et atteint 100 % de précision très fréquemment et rapidement. Cette fois ci, l'erreur d'entropie fluctue fortement mais atteint très rapidement des valeurs faibles, voire quasiment nulles. Voici un exemple de l'évolution de la perte avec la **Version 2** :

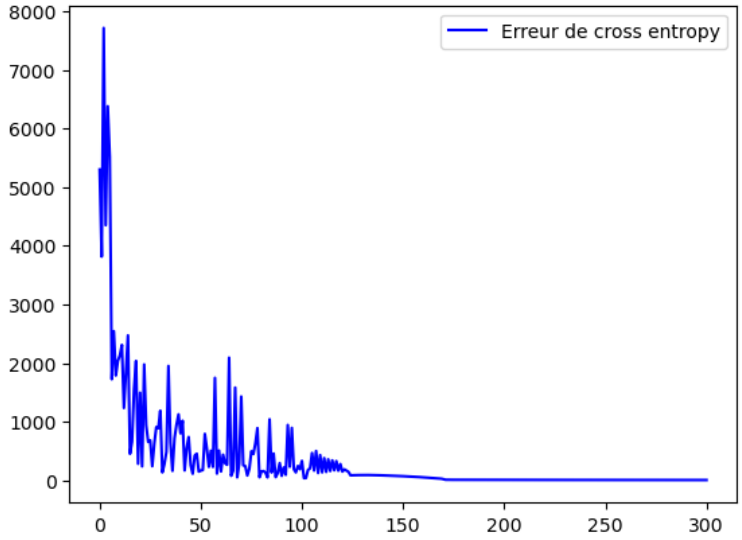

Figure 2.10 – Evolution de l'erreur d'entropie en fonction du nombre d'itérations.

La **Version 2** est donc bien plus efficace que la **Version 1**. Cela peut s'expliquer par le fait que la fonction *sigma* contrairement à *softmax* renvoie des probabilités indépendantes pour chaque neurone là ou softmax renvoie des probabilités telles que leur somme fasse 1. Les couches cachées, avec *sigma*, ne sont pas soumises à la contrainte de normalisation de la somme des activations, ce qui leur permet de mieux modéliser des relations complexes dans les données. En affichant la séparation des zones avec notre fonction `repartition_proba`, on voit bien que la classification est beaucoup plus complexe que précedemment :

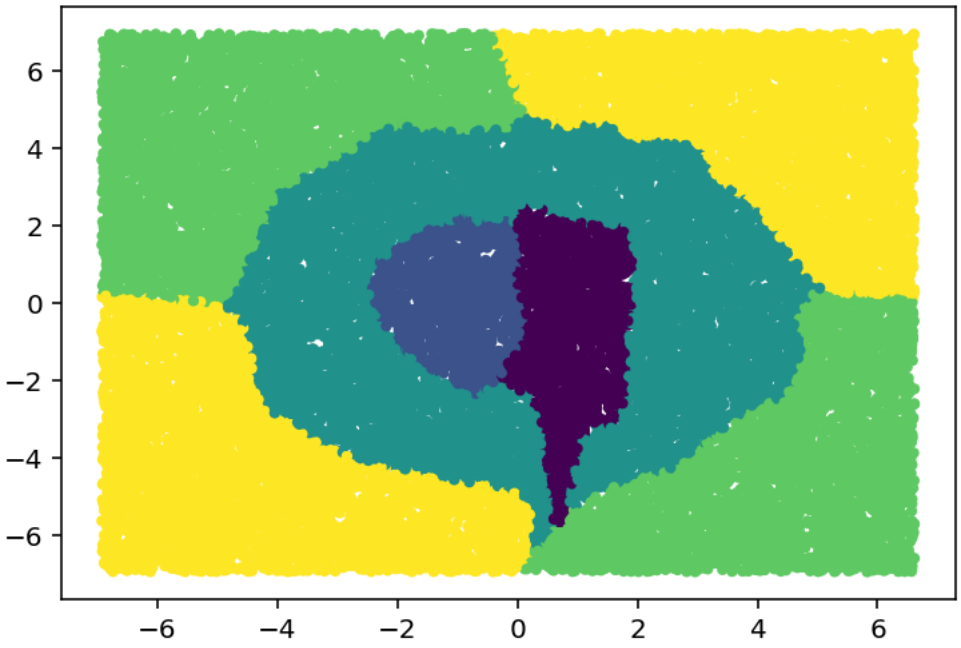

Figure 2.11 – Fonction `repartition_proba` appliquée au données `probleme_6_classes`.

.

3 Partie 3

Dans cette partie, nous explorons différentes techniques permettant de transformer et d'analyser des images. On utilise d'abord la méthode de *pooling* sur une image pour en extraire les informations pertinentes tout en réduisant la qualité de l'image, ce qui permet de limiter la complexité des données et la quantité de calculs nécessaires pour manipuler l'image par la suite. On applique ensuite des filtres à cette image par le biais de la *convolution* entre l'image et un filtre donné, lui attribuant certains effets que nous verrons dans la suite.

3.1 Exercice 1 (Pooling)

3.1.1 Question 1, 2 et 3

- Implémenter une fonction `pooling_max(X,ratio_x,ratio_y)` qui prend en argument une matrice X de taille (D_x, D_y) et des ratios (r_x, r_y) et qui renvoie la matrice Y de taille $\left(\frac{D_x}{r_x}, \frac{D_y}{r_y}\right)$ définie par le *pooling max*.
- Implémenter des fonctions `pooling_moy` et `pooling_median`.
- Appliquer ces fonctions à l'image X de telle manière à obtenir des images X_{max}, X_{moy} et X_{median} de taille $(120, 107)$. Dans la suite de l'énoncé, on appliquera nos fonctions à ces nouvelles images, car elles seraient beaucoup plus longues sur l'image originaal X.

cf. fichier de code fournit

3.2 Exercice 2 (Convolution et traitement d'images)

3.2.1 Question 1 et 2

- Écrire une fonction `convolution1D(X,F)` qui réalise la convolution $1D$ pour la donnée X par le filtre F, c'est-à-dire qui renvoie la liste Z de taille $(N - H + 1)$ définie en (1).
- Implémenter une fonction `cross_correlation1D(X,F)` qui construit la liste Z de taille $(N - H + 1)$ définie en (2) par :

$$\forall i \in [\![0, N - H]\!], z_i = \sum_{h=0}^{H-1} (x_{i+1}) \times (f_h) \tag{2}$$

cf. fichier de code fournit

3.2.2 Question 3

La tester sur les filtres données dans le fichier `.py` et expliquer le rôle des différents filtres F proposés.

➳ **Filtre 1 :**
La première image, hormis sa taille, reste inchangée. Sur la 2ème image, les valeurs ont été amplifiées : les valeurs faibles encore plus que les valeurs élevées. Sur la 3ème image, on peut voir le même principe : multiplication par un facteur 8 pour 10 et par 3,5 pour 80. Après avoir essayé, on voit aussi que le facteur descend même à 2 si, par exemple, on mettait une des valeurs à 200. Ainsi, comme ce filtre amplifie les valeurs faibles et atténue les valeurs élevées, il produit un effet de lissage de l'image.

➳ **Filtre 2 :**
L'image 1 voit juste ses valeurs devenir négatives. Sur la 2ème image, on peut observer qu'il y a des valeurs non nulles et négatives seulement lorsqu'il y a une grosse différence entre deux valeurs côte à côte (par exemple `0` et `20` dans `0,10,20`). On peut voir que le même principe est plus ou moins présent pour l'image 3. Ce principe se voit d'autant plus pour `X = [10,20,30,80,30,20,10]` on a `X * F = [0,-40,100,-40,0]`. On peut conclure que l'effet de ce filtre serait de détecter les contours ou les transitions de couleurs (grosse différence de valeur). On peut le voir sur le filtre directement, en effet la valeur central du filtre possède une plus grosse valeur que les valeurs en bordure qui sont de plus négative. Si par exemple on observe une portion d'image où trois valeurs côte à côte sont une constante a, on aurait pour la valeur centrale `a*(-1) + a*2 + a*(-1) = 0` Ce qui montre bien l'effet de "détection de variation" du filtre.

➳ **Filtre 3 :**
D'après la forme du filtre `[0,1,2]`, cela signifie que, dans une fenêtre de l'image ou du signal, seule la valeur centrale et celle de droite (pondérées respectivement par 1 et 2) contribuent au calcul, tandis que la valeur la plus à gauche n'est pas prise en compte (multipliée par 0). Puisque le filtre accorde un poids plus élevé à la valeur de droite, il accentue une tendance locale à l'augmentation dans les valeurs de l'image. Si, dans une région donnée, les valeurs augmentent de la gauche vers la droite, la contribution de la valeur pondérée par 2 sera plus forte, ce qui fera apparaître une réponse élevée. À l'inverse, si les valeurs ne changent pas ou diminuent, la réponse sera faible. Cependant, la fonction `convolution1D` appliquant le filtre de manière inversée, il s'agira donc des valeurs de gauche amplifiées et des valeurs de droite atténuées. C'est bien ce que l'on peut voir sur les 3 images ; la moins évidente à voir est l'image 3, mais on voit bien qu'à gauche, le coefficient multiplicatif par rapport à la valeur sans le filtre est de 4 et est d'environ 2 à droite (il est même inférieur à 1 pour les images 1 et 2).

3.2.3 Question 4

Établir un lien entre les fonctions `convolution1D(X,F)` et `cross_correlation1D(X',F')`.

Les fonctions `convolution1D(X,F)` et `cross_correlation1D(X',F')` sont égales pour `X=X'` et `F'` l'inverse de `F` pour une même image, la différence entre `convolution1D` et `cross_correlation1D` est que le filtre est appliquée inversement. On peut le voir en avance en regardant la définition de `convolution1D` et `cross_correlation1D` : dans `convolution1D`, les valeurs du filtre `F` sont parcourus dans la somme dans le sens direct tandis que les valeurs de `X` sont parcourus dans le sens inverse. À l'inverse pour `cross_correlation1D`, les valeurs de `X` et de `F` sont parcourus dans le sens direct toute les deux.

3.2.4 Question 5, 6, 7 et 8

- Implémenter et tester une fonction `convolution1D_padding(X,F)` qui calcule la convolution avec *padding* de X par F.
- Implémenter et tester une fonction `convolution1D_stride(X,F,k)` qui calcule la convolution *k-stride* de X par F.
- Implémenter une fonction `cross_correlation2D(X,F)` qui prend en argument une matrice X de taille (D_x, D_y) et un filtre F de taille (H_x, H_y) et renvoie la cross_correlation $X * F$ de taille $(D_x - H_x + 1, D_y - H_y + 1)$ définie par :

$$\forall (i,j), (F * H)_{ij} = \sum_{i'=1}^{H_x} \sum_{j'=1}^{H_y} X_{(i+i')(j+j')} F_{i'j'}$$

- Implémenter une fonction `applique_filtre(X, F)` qui applique le filtre `F` à l'image `X`. On affichera le résultat et l'image de départ côte à côte pour observer l'effet obtenu.

cf. fichier de code fournit

3.2.5 Question 9

Appliquer cette fonction aux différents filtres fournis dans l'énoncé et décrire leurs effets sur l'image.
On appliquera le filtre à une matrice `X_pool` obtenue par application du *pooling* à la matrice X originale (qui est trop volumineuse pour des calculs rapides).

Choix de la méthode de pooling :
Pour obtenir une image avec laquelle on pourra appliquer nos filtres avec une vitesse de calcul rapides, on utilise donc une des trois méthodes de pooling pour effectuer un pooling sur notre image d'origine et la transformer en une image de taille `(120,107)`. Voici les images produites par les trois différentes méthode de pooling (*pooling max* à gauche, *pooling moyen* au centre et *pooling median* à droite).

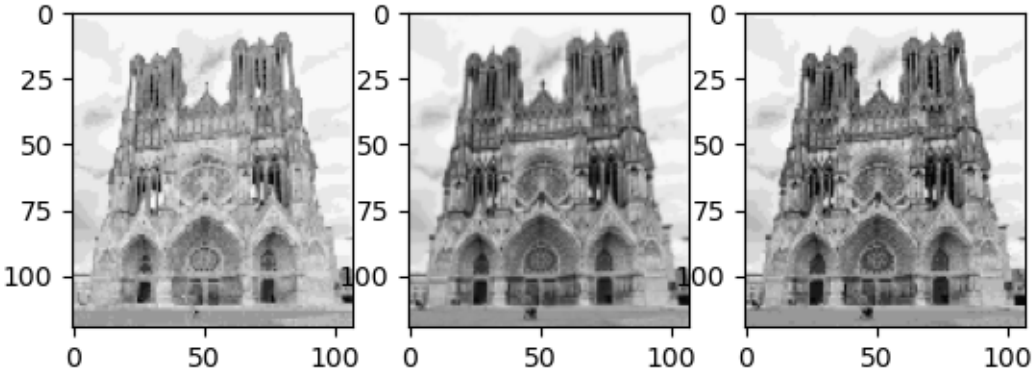

Figure 3.1 – Image renvoyée après chaque méthode de pooling

On voit que lorsque l'on applique `pooling_moy` et `pooling_median` à l'image, les images produites sont bien plus similaires entre elles que par rapport à l'image produite

par `pooling_max`. On remarque cependant que l'image produite par `pooling_moy` est plus lissée (ou plus adoucie) que celle produite par `pooling_median`. On pourrait donc dire que l'image produite par `pooling_median` est plus nette, mais elle paraît plus pixelisée que l'image produite par `pooling_moy`. Ainsi, nous choisissons de prendre l'image produite par `pooling_median` (celle de droite) car il y a déjà plus de contraste entre chaque pixel que dans l'image produite par `pooling_moy`. Cela permettra sûrement de mieux percevoir les effets des différents filtres que nous appliquerons...

Test de différents filtres fournis sur l'image :

☞ **Filtre 1 :**

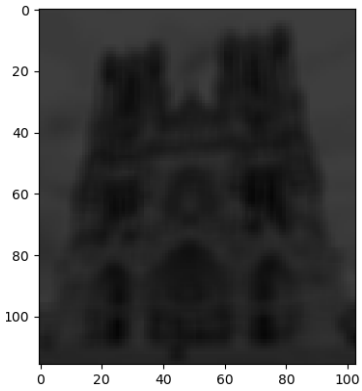

Figure 3.2 – Filtre 1 appliqués à l'image

Ce filtre réalise une moyenne sur une fenêtre assez large (puisqu'il est de taille $(5, 5)$). L'image finale apparaît donc plus floue et moins détaillée.

☞ **Filtre 2 :**

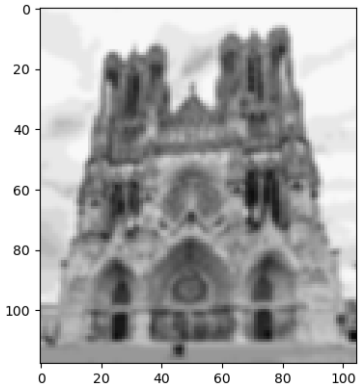

Figure 3.3 – Filtre 2 appliqués à l'image

A le même effet de floutage que le filtre n°1 ; cependant, ce filtre, lorsqu'il s'applique à une fenêtre donnée de l'image, ne s'applique pas uniformément comme le faisait le filtre n°1. La valeur centrale de cette fenêtre possède tout de même une importance légèrement plus élevée que les autres valeurs. Le résultat est une image floutée avec une transition douce entre les zones de contraste.

☞ **Filtre 3 :**

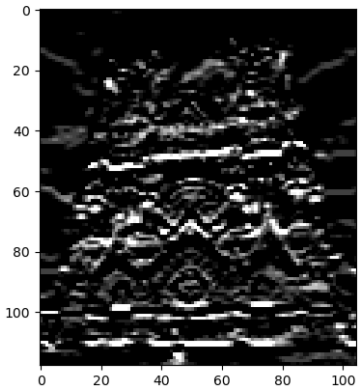

Figure 3.4 – Filtre 3 appliqués à l'image

Ce filtre ressemble, par sa forme et par ses effets, au filtre `1D` n°2. Il met en évidence les zones de l'image où l'intensité du niveau de gris change brusquement de haut en bas. "L'inversion" des couleurs (de plusieurs niveaux de gris au noir et blanc) est due au fait que, pour une faible variation des niveaux de gris, le filtre renvoie des valeurs proches de 0 (noir) et très élevées (blanc) pour des variations significatives. D'où cet effet de "détecteur de contours".

☞ **Filtre 4 :**

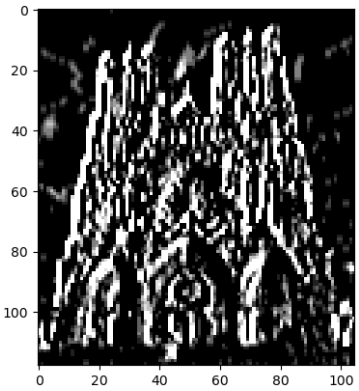

Figure 3.5 – Filtre 4 appliqués à l'image

Ce filtre accentue encore plus la détection des contours, mais cette fois-ci en fonction des variations horizontales.

☞ **Filtre 5 :**

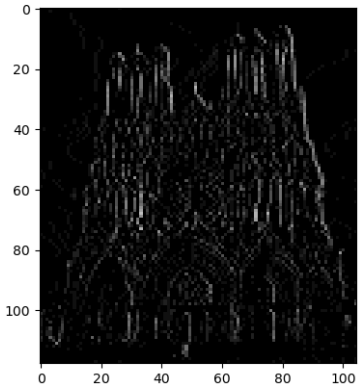

Figure 3.6 – Filtre 5 appliqués à l'image

Ce filtre regarde la différence entre la valeur centrale et son voisin de ligne à gauche. Il ne prend pas en compte les autres valeurs autour de la valeur centrale et perd donc de l'information sur la fenêtre dans laquelle il se trouve. Il ne rend donc pas compte des variations fines de l'image. Ainsi, les zones où il y a peu de différence entre les valeurs restent proches du noir, et les zones où il y aurait, en temps normal, une grande variation entre les valeurs ne sont pas significativement représentées en raison de la forme de ce filtre.

Du filtre 6 à 7 il fallait faire varier la valeur centrale entre 0 et -200.

Quand la valeur centrale est proche de 0 :
Le filtre n'introduit pas de forte soustraction entre la valeur centrale et les valeurs voisines. Ce qui donne lieu à une accentuation faible des contours.

Quand la valeur centrale devient très négative (tends vers -200) :
Le filtre soustrait de manière extrême la valeur centrale par rapport aux alentours. Ce qui cause une augmentation du contraste trop élevé et donne un effet de noir et blanc.

☞ **Filtre 6 :** La disposition du filtre la plus claire et lisible nécéssite une valeur centrale de -3/-4

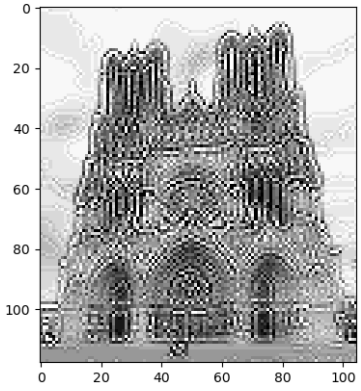

Figure 3.7 – Filtre 6 appliqués à l'image

Dans la même idée que pour le filtre 5, ce filtre prend cependant plus de valeurs en compte (ses valeurs voisines horizontalement et verticalement). Les contours de l'image sont alors un

peu plus définis. La différence totale de poids entre la valeur centrale et les valeurs voisines n'est pas élevée, ce qui donne un petit effet de flou sur l'image.

☞ **Filtre 7 :** La disposition la plus claire et lisible est autour d'une valeur centrale de -7 (c'est d'ailleurs en fait une valeur seuil puisqu'au délà, l'image devient de plus en plus noir et blanche).

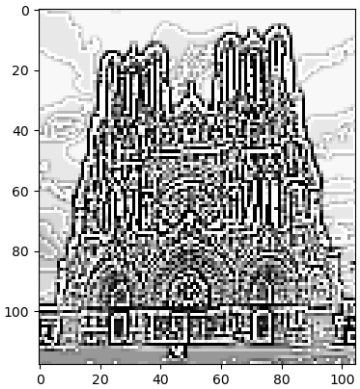

Figure 3.8 – Filtre 7 appliqués à l'image

Ce filtre accentue la différence entre la valeur centrale et les autres valeurs, ce qui donne un effet de contraste élevé. Les contours sont donc plus nets que pour les filtres précédents.

Du filtre 8 à 10 il fallait faire varier la valeur centrale entre 0 et 200.

Quand la valeur centrale est proche de 0 : L'effet du filtre se base alors sur l'importance donnée aux valeurs autour du centre. On perd alors l'information du centre de la partie de l'image. De plus, les valeurs autour du centre étant négatives ou nulles, l'image finale est en noir ou blanc et tend vers le noir total quand la valeur centrale tend vers 0.

Quand la valeur centrale est proche de 200 (en réalité une valeur proche de 20 pourrait même suffire...) : Si la valeur centrale est trop élevée, la contribution de cette valeur domine largement par rapport à celle des valeurs environnantes. Cette amplification peut créer un contraste extrême, où les détails sont trop accentués, ce qui peut conduire à une image présentant des zones très proches du blanc.

☞ **Filtre 8 :** La disposition la plus claire et lisible est autour d'une valeur centrale de 5.

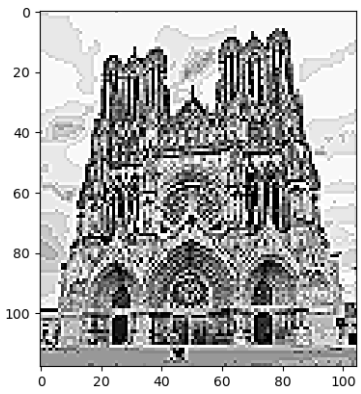

Figure 3.9 – Filtre 8 appliqués à l'image

Ce filtre démarque aussi un peu les contours en donnant plus d'importance à la valeur centrale et moins, cependant non nulle, aux valeurs environnantes. L'image semble peut-être un peu plus nette.

☞ **Filtre 9 :** La disposition la plus claire et lisible est autour d'une valeur centrale de 9.

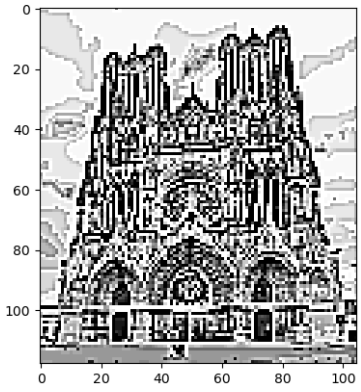

Figure 3.10 – Filtre 9 appliqués à l'image

Ce filtre a les mêmes effets que le filtre 8. Cependant, ce dernier prend aussi en compte les valeurs supérieures/inférieures et gauches/droites.

☞ **Filtre 10 :** La disposition la plus claire et lisible est autour d'une valeur centrale de 9.

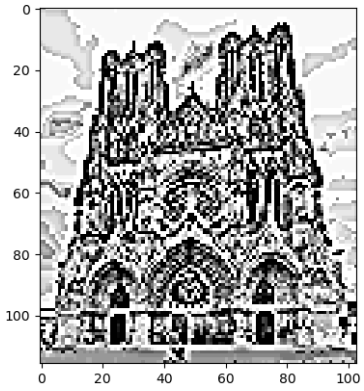

Figure 3.11 – Filtre 10 appliqués à l'image

Ce filtre, encore une fois, donne les mêmes effets que les deux filtres précédents ; il ne prend cependant pas en compte les valeurs voisines diagonalement. Il applique cependant le filtre à une échelle plus large, ce qui permet de renforcer la définition de certains détails.

3.2.6 Question 10

Il existe des réseaux de neurones, pour lesquels certaines couches intermédiaires sont de la forme :

$$Z_{i+1} = \sigma(Z_{(i)} * F_{(i)})$$

Expliquer l'intérêt que pourraient avoir ce type de réseau en termes de taille de pramètres et d'avantages qu'ils pourraient procurer.

En analysant l'expression de la formule de la `cross_correlation` en 2D, on peut observer :

➳ Que pour chaque position (i, j) sur l'image `X`, on prend une petite région (une fenêtre) de la même taille que celle du filtre, et on calcule les produits terme à terme entre cette région et le filtre, puis on somme le tout.

➳ Les indices définissant les valeurs du filtre (i', j') sont indépendants de i et j. Ce qui signifie que l'on utilise les mêmes coefficients partout sur la fenêtre et donc plus largement sur l'image entière `X`.

On peut donc en déduire que les mêmes coefficients sont appliqués partout sur l'image. Cela diminue le nombre de poids à ajuster et rend donc le modèle moins complexe.

Ce type de réseau de neurones permettrait de reconnaître des motifs partout sur une image. En effet, comme le filtre appliqué est le même partout sur l'image, le réseau peut reconnaître un motif quelle que soit la position où il se trouve dans l'image. Cela fait qu'un motif peut être déplacé n'importe où dans l'image et être facilement détecté, car il est toujours détecté de la même manière.