INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Louis BAGOT
Edouard DONZÉ
Simon DELECOURT

Rapport de projet semestriel

IMAGE DENOISING

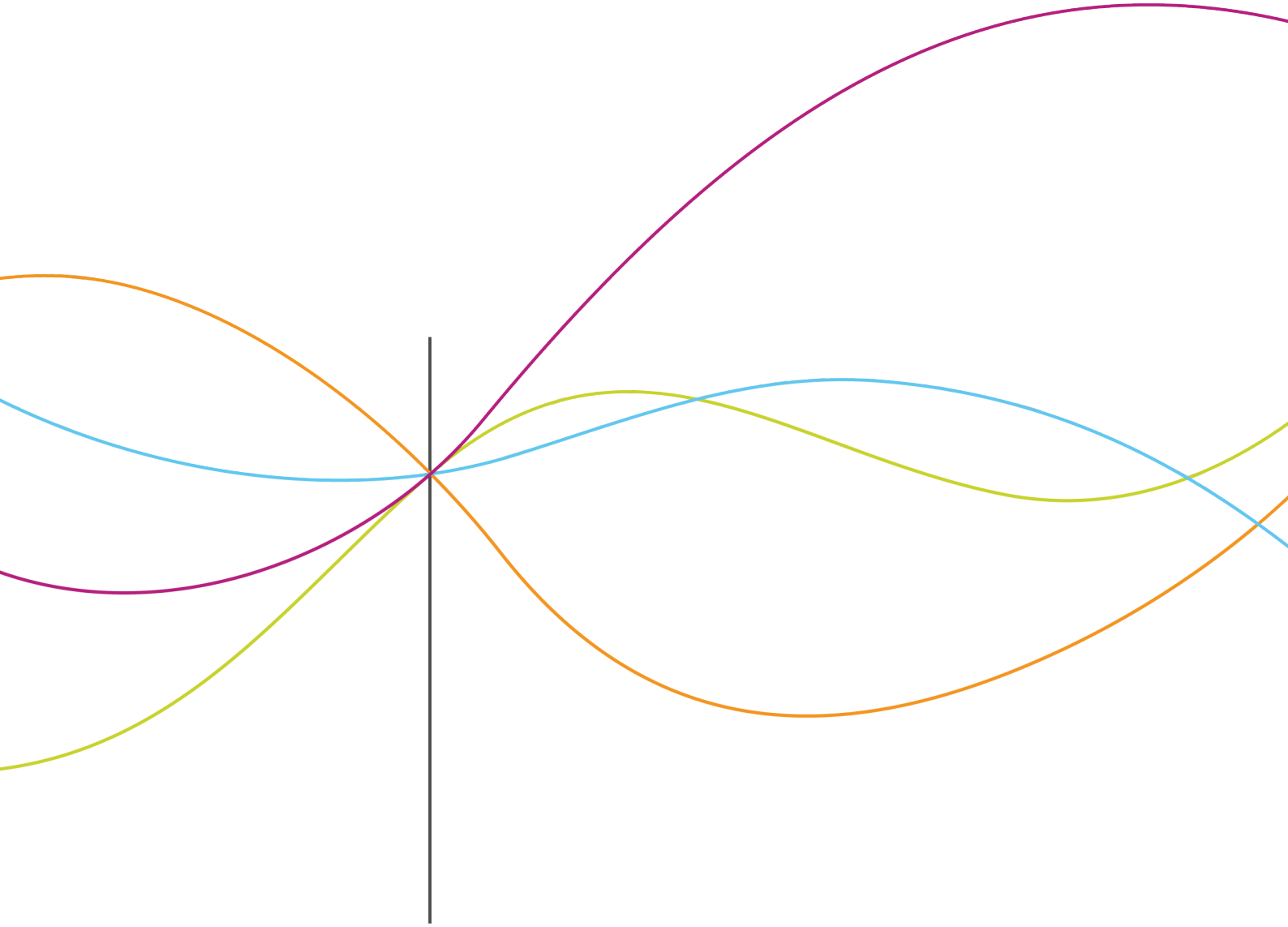


Table des matières

1 Introduction

Dans le cadre des projets semestriels en GM4, nous avons voulu faire un projet de traitement d'images. Mme Le Guyader nous a alors proposé d'étudier et d'implémenter une version de l'algorithme qui se trouve dans l'article suivant : Removal of Curtaining Effects by a Variational Model with Directional Forward Differences.
Cet article traite d'une méthode dont l'objectif est de retirer un bruit spécifique sur des images issues d'une acquisition physique. Le papier original traitait conjointement de la correction de bruit sous forme de *stripes* (bandes) et de *laminar* (zone masquée) en 3D ; mais nous avons simplifié le problème au traitement des seules *stripes* en 2D.
Au cours de ce projet, les notions mathématiques nécessaires à la compréhension du papier scientique ont d'abord été étudiées. Puis une implémentation de l'algorithme de Douglas-Rachford sous Matlab a été faite afin d'arriver à une résolution du problème mathématique posé.

2 Construction du modèle

Les images sont donc bruitées par des *stripes*, qui se manifestent comme des bandes uniformes selon l'axe y. C'est en se basant sur cette spécificité que nous avons construit le modèle. De plus, c'est en exploitant le modèle que nous avons pu développer un algorithme (basé sur celui de Douglas Rachford) afin de retirer ce bruit des images.

Les images peuvent être décrites mathématiquement comme suit :

$$f = u + s \tag{1}$$

Où f est l'image à traiter. u réprésente l'image originale et s une sorte d'image constituée uniquement de "stripes".

On definit alors l'espace de travail $C = \left\{(u,s)\,|\,f = u + s,\ u \in [0,1]^N\right\}$ et introduit la fonction h telle que $h(y_1,y_2) = \mu_1 \|y_1\|_1 + \mu_2 \|y_2\|_1$ avec $y_1 = \nabla_x u$ et $y_2 = \nabla_y s$ et en utilisant

$$\left\|\nabla_y s\right\|_1 = \sum_{i,j=1}^{nx,ny} \left|s_{i+1,j} - s_{i,j}\right|$$

$$\|\nabla_x u\|_1 = \sum_{i,j=1}^{nx,ny} \left|u_{i+1,j} - u_{i,j}\right|$$

En exploitant le fait que les stripes sont sur l'axe y et que l'image débruitée ne comporte pas de grande variations et est donc globalement lisse, on peut écrire le problème sous la forme d'un problème de minimisation :

$$\underset{z=(u,s,y_1,y_2)}{argmin}\ \{\iota_C(u,s) + h(y_1,y_2)\}$$

Pour l'utilisation de l'algorithme de Douglas-Rachford, il est intéressant de réécrire le problème. Introduisons alors deux autres espaces :

$$\begin{aligned}\widehat{C} &= \{(u,y_1)\,|\,y_1 = \nabla_x u\}\\ \widetilde{C} &= \{(u,y_2)\,|\,y_2 = \nabla_y s\}\end{aligned} \tag{2}$$

Le problème se réécrit alors comme suit :

$$\underset{z=(u,s,y_1,y_2)}{inf}\ \underbrace{\iota_C(u,s) + h(y_1,y_2)}_{G_1(z)} + \underbrace{\iota_{\widehat{C}}(u,y_1) + \iota_{\widetilde{C}}(s,y_2)}_{G_2(z)}$$

Autrement dit :

$$\begin{aligned}&\underset{z=(u,s,y_1,y_2)}{inf} && G_1(z) + G_2(z)\\ &avec && G_1(z) = \iota_C(u,s) + h(y_1,y_2)\\ & && G_2(z) = \iota_{\widehat{C}}(u,y_1) + \iota_{\widetilde{C}}(s,y_2)\end{aligned} \tag{3}$$

L'intérêt de cette réécriture est de créer deux fonctions d'une même variable, composées de fonctions indépendantes. Cela permettra de simplifier fortement le calcul des opérateurs *prox* que nous allons définir ci-dessous.

3 Algorithme de Douglas-Rachford

L'algorithme a pour objectif de résoudre un problème sous la forme suivante :

$$\min_{x\in\mathbb{R}^N} f_1(x) + f_2(x) \tag{4}$$

Algorithm 1 Algorithme de Douglas-Rachford

for n=0,1... **do**
$x_n = prox_{\gamma f_2}\, y_n$
$\lambda_n \in [\epsilon, 2-\epsilon]$
$y_{n+1} = y_n + \lambda_n(prox_{\lambda f_1}\,(2x_n - y_n) - x_n)$
end for

Au vu de cet algorithme il va être nécessaire de calculer $prox_{\lambda G_1}(z)$ et $prox_{\gamma G_2}(z)$.
Pour une fonction convexe quelconque la définition est que $prox_f(x) := \underset{u}{argmin}\left(f(u) + \frac{1}{2}\|u-x\|_2^2\right)$.
Cela implique en plus une propriété intéressante sur l'addition de fonctions de variables indépendantes, que nous illustrons directement sur G_1 :

$$G_1\,(z=(u,s,y_1,y_2)) = \iota_C\,(u,s) + h\,(y_1,y_2)$$
$$prox_{\gamma G_1}\,(z) = \begin{bmatrix} prox_{\gamma\iota_C}\,(u,s) \\ prox_{\gamma h}\,(y_1,y_2) \end{bmatrix}$$

Le raisonnement est le même pour G_2. Nous commençons par calculer le *prox* de G_2 ; on a d'abord, par définition :

$$\begin{aligned} prox_{\iota_{\widetilde{C}}}(u,y_1) &= \underset{(\widetilde{u},\widetilde{y_1})}{argmin}\,\iota_{\widetilde{C}}(\widetilde{u},\widetilde{y_1}) + \frac{1}{2}\|(\widetilde{u},\widetilde{y_1}) - (u,y_1)\|_2^2 \\ &= pro\,j_{\widetilde{C}}(u,y_1) \\ &= \underset{(\widetilde{u},\widetilde{y_1})\in\widetilde{C}}{argmin}\,\frac{1}{2}\|(\widetilde{u},\widetilde{y_1}) - (u,y_1)\|_2^2 \end{aligned}$$

Sous la contrainte $\widetilde{y_1} = \nabla_x\widetilde{u}$, que l'on peut réecrire avec l'opérateur linéaire K_1 sous la forme : $\widetilde{y_1} = K_1\widetilde{u}$, nous avons donc la valeur suivante pour le premier élément du prox :

$$\begin{aligned} &\underset{\widetilde{u}}{argmin}\,\frac{1}{2}\|(\widetilde{u},K_1\widetilde{u}) - (u,y_1)\|_2^2 \\ &= \underset{\widetilde{u}}{argmin}\,\frac{1}{2}\|\widetilde{u}-u\|_2^2 + \frac{1}{2}\|K_1\widetilde{u} - y_1\|_2^2 \end{aligned}$$

D'où, en dérivant en fonction de $\widetilde{u}$:

$$\begin{aligned} &\widetilde{u} - u + K_1^T(K\widetilde{u} - y_1) = 0 \\ \Leftrightarrow\ &\widetilde{u} = (I + K_1^TK_1)^{-1}(u + K_1^Ty_1) \end{aligned}$$

On trouve finalement $\widetilde{y_1}$ simplement par $\widetilde{y_1} = K_1\widetilde{u}$. En procédant exactement de la même manière on calcule : $prox_{\widetilde{C}}(s,y_2) = \underset{(\widetilde{s},\widetilde{y_2})}{argmin}\,\iota_{\widetilde{C}}(\widetilde{s},\widetilde{y_2}) + \frac{1}{2}\|(\widetilde{s},\widetilde{y_2}) - (s,y_2)\|_2^2$ en reecrivant la contrainte $\widetilde{y_2} = \nabla_y\widetilde{s}$ sous la frome : $\widetilde{y_2} = K_2\widetilde{s}$ on obtient un resultat similaire au premier calcul

$$\widetilde{s} = (I + K_2^TK_2)^{-1}(s + K_2^Ty_2)$$

Passons maintenant au calcul de $prox_{G_1}$ et de la même manière calculons indépendamment les *prox*.

$$\begin{aligned} prox_{\iota_C}(u,s) &= \underset{(\widetilde{u},\widetilde{s})}{argmin}\, \iota_C(\widetilde{u},\widetilde{s}) + \frac{1}{2}\|(\widetilde{u},\widetilde{s}) - (u,s)\|_2^2 \\ &= proj_C(u,s) \\ &= \underset{(\widetilde{u},\widetilde{s})\in C}{argmin} \frac{1}{2}\|(\widetilde{u},\widetilde{s}) - (u,s)\|_2^2 \end{aligned}$$

Donc sous la contrainte $f = \widetilde{u} + \widetilde{s},\ u \in [0,1]^N$.

Soit, en remplaçant $\widetilde{s} = f - \widetilde{u}$, $\widetilde{u}$ est donné par :

$$\begin{aligned} &\underset{\widetilde{u}\in[0,1]^N}{argmin}\ \|\widetilde{u} - u\|_2^2 + \|(f-\widetilde{u}) - s\|_2^2 \\ &= \underset{\widetilde{u}\in[0,1]^N}{argmin} \left\|\widetilde{u} - \tfrac{u+f-s}{2}\right\|_2^2 \qquad \textit{Par developpement des normes} \\ &= proj_{\widetilde{u}\in[0,1]^N}\left(\frac{u+f-s}{2}\right) \end{aligned}$$

Finalement, pour $\widetilde{u_{ij}}$ chaque coefficient (/pixel) de $\widetilde{u}$, on a :

$$\widetilde{u}_{ij} = min\left(max\left(0, \frac{u_{ij} + f_{ij} - s_{ij}}{2}\right), 1\right)$$

et $\widetilde{s}_{ij} = f_{ij} - \widetilde{u}_{ij}$

Enfin pour le calcul de $prox_h$ nous allons utiliser une propriété du $prox$, à savoir :

$$prox_{\lambda\|.\|_1}(x) = \begin{cases} x - \lambda\frac{x}{\|x\|_1} & si\ \|x\|_1 > \lambda \\ 0 & sinon \end{cases}$$

On rappelle que $h(y_1,y_2) = \mu_1\|y_1\|_1 + \mu_2\|y_2\|_1$ ainsi, en utilisant l'indépendance des deux variables :

$$prox_h(y_1,y_2) = \begin{bmatrix} prox_{\mu_1\|.\|_1}(y_1) \\ prox_{\|.\|_1}(y_2) \end{bmatrix} = \begin{bmatrix} \begin{cases} y_1 - \mu_1\frac{y_1}{\|y_1\|_1} & si\ \|y_1\|_1 > \mu_1 \\ 0 & sinon \end{cases} \\ \begin{cases} y_2 - \frac{y_2}{\|y_2\|_1} & si\ \|y_2\|_1 > 1 \\ 0 & sinon \end{cases} \end{bmatrix}$$

Nous avons donc finalement tous les éléments nécessaire au calcul des $prox$, et donc à l'implémentation de l'algorithme.

4 Implémentation et Résultats

Nous avons implémenté sous Matlab l'algorithme de Douglas-Rachford (présenté précedemment) au problème mathématique posé.

Cet algorithme présente plusieurs hyper-paramètres :

— n, le nombre d'itérations. Nous avons empiriquement déterminé qu'une valeur de n entre 500 et 1000 fournissait des résultats corrects, et que ces derniers ne s'amélioraient presque plus pour des valeurs plus grandes.
— μ, dans la formule de h. Ce paramètre permet de donner du poids aux éléments à minimiser.
Pour $\mu > 1$, on privilégie le paramètre qui le porte, c'est-à-dire le gradient de u selon x. En d'autres termes, on favorise la régularité de ce terme.
Ituitivement, on sent qu'il vaudra mieux donner plus de poids à l'aspect invariable de s sur l'axe y. L'expérience confirme ceci et finalement, 0.5 est la valeur retenue.

— λ est le pas d'actualisation de la variable y, 0.5 a empiriquement semblé fournir de meilleurs résultats. Dans ce cas, nous avons choisi les valeurs qui semblaient minimiser la valeur du gradient de s selon y issus du problème.

Les résultats obtenus sont satisfaisants. Pour une image arbitraire qui a été artificiellement (et idéalement) *stripée* dans un objectif de test, cela donne :

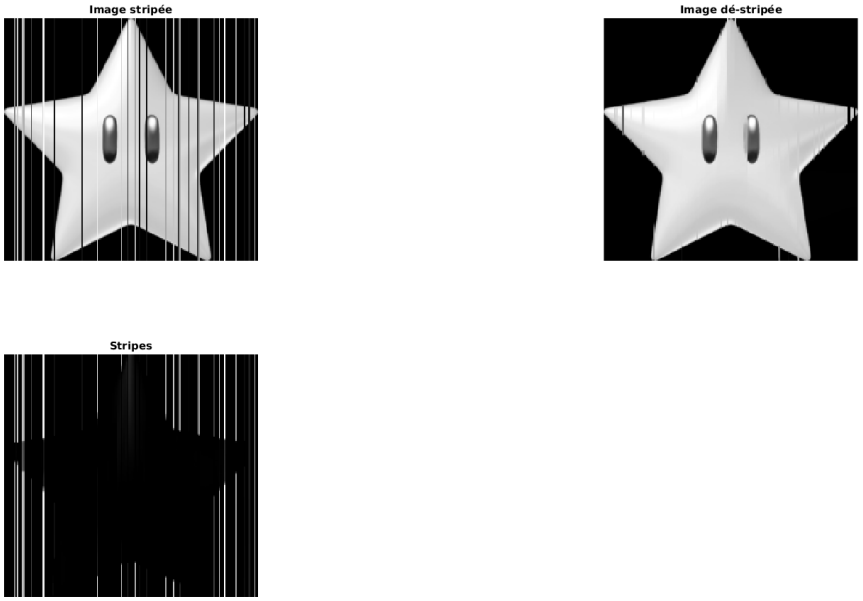


FIGURE 1 – Exemple de résultat 1

On remarque que l'image obtenue est presque parfaite. Il reste malgré tout quelques stripes sur les branches de l'étoile, de plus on peut encore voir les contours de l'étoile de l'image des *stripes*. Cela indique que notre algorithme n'a pas "compris" l'idée de *stripes* ; on devine que c'est bien cette formation qui minimise les gradients comme demandé.

Puis pour une image issue de l'article, nous obtenons le résultat suivant :

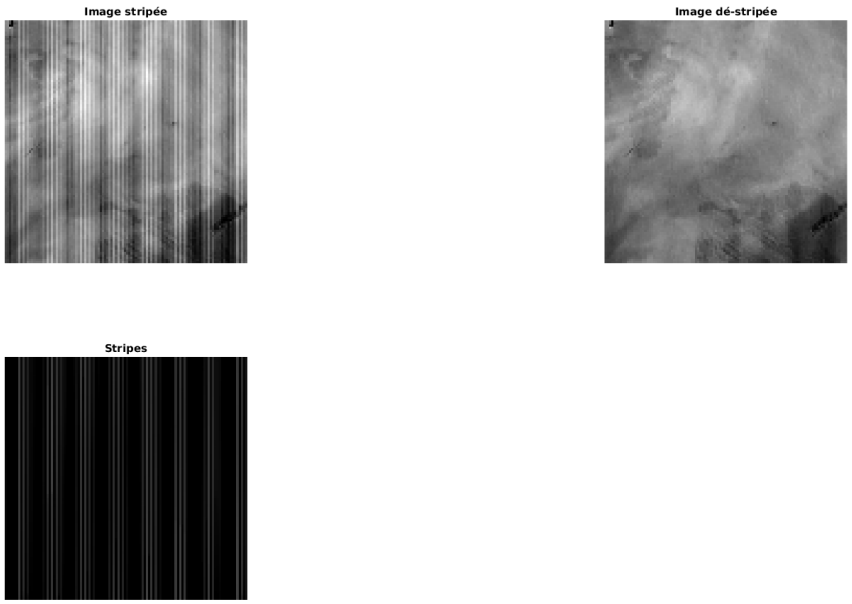


FIGURE 2 – Exemple de résultat 2

Pour cette image, très peu de *stripes* restent après l'application de l'algorithme. L'image devient beaucoup plus exploitable pour un usage quelconque. On peut également apercevoir sur l'image des *stripes* que celles-ci ont bien été en grande partie extraites.
La qualité du résultat est cependant, dans ce cas, bien plus dure à estimer puisque nous ne comprenons pas l'image d'origine, contrairement à l'exemple précédent où il est très simple de deviner le résultat attendu.

5 Une autre approche

Nous avons vu, dans les sections précédentes, que la construction d'un modèle mathématiques bien pensé et avec l'aide d'algorithme de résolution de problème convexe, nous pouvions arriver à un résultat très satisfaisant.

L'inconvénient de telles techniques de résolution est qu'elles sont spécifiques au problème donné initialement. Cela signifie que si par ailleurs on veut traiter par la même méthode un autre type de bruit, le modèle mathématique doit être reconsidéré et peut même être complètement différent si le bruit n'a pas la même forme (e.g. un bruit Gaussien).

Nous allons voir ici une approche par Machine Learning qui peut résoudre le même problème, c'est à dire retirer les stripes d'une image. Nous verrons que cette technique pourra se généraliser très facilement à d'autres types de problèmes.

Les techniques de machine learning et de réseaux de neurones profonds connaissent un intérêt exponentiel depuis une petite dizaine d'années. Les recherches sur ces techniques ont permis de résoudre des problèmes de plus en plus complexes, notamment sur le traitement et la reconnaissance d'image. De plus, ayant fait un projet semestriel d'introduction à certaines techniques de Machine Learning, nous avons eu envie de poursuivre notre apprentissage en les appliquant sur ce problème. Nous avions la conviction que cela était envisageable et après quelques recherches nous avons trouvé une technique qui semblait s'adapter à notre problème.

La technique proposée est celle de l'Autoencoder. Un autoencoder fait partie de la famille des Réseaux de Neurones Artificiels. L'idée générale des autoencoders est de réduire à chaque couche du réseau la dimension de l'espace de la couche précédentes jusqu'à un certain seuil, puis d'augmenter à nouveau la dimension des couches afin d'obtenir le même espace qu'en entrée. Cela a comme intérêt de réduire la quantité d'information contenue dans l'image afin de ne garder que l'essentiel et ainsi supprimer le bruit puis de revenir à l'espace

voulu. Le schéma ci-dessous illustre cette architecture particulière :

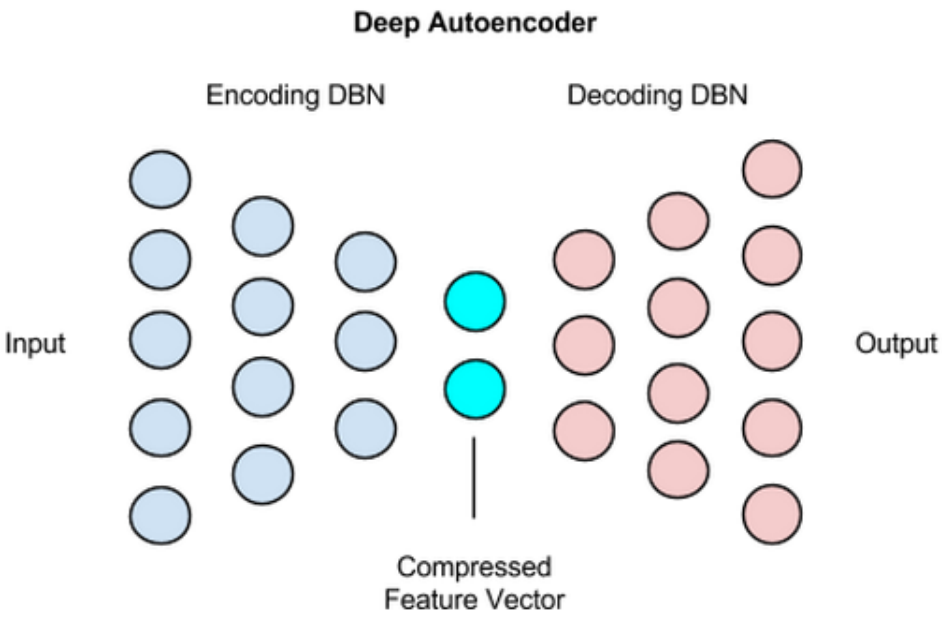


image : https://deeplearning4j.org/deepautoencoder

Il est à noter que les topologies de réseaux utilisés peuvent varier. On peut utiliser des couches Convolutional ou Dense. Dans notre cas nous travaillons sur des images, nous utiliserons donc des couches Convolutional car c'est la topologie la plus adaptée.

Pour implémenter la méthode nous avons utilisé Python et la librairie Keras. Les images utilisées, à titre d'exemple, sont des chiffres écrits à la main venant de la célèbre base de données MNIST.

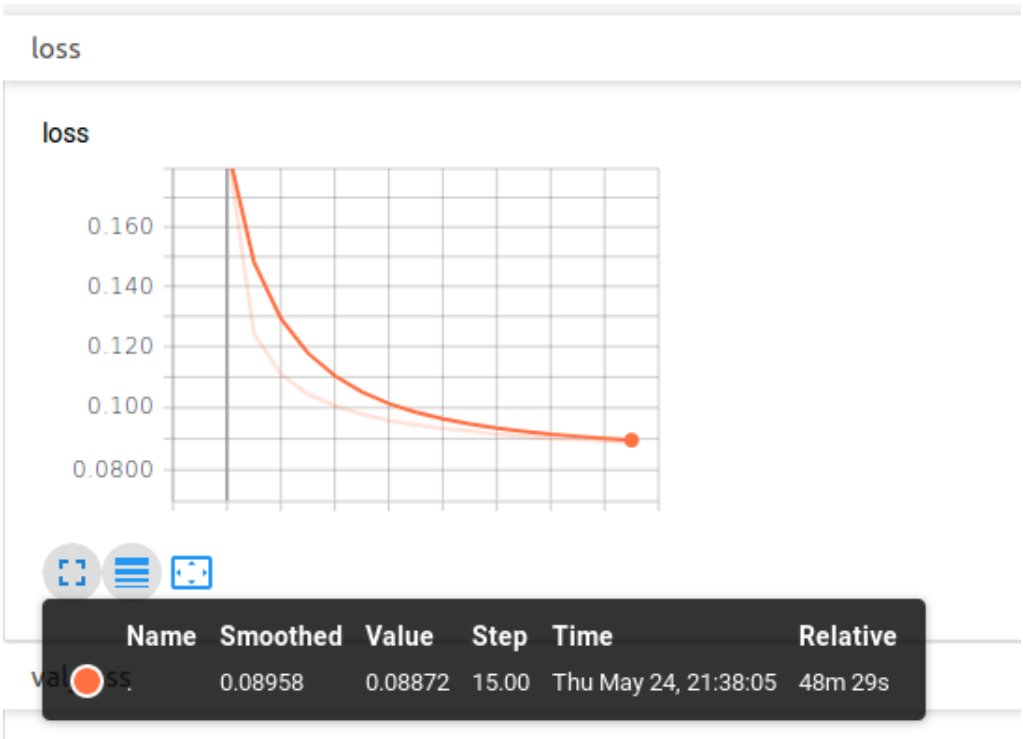
La stratégie adoptée pour entrainer l'autoencoder a été de produire des *stripes* sur les images. Puis, comme pour toutes méthodes d'apprentissage supervisé, nous donnons en entrée du réseau les images *stripées*, et adaptons les poids pour que l'image qu'il produit en sortie se raproche de l'image non *stripée*. Finalement, pour tester l'autoencoder, on lui demande de produire des résultats pour des images qu'il n'a pas encore vues.

L'architecture de l'autoencoder ainsi que la tactique d'entrainement est présenté dans l'extrait du code ci-dessous :

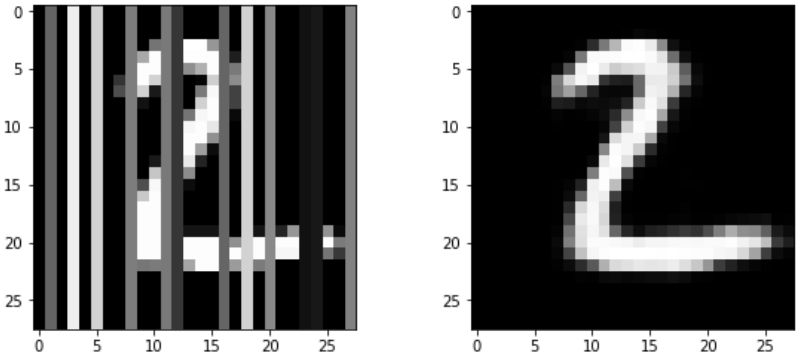
```
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, ZeroPadding2D
from keras.models import Model
from keras.callbacks import TensorBoard


(y_train, _), (y_test, _) = mnist.load_data()

x_train = np.copy(y_train)
x_test = np.copy(y_test)
x_train = stripify(x_train)
x_test = stripify(x_test)

input_img = Input(shape=(x_train[0].shape[0], x_train[0].shape[1], 1))

```

```
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (7, 7, 28)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

autoencoder.fit(x_train, y_train,
                epochs=100,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, y_test),
                callbacks=[TensorBoard(log_dir='/tmp/tb_stripes-mnist', histogram_freq=0,
                 ↪  write_graph=False)])

score = autoencoder.evaluate(x_test, y_test, verbose=0)
```

Voici la courbe montrant l'évolution de la fonction coût à chaque "epoch" (chaque tour d'entrainement). Cet entrainement a nécessité à peu près 1h pour 15 "épochs" avec une base de données d'entrainement de 50 000 images.

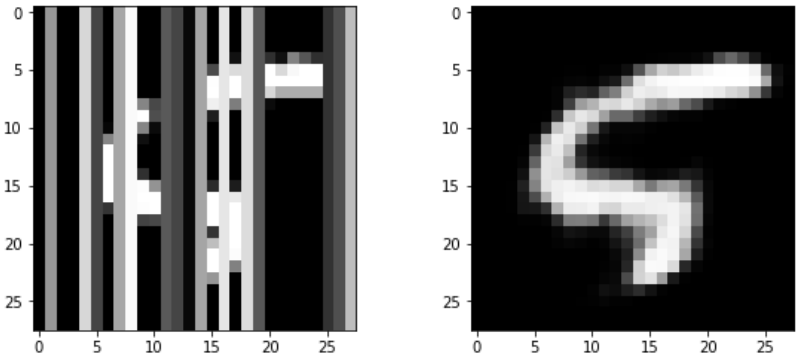


Les résultats obtenus avec les images de la phase de test (c'est à dire des images que le réseau n'a jamais vu) sont les suivants :



(a) Image stripée (b) Image dé-stripée

FIGURE 3 – Exemple de résultat



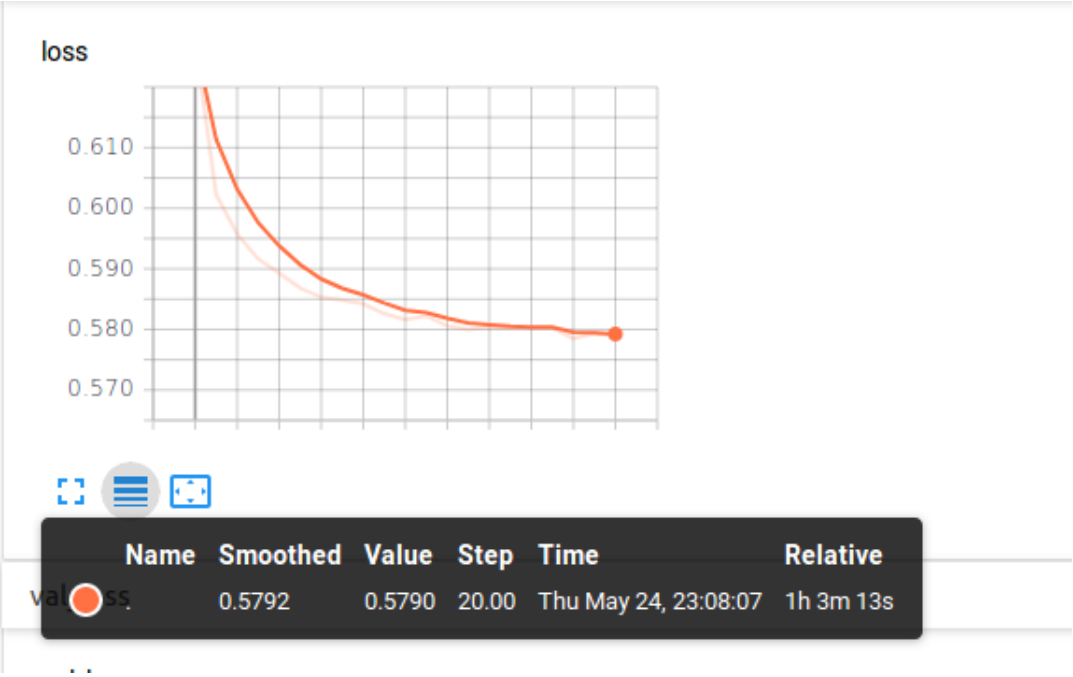
(a) Image stripée (b) Image dé-stripée

FIGURE 4 – Exemple de résultat

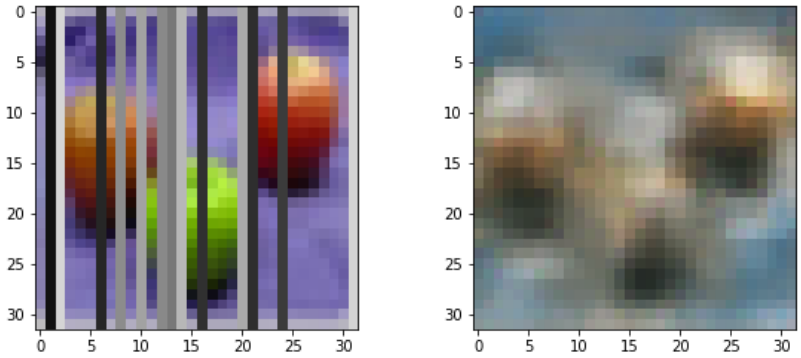
On peut remarquer que les résultats sont très satisfaisants. L'image obtenue est quasiment parfaite à un petit nombre de pixels près. Les stripes sont retirées entièrement.

Malgré les résultats très encourageants de cette méthode, il est toutefois possible de la critiquer. En effet, l'entraînement c'est fait sur une base de données de chiffres dont la diversité n'est pas très importante. Ainsi on peut se demander si le réseau sait généraliser sur d'autres types d'image. On peut se poser la question de ce qu'il a vraiment appris. Peut-être qu'il a appris à reconnaître partiellement le chiffre et à corriger l'image pour qu'elle ressemble encore plus à un chiffre, plus qu'il n'aurait appris à reconnaître une *stripe* et à la retirer d'une image.

Pour illustrer ce phénomène nous avons alors reproduit cette phase d'entrainement sur une autre base de données connue : la base cifar-100. Cette base est constituée d'images couleur et est généralement utilisée pour de la classification, puisqu'elle catégorisée en 100 *features*. Elle offre ainsi une plus grande diversité que la base MNIST. Après quasiment le même temps d'entrainement et le même nombre d'epoch, on pourra à nouveau remarquer la diminution de la valeur de la fonction coût au cours du temps, pour atteindre un plateau.

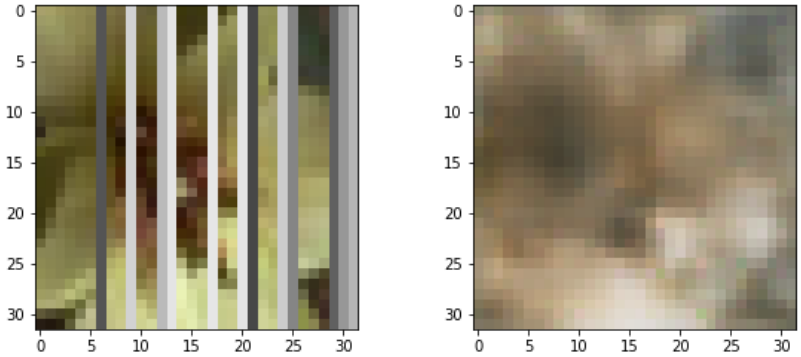


Les résultats obtenus sont les suivants :



(a) Image stripée (b) Image dé-stripée

FIGURE 5 – Exemple de résultat



(a) Image stripée (b) Image dé-stripée

FIGURE 6 – Exemple de résultat

On peut constater que les *stripes* ont bien été retirées, ce qui est intéressant comme résultat. Par contre le fond n'a pas été retranscrit comme il devrait. On peut apporter plusieurs raisons à cela. Premièrement : un

manque d'entraînement. En effet la valeur de la fonction coût semblait encore diminuer. Ainsi après une plus longue phase d'entrainement, peut-être que l'autoencoder aurait vraiment appris à reconnaitre les stripes et ne pas toucher au fond comme il semblait l'avoir fait pour MNIST. Deuxièment, on peut mettre en cause la structure de l'autoencoder qui réduit peut-être trop l'espace et donc perd tous les détails de l'image. Mais ce ne sont que des hypothèses, il nous manque de la théorie sur les réseaux de neurones pour avoir une analyse plus fine.

Finalement l'autoencoder a montré ses limites. De plus ici les images sont de très petite taille (32*32), entrainer l'autoencoder sur des images réelles de taille plus importante (e.g. 500*500 ou plus) sera beaucoup plus long et plus lourd en mémoire. Construire un modèle mathématiques spécifiquement pour le problème peut alors s'avérer plus intéressant. Cette phase d'entraînement peut rester toutefois envisageable si on a les ressources matérielles (et à condition d'avoir trouvé la topologie de réseau). D'autant plus qu'après l'entraînement, produire l'image corrigée est très rapide, cela n'est que l'enchaînement de plusieurs multiplications matricielles.

source : https://blog.keras.io/building-autoencoders-in-keras.html

6 Conclusion

C'était, pour chacun de nous, la première expérience de traitement d'image ainsi que la première utilisation concrète des notions abordées en Différences Finies. Ce projet a été particulièrement intéressant car il nous a permis de nous rendre compte que notre niveau en mathématiques nous permet à présent de résoudre des problèmes qui peuvent sembler très complexes ; la formulation mathématique permet en effet de briser le flou qui entoure le problème.
Le seul point qui contredit ceci est que nous n'avons pas pu étudier et comprendre les modèles mathématiques cachés derrière l'algorithme de Douglas-Rachford et les opérateurs de *prox*. Cependant cela ne gache en rien le plaisir que nous avons eu à l'obtention des résultats.
Un autre point particulièrement intéressant a été l'introduction concrète à des problèmes d'optimisation informatique : l'algorithme prenait plusieurs secondes à tourner, malgré des images en noir et blanc de petite taille. Cela nous a indirectement introduit aux idées de calcul parallèle et à plusieurs méthodes d'optimisation, dont nous comprenons à présent l'utilisation avec bien plus de clarté. De la même manière, nous utilisions des matrices de très grande taille et nous avons donc été introduits à la gestion de ce type de problèmes mémoire, bien qu'à petite dose. Finalement, ce projet a couvert bien plus que ce que nous n'imaginions, et s'est révélé très bon enseignant.